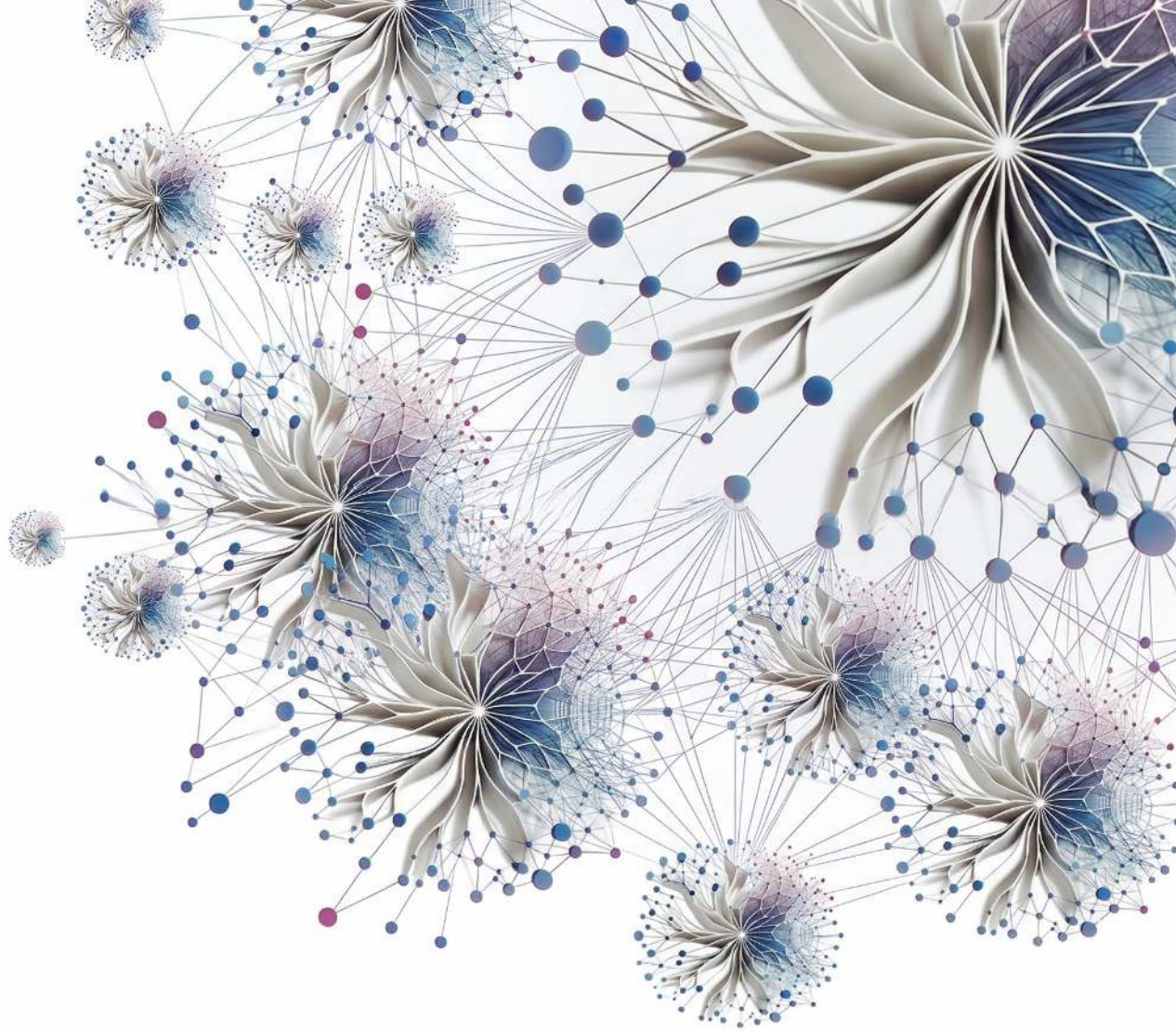


At Scale

Machine Learning Systems

Vijay Janapa Reddi

Machine Learning Systems at Scale

Machine Learning Systems at Scale

Prof. Vijay Janapa Reddi
School of Engineering and Applied Sciences
Harvard University

July 3, 2026

Table of contents

Welcome **xv**

Dedication **xvii**

Foreword **xix**

Author's Note **xxi**

Preface **xxiii**
Who This Book Is For . . . xxiii
Why a Fleet-Level Textbook . . . xxiii
What This Book Covers . . . xxiv
Suggested Reading Paths . . . xxvi
Conventions . . . xxvi
Companion Resources . . . xxvii
Prerequisites . . . xxvii
Beyond This Book . . . xxviii
Using This Book in a Course . . . xxviii
Copyright and Licensing . . . xxviii

Acknowledgments **xxix**
Origins . . . xxix
Collaborators and Colleagues . . . xxix
Support . . . xxix

A Note on AI Assistance **xxxi**

Notation **xxxiii**
The Iron Law of ML Systems . . . xxxiii
The Degradation Equation . . . xxxiv
The Energy Corollary . . . xxxiv
Deep Learning Notation . . . xxxv
Performance, Serving, and Memory Notation . . . xxxv
Units and Precision . . . xxxvi
Quick Reference: Resolving Collisions . . . xxxvii
Distributed Systems Notation . . . xxxvii

Main Matter

Chapter 1 Introduction **3**
1.1 The Scale Moment . . . 4
1.2 The Fleet Stack: A Hierarchy of Architecture . . . 8
1.3 AI Scaling Laws . . . 12
1.4 Constraints of Scale . . . 19

1.5 The C^3 Taxonomy: Foundations of Scale . . . 22
1.6 Foundational Concepts . . . 25
1.7 The Structure of This Textbook . . . 28
1.8 Fallacies and Pitfalls . . . 29
1.9 Summary . . . 30

Part I The Fleet

Fleet Principles **35**

Chapter 2 Compute Infrastructure **39**
2.1 The Infrastructure Walls . . . 40
2.2 Accelerator Spectrum . . . 40
2.3 The Memory Wall . . . 49
2.4 The Node . . . 74
2.5 The Rack . . . 88
2.6 The Pod . . . 92
2.7 Infrastructure Technologies That Move the Walls . . . 95
2.8 Fallacies and Pitfalls . . . 97
2.9 Summary . . . 99

Chapter 3 Network Fabrics **103**
3.1 The Synchronization Backbone . . . 104
3.2 How ML Networking Inverts Data-Center Assumptions . . . 106
3.3 Level 1: Wire and Link . . . 108
3.4 Level 2: Transport and the Performance Model . . . 110
3.5 Level 3: Switch and Topology . . . 116
3.6 Level 4: Fabric Behavior (Congestion, Routing) . . . 124
3.7 Level 5: Cluster Design and Case Studies . . . 129
3.8 Network Virtualization . . . 130
3.9 Monitoring and Debugging . . . 131
3.10 Network Technologies That Move the Ceiling . . . 133
3.11 Fallacies and Pitfalls . . . 135
3.12 Summary . . . 136

Chapter 4 Data Storage **139**
4.1 The Fuel Line . . . 140
4.2 How ML Workloads Invert Storage Assumptions . . . 141
4.3 The ML Storage Hierarchy . . . 146
4.4 The Data Pipeline Equation . . . 162
4.5 GPUDirect Storage and the CPU Bypass . . . 170
4.6 Storage Economics . . . 173
4.7 Checkpoint Storage . . . 177
4.8 Retrieval Infrastructure: Vector Indexes . . . 181
4.9 The Synthetic Fuel Line . . . 182
4.10 Fallacies and Pitfalls . . . 183
4.11 Summary . . . 184

Part II Distributed ML

Distributed ML Principles **189**

Chapter 5 Distributed Training **193**
5.1 Why Distribution Is Necessary . . . 194
5.2 The Distributed Training Step . . . 197

5.3 Data Parallelism . . . 198
5.4 Scaling Efficiency and Convergence . . . 211
5.5 Model Parallelism . . . 223
5.6 FP8 for distributed training . . . 235
5.7 Hybrid Parallelism . . . 236
5.8 Multi-Model Training: RLHF and Alignment . . . 241
5.9 Parallelism Strategy Comparison . . . 245
5.10 From Principles to Systems . . . 246
5.11 Fallacies and Pitfalls . . . 247
5.12 Summary . . . 250

Chapter 6 Collective Communication **253**
6.1 From Parallelism to Communication Patterns . . . 254
6.2 Mapping the Terrain: Network Performance Modeling . . . 258
6.3 Choosing the Vehicle: Collective Operation Primitives . . . 263
6.4 Engineering the Flow: AllReduce Algorithms . . . 268
6.5 Hierarchical Communication . . . 277
6.6 Gradient Compression Under Bandwidth Scarcity . . . 282
6.7 The Communication Library Landscape . . . 288
6.8 Communication-Computation Overlap . . . 293
6.9 Fallacies and Pitfalls . . . 295
6.10 Summary . . . 296

Chapter 7 Fault Tolerance **301**
7.1 Failure Analysis at Scale . . . 302
7.2 Hardware Fault Taxonomy . . . 318
7.3 Software Faults . . . 327
7.4 Check-and-Verify: Defending Against Silent Data Corruption . . . 329
7.5 Fault Injection Tools and Frameworks . . . 330
7.6 Checkpointing: Preserving Progress . . . 333
7.7 Failure Detection and Recovery . . . 341
7.8 Elastic Recovery . . . 346
7.9 Serving Fault Tolerance . . . 350
7.10 Graceful Degradation . . . 354
7.11 Distributed Debugging and Observability . . . 358
7.12 Case Studies . . . 363
7.13 Fallacies and Pitfalls . . . 366
7.14 Summary . . . 369

Chapter 8 Fleet Orchestration **373**
8.1 The Scheduling Problem . . . 374
8.2 Orchestration Paradigms . . . 384
8.3 Topology-Aware Scheduling . . . 393
8.4 Elastic Scheduling . . . 398
8.5 Cost Optimization . . . 402
8.6 Custom ML Schedulers . . . 407
8.7 Serving Resource Management . . . 411
8.8 Multi-Tenancy and Quotas . . . 417
8.9 Debugging Cluster Utilization . . . 422
8.10 Fallacies and Pitfalls . . . 426
8.11 Summary . . . 428

Part III Deployment at Scale

Deployment-at-Scale Principles **431**

Chapter 9 Performance Engineering **433**
9.1 The Memory Wall and Roofline Diagnosis . . . 434
9.2 Serving Regimes: Batch Size and Prefill-Decode . . . 442
9.3 Operator Fusion and Kernel Engineering . . . 445
9.4 Precision Engineering . . . 449
9.5 Graph Compilation . . . 453
9.6 Algorithmic Performance Transformations . . . 460
9.7 Communication-Computation Overlap . . . 461
9.8 System Profiling . . . 464
9.9 Measurement at Scale . . . 471
9.10 The Optimization Playbook: A 70B LLM Case Study . . . 473
9.11 Fallacies and Pitfalls . . . 477
9.12 Summary . . . 479

Chapter 10 Inference at Scale **483**
10.1 The Economics and Architecture of Inference . . . 484
10.2 Serving Architecture Dimensions . . . 491
10.3 Batching Strategies at Scale . . . 494
10.4 Memory and Decode-Time Management . . . 513
10.5 Model Sharding for Inference . . . 528
10.6 Load Balancing and Request Routing . . . 538
10.7 Multi-Tenancy and Isolation . . . 549
10.8 Autoscaling and Global Infrastructure . . . 555
10.9 Weight Quantization for Serving . . . 566
10.10 Case Studies . . . 572
10.11 Fallacies and Pitfalls . . . 578
10.12 Summary . . . 579

Chapter 11 Edge Intelligence **581**
11.1 The Edge Learning Paradigm . . . 582
11.2 Design Constraints . . . 588
11.3 Model Adaptation . . . 598
11.4 Data Efficiency . . . 607
11.5 Federated Learning: Algorithms . . . 612
11.6 Federated Learning: Systems at Scale . . . 618
11.7 Continual Adaptation Under Edge Budgets . . . 627
11.8 Production Integration . . . 629
11.9 Putting the Pillars Together . . . 634
11.10 Engineering Challenges and Mitigations . . . 636
11.11 Fallacies and Pitfalls . . . 643
11.12 Summary . . . 644

Chapter 12 ML Operations at Scale **647**
12.1 From Single-Model to Platform Operations . . . 648
12.2 The N-Models Problem . . . 649
12.3 Fleet Economics and Utilization . . . 660
12.4 Multi-Model Management . . . 670
12.5 CI/CD for ML at Scale . . . 677
12.6 Monitoring at Scale . . . 697
12.7 Platform Engineering . . . 707
12.8 Feature Store Operations . . . 721
12.9 Organizational Patterns . . . 728
12.10 Case Studies . . . 730
12.11 Production Debugging and Incident Response . . . 733
12.12 Fallacies and Pitfalls . . . 739
12.13 Summary . . . 740

Part IV The Responsible Fleet

Responsible Fleet Principles **745**

Chapter 13 Security & Privacy **747**
13.1 The Expanded Attack Surface . . . 748
13.2 Learning from Security Breaches . . . 752
13.3 Systematic Threat Analysis and Risk Assessment . . . 754
13.4 Model-Specific Attack Vectors . . . 759
13.5 Hardware-Level Security Vulnerabilities . . . 773
13.6 When ML Systems Become Attack Tools . . . 782
13.7 Comprehensive Defense Architectures . . . 784
13.8 Differential Privacy . . . 806
13.9 Security and Privacy Maturity Model . . . 816
13.10 Fallacies and Pitfalls . . . 817
13.11 Summary . . . 819

Chapter 14 Robust AI **821**
14.1 The Silent Failure Problem . . . 822
14.2 Real-World Robustness Failures . . . 824
14.3 A Unified Framework for Robust AI . . . 828
14.4 Environmental Shifts . . . 834
14.5 Input-Level Attacks and Model Robustness . . . 843
14.6 Adversarial Defenses . . . 852
14.7 Data Poisoning Defenses . . . 856
14.8 Fallacies and Pitfalls . . . 859
14.9 Summary . . . 860

Chapter 15 Sustainable AI **863**
15.1 The Energy Ceiling . . . 864
15.2 Energy Measurement and Modeling . . . 872
15.3 Carbon Footprint Calculation . . . 885
15.4 Data Center Energy and Resource Consumption . . . 889
15.5 Training vs. Inference Energy Analysis . . . 904
15.6 Hardware Lifecycle and E-Waste . . . 912
15.7 Mitigation Strategies . . . 917
15.8 Policy, Regulation, and the Path Forward . . . 926
15.9 Fallacies and Pitfalls . . . 928
15.10 Summary . . . 929

Chapter 16 Responsible AI **933**
16.1 The Governance Imperative . . . 934
16.2 Core Principles and the ML Lifecycle . . . 936
16.3 Responsible AI Across Deployment Environments . . . 954
16.4 Bias Detection and Fairness Monitoring . . . 959
16.5 Privacy, Unlearning, and Robustness Mitigations . . . 965
16.6 Explainability and Interpretability . . . 972
16.7 Sociotechnical Dynamics . . . 979
16.8 Implementation Challenges and AI Safety . . . 988
16.9 Fallacies and Pitfalls . . . 1001
16.10 Summary . . . 1002

Synthesis

Chapter 17 Conclusion **1007**
17.1 Synthesizing Distributed ML Systems . . . 1008
17.2 Six Principles of Distributed ML Systems . . . 1008
17.3 The Complete Production System . . . 1011
17.4 Competencies Mastered . . . 1013
17.5 The Fleet Stack as Discipline . . . 1013
17.6 Engineering Intelligence at Scale . . . 1015
17.7 Fallacies and Pitfalls . . . 1015
17.8 Summary . . . 1016

References

References **1021**

Appendices **1047**

Chapter A D·A·M Foundations **1047**
How to Use This Appendix . . . 1048
A.1 Diagnostic Summary . . . 1048
A.2 Iron Law Mapping . . . 1048
A.3 Arithmetic Intensity Boundary . . . 1049
A.4 Tooling Map . . . 1050
A.5 D·A·M Scorecard . . . 1050
A.6 Scaling the Taxonomy: From Node to Fleet . . . 1050
A.7 Summary . . . 1051

Chapter B The C^3 Taxonomy **1053**
How to Use This Appendix . . . 1053
B.1 From D·A·M to C^3 . . . 1053
B.2 Diagnostic Summary . . . 1054
B.3 The Fleet Law . . . 1054
B.4 Intersection Landscape . . . 1055
B.5 Rules of Thumb . . . 1056
B.6 C^3 Case Studies . . . 1056
B.7 Production Troubleshooting . . . 1058
B.8 Tooling Map . . . 1058
B.9 C^3 Scorecard . . . 1059
B.10 Scaling Laws Through the C^3 Lens . . . 1059
B.11 Summary . . . 1060

Chapter C Fleet Foundations **1061**
How to Use This Appendix . . . 1061
C.1 The Three System Paradigms . . . 1061
C.2 Numbers Every Fleet Engineer Should Know . . . 1063
C.3 Scaling Physics . . . 1068
C.4 Thermal and Power Physics . . . 1071
Summary . . . 1072

Chapter D Communication **1073**
How to Use This Appendix . . . 1073
D.1 The α-β Communication Model . . . 1073
D.2 Collective Operation Complexity . . . 1075
D.3 Algorithm Selection . . . 1078

D.4 Pipeline Arithmetic . . . 1079
D.5 Compression Economics . . . 1080
Summary . . . 1081

Chapter E Reliability Foundations **1083**
How to Use This Appendix . . . 1083
E.1 Failure Probability at Scale . . . 1083
E.2 Checkpoint Optimization . . . 1086
E.3 Recovery Budgets . . . 1087
E.4 Strategy Selection . . . 1089
Summary . . . 1090

Chapter F Inference Foundations **1091**
How to Use This Appendix . . . 1091
F.1 Serving performance analysis . . . 1091
Summary . . . 1097

Chapter G System Assumptions **1099**
How to Use This Appendix . . . 1099
Foundational Hardware Recap . . . 1100
Cluster Reference Configurations . . . 1100
Network and Interconnect Specifications . . . 1101
Reliability Assumptions . . . 1102
Communication Model Parameters . . . 1103
Sustainability Assumptions . . . 1104
Economic Assumptions . . . 1105
Capacity Planning Assumptions . . . 1106
Unit Conventions . . . 1107
G.1 Assumption Provenance . . . 1108
Summary . . . 1109

Glossary **1111**
A . . . 1111
B . . . 1111
C . . . 1112
D . . . 1112
E . . . 1113
F . . . 1113
G . . . 1114
H . . . 1114
I . . . 1114
K . . . 1114
L . . . 1115
M . . . 1115
N . . . 1116
O . . . 1116
P . . . 1116
Q . . . 1116
R . . . 1116
S . . . 1117
T . . . 1118
V . . . 1118
W . . . 1118
Z . . . 1118

Index **1119**

Welcome

Dedication

For Kari,
who gave me the time this book required
and never asked for it back.

And for Nora and Maya,
the reason any of it matters.

Foreword

Coming soon.

Author's Note

When a new model is announced, the world looks at the model. It almost never looks at the fleet. Behind every frontier model, however, stands tens of thousands of accelerators held in lockstep across a network where one slow link can stall the whole run, cooled by systems that move megawatts of heat, kept alive by protocols that treat failure not as an exception but as a scheduled event. None of this fits in the announcement. The model gets the paper. The fleet gets a footnote, if that.

This book is an argument that the fleet deserves more than a footnote. We have built vast distributed systems before, but none quite like this one, because none of the others had to be right about two things at once. The telephone network had to stay available; it never had to converge. The internet had to deliver the packet; it never had to preserve the gradient inside it. A fleet has to do both. It must move data the way any distributed system does, and it must keep the mathematics of learning intact while it moves. A node that drops out does not only cost capacity; it can break the synchronization a training step depends on, and a single broken step can corrupt days of work. That double obligation, computational and statistical, sustained at the scale of a small campus, is the engineering this book makes visible.

The obligation does not end when training does. A model that cannot be trained is an idea, not a system. A model that cannot be served is a research result, not a product. A model that cannot be governed is a liability, not an asset. At every stage, from the first allocated node to the last served request, the fleet decides what is possible and what remains imagined.

The question under every chapter is easy to ask and hard to answer: what it takes to build not one machine learning system but a thousand, make them work as one, and run them responsibly. I believe that is a defining engineering problem of this generation, and that the people who take it on, the ones who rarely get the headline, deserve a book that treats it as one.

— Vijay Janapa Reddi
Cambridge, Massachusetts
2026

Preface

Who This Book Is For

This book is for anyone who understands machine learning on a single machine and now faces the challenge of making it work at scale. The problems of scale are not the problems of a single node, only bigger. They are qualitatively different: networks partition, hardware fails as a statistical certainty, and the societal impact of every design decision is amplified by millions of users.

The scope ranges from designing the physical substrate of an AI data center, to scaling training beyond the limits of a single accelerator, to reasoning about what happens when a production fleet serves a global user base. Throughout, the focus is on the physics of distribution—the invariants that govern every fleet-scale ML system regardless of the framework, the model, or the era.

Why a Fleet-Level Textbook

In 2012, training AlexNet took five to six days on two GPUs (Krizhevsky et al. 2012). By 2023, public GPT-4-class estimates placed frontier training at roughly 25,000 GPUs running for about three months (SemiAnalysis 2023); OpenAI did not disclose the actual hardware configuration (OpenAI et al. 2023). This is not the same engineering problem at a larger scale. It is a categorically different engineering problem.

On a single machine, performance is governed by the memory wall—the gap between processor speed and memory bandwidth. In a distributed cluster, a new wall emerges: the bisection bandwidth wall, where the minimum network bandwidth that bisects the cluster caps total system throughput. Data no longer moves through local hierarchies alone; it traverses optical fabrics governed by the speed of light between racks and across continents. Hardware failures, rare enough on a single machine to be treated as exceptions, become routine statistical events at fleet scale. A training job that does not plan for failure *will* fail.

These are not operational annoyances. They are physical constraints—as fundamental and as permanent as the memory wall, Amdahl's Law, or the energy cost of data movement. They require their own engineering discipline, with its own invariants:

- **The Bisection Bandwidth Theorem**: The performance of a distributed system is limited by the minimum bandwidth that bisects the network topology.
- **The Fleet Law**: Every training step pays a communication tax that grows with cluster size while per-node computation shrinks.
- **The Young-Daly Checkpoint Law**: The optimal checkpoint interval balances the cost of writing state against the expected cost of lost work due to failure.
- **The Serving Cost Dominance Law**: For high-traffic production models, cumulative inference operational expenditure can exceed the one-time training cost, making serving efficiency a lifecycle optimization target.
- **The Fairness Impossibility Law**: No system can simultaneously satisfy calibration, equalized odds, and demographic parity when base rates differ between groups.

If single-machine ML systems engineering asks *"what can one machine build?"*, then fleet-scale ML systems engineering asks *"what can a thousand machines build together, and at what cost to reliability, efficiency, and society?"*

That final clause is what sets fleet-scale engineering apart. A single machine affects its user; a fleet affects everyone its outputs touch, across millions of decisions per second. At that scale, reliability, efficiency, and fairness stop being features and become engineering constraints as binding as bandwidth or power. A system that scales but cannot be trusted has not been engineered. It has only been enlarged.

This book provides that foundation. It follows the Hennessy and Patterson pedagogical model, extending the quantitative, principles-first approach to the distributed scale (Hennessy and Patterson 2011; Hennessy and Patterson 2019; Patterson and Hennessy 2017). Just as their work taught a generation to reason about processor performance from first principles, this book teaches the reader to reason about fleet performance—replacing intuition about distributed systems with measurement, and ad hoc scaling with engineering.

Introduction to Machine Learning Systems taught the bricks: the tensors, kernels, memory hierarchies, and optimization loops that compose any single-node system. A fleet is not built from more bricks. It is an orchestra. A single musician can produce music of remarkable beauty, but an orchestra is not simply many musicians playing at once. It requires a conductor, a score, precise synchronization, and the ability to recover gracefully when a player drops out. The acoustics of the concert hall shape the sound in ways that no individual instrument can control. The principles that govern orchestral performance—coordination, communication latency, fault tolerance, and the physics of the performance space—are categorically different from those that govern a solo performer. ML fleets work the same way. A training cluster, a serving fleet, and an edge deployment are vastly different systems, but beneath them sit the same building blocks: parallelism strategies, collective communication primitives, checkpoint protocols, scheduling algorithms, and the trade-offs between throughput, latency, fault tolerance, and cost.

What This Book Covers

This book focuses on the **Machine Learning Fleet**: the warehouse-scale computer where the network is the bus, power density is the speed limit, and failure is not an exception but a statistical certainty. This is the unit of modern ML computation, where models too large for any single device are trained across thousands of accelerators, served to millions of users, and governed by constraints that span engineering, economics, and society.

The content is organized into four parts:

- **Part I: The Fleet** builds the physical substrate of the AI data center from the ground up: the landscape of distributed ML systems, the silicon and cooling of compute infrastructure, the network fabrics that connect thousands of accelerators, and the scalable data storage that feeds training pipelines.
- **Part II: Distributed ML** establishes the logic of distribution beyond a single machine: the parallelism strategies (data, tensor, and pipeline) for training models too large for single devices, the collective communication primitives that synchronize gradients, the fault tolerance mechanisms that ensure reliability when failure is routine, and the orchestration systems that manage the fleet.
- **Part III: Deployment at Scale** takes the trained model from the cluster to the world: performance engineering for efficiency, inference serving at massive scale, edge intelligence for resource-constrained devices, and the operational lifecycle required to manage production fleets.
- **Part IV: The Responsible Fleet** confronts the forces that determine whether the fleet serves its users or harms them: security and privacy, robust system design, environmental sustainability, and responsible engineering as constraints as fundamental as power density or bisection bandwidth.

These four parts trace a path through the **Fleet Stack**, a layered abstraction that extends the classical ML systems stack to the distributed scale. At the bottom, **Facility & Accelerators** and **Fabric & Storage** form the physical substrate. Above them, **Distributed Execution** and **Reliability & Control** manage parallel work, communication, scheduling, and recovery. Higher still, **Runtime & Serving** and **Operations** take models to users and manage the production lifecycle. At the top, **Assurance** and **Governance & Sustainability** turn trust, risk, policy, and environmental cost into

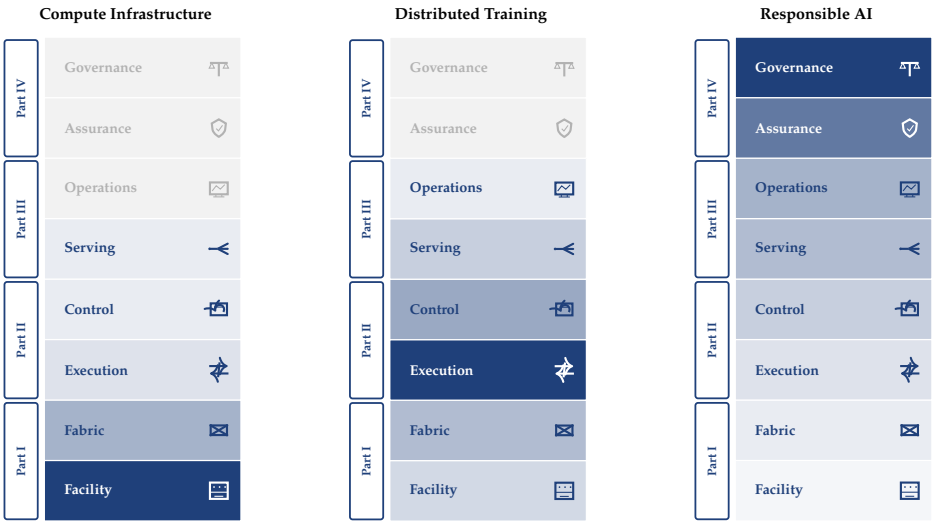
engineering constraints. Each layer provides an abstraction to the layer above and consumes the layer below. Engineering decisions at the bottom constrain possibilities at the top.

Throughout the book, a margin figure highlights which layer each chapter addresses, providing a navigation cue at every chapter opening:



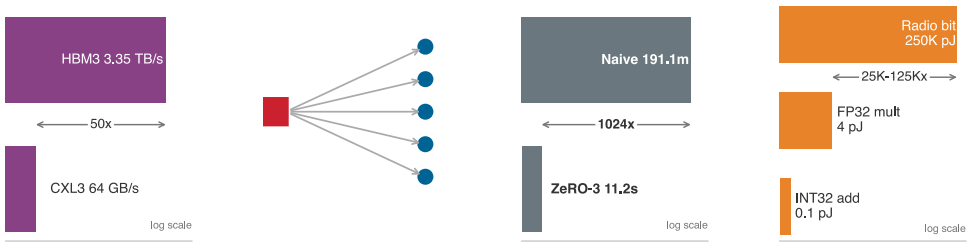
The full stack above shows each row with its scope, grouped into the four Parts of the book. The Roman numerals (I–IV) mark those Parts; the row intensity marks emphasis. The stack uses one Volume II accent color throughout so that shade, not hue, carries the chapter-focus signal.

In the margin figures throughout the book, the intensity of each layer reflects how strongly the chapter engages that tier—brighter (higher intensity) means the chapter's primary focus; dimmer (lower intensity) means contextual relevance as constraints propagate upward. Consider three chapters that illustrate how the emphasis shifts:



In Compute Infrastructure, the bottom row dominates because the chapter is about silicon, power, cooling, and physical deployment. In Distributed Training, **Distributed Execution** is brightest while **Reliability & Control** remains visibly relevant because large training runs depend on scheduling and recovery. In Responsible AI, **Governance & Sustainability** is primary while **Assurance** stays strong because accountability depends on measurable trust and risk controls. The same stack, three different stories: decisions made at one layer ripple through the layers above and below it.

Small visual notes sit in the margins wherever a relationship is fastest to grasp by eye. They are reading aids, not figures to cite later. At fleet scale, this visual grammar makes bandwidth gaps, failure blast radius, checkpoint ratios, and edge-energy trade-offs visible without interrupting the argument. When a miniature uses a log scale, it says so inside the image.



Each part builds on the previous one, so sequential reading is recommended. The reading paths below offer alternatives for readers with specific goals.

Suggested Reading Paths

Readers with different backgrounds and goals can choose among several paths through the material:

The Complete Path involves reading all chapters in sequence, following the causal chain of constraints that governs the ML fleet. The physical silicon and cooling limits dictate the wiring of the data center, which determines how data is supplied. With the hardware built, the algorithms partition the math across it, requiring precise network routing mechanics and the ability to survive inevitable hardware loss. This surviving fleet must be scheduled, tuned, and scaled out to serve millions of users. Finally, deploying at this scale demands rigorous operational frameworks that are inextricably linked to defending the system and ensuring it serves society responsibly. This path develops every concept from first principles of distribution.

The Infrastructure Path focuses on the physical fleet and orchestration. Readers should focus on Part I (Compute Infrastructure, Network Fabrics, Scalable Data Storage) and Fleet Orchestration. This path builds a deep understanding of the physical and operational substrate.

The Scaling Path provides depth in distributed training. Readers should skim Part I for infrastructure vocabulary, then focus on Distributed Training, Collective Communication, Fault Tolerance, and Performance Engineering.

The Production Path covers deployment and operations. After the introduction, readers should move to Inference at Scale, Operations at Scale, Edge Intelligence, and the Responsible Fleet chapters. This path covers everything from serving architecture to governance.

Conventions

This book uses a small set of typographic conventions to mark what each piece of prose is doing.

Bold marks first definitions

A term in **bold** within body prose is its formal first introduction in the volume. Each term receives this treatment exactly once. Subsequent appearances are unbolded; by then the term is part of the working vocabulary the chapter carries forward.

> Every machine learning system has a **dual mandate**: it must manage statistical uncertainty and physical constraints simultaneously.

The bold is a contract: register this term, it will recur. A matching index entry in the back of the book points to the defining occurrence, providing a way to return to it.

Bold also appears in a second, structural role: as a functional label at the start of an item inside a callout box or a summary bullet—for example, **Recommendation**:, **Trade-off**:, or **Result**:. In that context, the bold is scaffolding rather than a first definition.

Italics carry specific rhetorical jobs

Italics are not used for general emphasis. Every italic span in this book performs one of a small number of specific jobs:

- **Direct opposition**: *training* vs. *inference*, *logical* vs. *physical*.
- **Impossibility, stated as physical, mathematical, or logical law**: "sub-10 ms systems *cannot* use distant cloud infrastructure—physics forbids it."
- **Word used as word**: "the term *gradient* refers to a vector of partial derivatives."
- **Thesis punchline**: a single italicized sentence that captures the controlling insight of a section, for example *Data is Source Code.*

An italicized word is performing one of these jobs. A word that is not italicized carries no implicit emphasis.

Callouts have visual identities that match their job

Callout boxes are not decoration. Each type signals a specific kind of content, so the visual identity announces what the box contains before the reader engages with it.

Callout	Contains
Definition	Formal definition of a key term, with its significance, distinction from the nearest neighbor concept, and a common pitfall
Example	Historical case or concrete scenario, structured as context, insight, and systems lesson
Perspective	Analytical observation or mathematical nuance that enriches the surrounding argument
Notebook	Worked calculation walked through step by step, with values computed inline from the book's constants; intermediate tables and equations are the math itself
Checkpoint	Self-check questions to verify understanding of the preceding section
War Story	Real production failure, structured as context, failure, consequence, and lesson
Lighthouse	Recurring workloads characterized by a profile (parameters, FLOPs, dominant constraint), used as anchored case studies across multiple chapters
Principle	Canonical book principle or invariant referenced from later chapters
Theorem	Formal, equation-backed result or bound, stated with its conditions and applied in later quantitative analysis
Takeaways	Four to seven most important results from a chapter, in summary form
Chapter Connection	Bridge from the chapter just ended to the chapter beginning, naming the open question the next chapter addresses

When a callout contains a table, figure, or equation, that artifact is part of the callout's pedagogical unit, not decoration. A Lighthouse's profile *is* its characterization. A Notebook's intermediate tables *are* the worked math. Reading these as separated artifacts loses the teaching arc. Standalone reference tables, by contrast, sit between paragraphs as numbered artifacts the reader returns to via `@tbl-` references; the list of tables at the back of each volume collects them. Both forms are valid; the visual marker (callout vs. standalone) tells the reader which one they are looking at.

Companion Resources

Whichever path readers choose, this book is part of a broader learning ecosystem. The complete text, interactive Colab notebooks, lecture slides, exercises, hands-on labs, and supplementary materials for every chapter are available online at **mlsysbook.ai**.

This book is built in the open. Its full text, figures, and build system are public, and contributions from readers worldwide have sharpened every chapter. Visit **mlsysbook.ai/git** to report an issue, suggest an improvement, or contribute directly.

Prerequisites

Readers should bring the following background:

- **Single-machine ML systems**: Understanding of ML workflows, neural network training, model optimization, and deployment on a single machine. Readers should be comfortable reasoning about memory hierarchies, computational graphs, and hardware-software interactions.
- **Programming**: Fluency in Python, including functions, classes, and data manipulation with NumPy. Familiarity with at least one ML framework (PyTorch, TensorFlow, or JAX).
- **Mathematics**: Comfort with linear algebra, basic calculus, and probability at the undergraduate level.

The following background is helpful but not required:

- **Distributed systems**: Familiarity with networking concepts (TCP/IP, bandwidth, latency), parallelism, and basic distributed computing deepens understanding of the fleet infrastructure and training chapters.
- **Cloud infrastructure**: Experience with container orchestration (Kubernetes, Docker) or cluster schedulers (Slurm) provides useful operational context.
- **Production engineering**: Understanding of monitoring, observability, and site reliability practices will enrich the deployment and operations chapters.

Beyond This Book

This book covers the Machine Learning Fleet as it exists today: warehouse-scale training clusters, global serving infrastructure, and the governance challenges of operating at scale. Topics at the research frontier—including fully autonomous fleet management, next-generation interconnects, neuromorphic computing at scale, and formal verification of fleet-level safety properties—extend beyond this scope but represent active areas of investigation.

Using This Book in a Course

This book grew out of CS249r at Harvard University and the TinyML edX professional certificate program. That teaching reinforced an insight that carries to fleet scale: the same physics governs every level of the hierarchy. Bandwidth, energy, and failure set the limits on a microcontroller and on a data center alike. What changes with scale is not the physics but which limit binds, and a new binding limit is a new engineering problem. On one machine, the memory wall dominates. Across a fleet, bisection bandwidth and routine hardware failure take over, and what was once a rare exception becomes the constraint the entire system is designed around. The physics is continuous; the engineering is not.

The book is designed to support a one-semester advanced course covering the full distributed ML systems stack. Instructors may also select individual parts for shorter modules:

- **Parts I–II** (The Fleet and Distributed ML) suit an advanced half-semester on distributed training infrastructure.
- **Parts III–IV** (Deployment at Scale and The Responsible Fleet) suit an applied half-semester on production ML at scale.

For survey courses or executive programs, the four Part introductions and their governing principles alone provide a condensed overview of the quantitative laws that govern fleet-scale ML systems.

Slides, assignments, and instructor materials for the advanced sequence are available at **mlsysbook.ai**.

Copyright and Licensing

© Vijay Janapa Reddi and MIT Press. This work is distributed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0). The source is available at **mlsysbook.ai/git**.

Acknowledgments

Origins

Earlier teaching on TinyML grew out of the realization that the constraints governing milliwatt devices are the same constraints governing data-center accelerators. This book grew out of a different realization, one that came from working on ML Fleet Efficiency as a visiting researcher at Google. Peter Mattson, who had been a collaborator on MLPerf, made that opportunity possible. What I encountered there changed how I think about ML systems: the engineering required to coordinate thousands of accelerators into a single coherent system is a discipline unto itself, and that discipline had no textbook.

That work introduced me to the ML Productivity Goodput (MPG) metric, fleet-level heterogeneity at a scale I had not seen before, and the daily reality of failure as a statistical certainty rather than an exception. The gap between what practitioners knew and what was written down was enormous. The researchers who built these systems published individual results, but the connective tissue between those results, the principles that made them cohere into a discipline, existed only in the heads of the people doing the work. This book is an attempt to document that connective tissue.

Collaborators and Colleagues

I learned most of what I know about fleet-scale systems from the ML Fleet Efficiency team at Google: Arissa Wongpanich, Tayo Oguntebi, Jose Baiocchi Paredes, Yu Emma Wang, Phitchaya Mangpo Phothilimthana, Ritwika Mitra, and Zongwei Zhou. Naveen Kumar led that effort, and the way he thought about fleet-level optimization shaped how I think about it still. Robert Hundt pushed my understanding of performance engineering at scale. Together, Naveen and Robert showed me what fleet-level efficiency looks like from the inside. David Kanter continued the MLPerf collaboration, and our discussions on benchmarking distributed systems sharpened the quantitative methodology that runs through every chapter. Greg Diamos provided invaluable perspective on scaling laws and their origins at Baidu, where some of the earliest work on large-scale distributed training took shape.

The researchers whose papers fill the bibliography of this book deserve particular acknowledgment. Practitioners and researchers who built the systems, measured the failures, and published what they learned discovered the principles taught here, rather than inventing them for this textbook. This book synthesizes their work into a coherent narrative. The debt is to them.

Support

The Harvard Data Science Initiative, Harvard Extension School, and the National Science Foundation supported this work; the Edge AI Foundation and ICTP supported educational outreach and scholarships; and our industry partners provided infrastructure and tooling that enabled practical experimentation.

At MIT Press, Susan Hartman believed in this project from the beginning and guided both volumes from an open-source experiment to published textbooks. Her editorial judgment and support made this book real. I am equally grateful to MIT Press for agreeing to keep this book available as open access. A textbook about a discipline shaped by the open-source community should be accessible to everyone in it.

For the complete and most up-to-date list of all GitHub contributors, please visit the online version at mlsysbook.ai. For those interested in contributing, please consult our GitHub repository.

Every GitHub star is a signal that someone cares about this material, is learning from it, and believes the work matters. The online community of learners, with their questions, their curiosity, and their willingness to engage with an unfinished textbook, kept this project moving when the scope felt overwhelming.

Finally, I thank my wife and my children. I wrote this book in the hours that belonged to them: weekends that disappeared into revisions, evenings that stretched past midnight, mornings when I was present but not quite there. They never complained. They asked how the writing was going, brought me coffee, and gave me the space to finish. Whatever merit this book has, their patience and their support made it possible.

A Note on AI Assistance

Every chapter of this book was researched, written, and rewritten by me. I also used AI tools throughout. What follows is more than the disclosure MIT Press requires of authors who use AI. It is a statement of the standard I held the book to, and a claim about what authorship means when the tools are this capable.

The standard is traceability. Every quantitative claim in these pages originates in a Python computation cell whose source code ships with the book. Every citation has been verified against its primary source. Every cross-reference resolves to a stable anchor. Every number in prose was computed by a purpose-built infrastructure modeling engine developed for this textbook — not typed by hand. A pre-commit test suite — over thirty automated checks spanning unit consistency, inline-reference resolution, cross-reference integrity, notation canonicality, and citation hygiene — enforces these invariants on every commit. AI tools made parts of this rigor mechanically tractable. The enforcement infrastructure certifies it.

Authorship is thinking. Which concepts belong in this book and which do not. How to sequence the chapters so that each idea arrives only after the reader has the tools to understand it. Which worked examples to trace end-to-end, and which numbers to compute so the reader can verify the argument independently. What level to pitch each explanation at — rigorous enough for a graduate engineer, concrete enough for a student encountering systems thinking for the first time. Where students will get stuck, because I have taught this material and watched them get stuck. These are judgment calls that no tool can make, because they require knowing the reader. I used AI tools throughout this process to brainstorm framings, explore alternatives, and pressure-test my reasoning. The choices are mine.

In practice, AI was genuinely useful for writing code — the computation cells, the pre-commit checks, the worked examples that are essentially programs. It helped me survey literature I then read in the original, draft passages I then rewrote substantially, and audit the manuscript for consistency across hundreds of cross-references, thousands of index entries, and tens of thousands of lines of prose. The pattern throughout was the same: the tool proposes, the human ratifies.

This pattern is older than the present moment. In 1683 Joseph Moxon printed the first English-language manual on printing, *Mechanick Exercises on the Whole Art of Printing*, using the press it described. Three centuries later Donald Knuth wrote *The TeXbook* in TeX, the typesetting system he created so his own books could exist. A book about ML systems infrastructure, written without the infrastructure it describes, would be a contradiction. The toolmaker documents the tool with the tool.

The reliability gap I describe in section 1.4.1 — that fleet-scale ML systems cannot be made failure-free, only engineered to recover — applies to AI-assisted authoring with equal force. The standard I held this book to is the standard I want you to demand of every system you ship: the tool can propose, but only an engineer can ratify. MIT Press does not list AI tools as authors because they cannot assume ethical and legal responsibility for their work. The same is true of every engineered system. Authorship is the assumption of that responsibility, and it belongs to the human.

Notation

Machine Learning Systems spans **Machine Learning** (computer science/statistics) and **Systems** (computer architecture/hardware). Each field developed its notation independently, and many symbols mean different things depending on the community, paper, or literature using them. This collision creates real confusion when the disciplines merge. The conventions below establish a single notation for eliminating ambiguity.

Consider a simple statement: *"Increasing B improves throughput."* To an ML researcher, B means batch size. To a hardware engineer, B means bandwidth. Both interpretations are correct in their respective fields, but in ML Systems we need both concepts in the same equation—hence the need for a single, consistent convention.

The Iron Law of ML Systems

THE fundamental performance equation of this book, introduced in the opening chapter, is:

$$T = \frac{D_{\text{vol}}}{\text{BW}} + \frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}} + L_{\text{lat}}$$

Each variable was chosen deliberately to avoid collision with standard ML terminology.

Symbol	Definition	Unit	Why This Symbol?
T	**Time**	seconds	Unambiguous. Wall-clock time for an operation.
D_{vol}	**Data Volume**	bytes	**Avoids collision with D (Dataset Size)**. In scaling laws, D means training tokens. Here we need bytes moved through memory. The subscript disambiguates.
BW	**Bandwidth**	bytes/s	**Avoids collision with B (Batch Size)**. Physics uses B for bandwidth, but every ML paper uses B for batch size. We preserve the ML convention.
O	**Operations**	FLOPs	Total floating-point operations. Clean in equations (vs. "Ops").
R_{peak}	**Peak Rate**	FLOP/s	**Avoids collision with P (Parameters)**. Roofline models use P for peak performance, but ML universally uses P for parameter count. We preserve the ML convention.
η_{hw}	**Efficiency**	—	Hardware utilization ($0 \leq \eta_{\text{hw}} \leq 1$). Avoids collision with learning rate (η).
L_{lat}	**Latency**	seconds	**Avoids collision with $\mathcal{L}$ (Loss)**. Fixed overhead time (kernel launch, network RTT). The subscript distinguishes from the loss function.

Why these choices matter

Without careful notation, sentences become ambiguous:

> *"Reducing D improves performance."*

The sentence has two possible readings:

- Reducing **dataset size** (fewer training samples) can speed training but may reduce accuracy.
- Reducing **data volume moved** (for example, through compression or quantization) can speed inference, with accuracy effects that depend on the technique and calibration.

With our notation, we can write precisely:

> *"Reducing D_{vol} through FP32-to-INT8 quantization cuts parameter memory traffic to one quarter while D (training data) remains unchanged."*

Our notation makes such ambiguity explicit: *"BW limits throughput"* is unambiguous.

Subscripted variants

When a multi-letter quantity name takes a descriptive subscript (for example, distinguishing storage bandwidth from network bandwidth), wrap both the root and the subscript in `\text{}`:

- $\mathrm{BW}_{\text{disk}}$, $\mathrm{BW}_{\text{network}}$, $\mathrm{BW}_{\text{accelerator}}$
- D_{vol}, R_{peak}, L_{lat}, η_{hw}
- E_{move}, E_{compute}, E_{total}

Without `\text{}`, math mode renders each letter as an italic variable and the spacing collapses to look like a product, not a label.

The Degradation Equation

The silent failure mode of ML systems is captured by the degradation equation introduced in the opening chapter. Some symbols below (such as τ) appear in chapter prose rather than in the block equation itself; defining them centrally keeps the canonical form unambiguous when they do appear.

$$\text{Accuracy}(t) \approx \text{Accuracy}_0 - \lambda \cdot \mathcal{D}(P_t \| P_0)$$

Symbol	Definition	Unit/Type	Notes
$\text{Accuracy}(t)$	**Accuracy at Time t**	Scalar	Model accuracy after the model has been deployed for time t.
Accuracy_0	**Initial Accuracy**	Scalar	Model accuracy at deployment time.
λ	**Sensitivity**	Scalar	Model sensitivity to distribution shift. Architecture-dependent. *(Not wavelength.)*
P_t	**Current Distribution**	Distribution	The data distribution at time t. *(Not parameters—use P for parameter count.)*
P_0	**Training Distribution**	Distribution	The data distribution at training time.
$\mathcal{D}(P_t \| P_0)$	**Statistical Divergence**	Scalar ≥ 0	Measures how far P_t has drifted from P_0. Common choices: KL divergence, total variation, Wasserstein. *(Calligraphic to avoid collision with D = dataset size.)*
τ	**Drift Threshold**	Scalar > 0	Retraining is triggered when $\mathcal{D}(P_t \| P_0) > \tau$.

The Energy Corollary

The energy cost of ML workloads follows from the iron law of ML systems and decomposes as:

$$E_{\text{total}} \approx D_{\text{vol}} \times E_{\text{move}} + O \times E_{\text{compute}}$$

Symbol	Definition	Unit	Notes
$E_{\textbf{total}}$	**Total Energy**	joules	Total energy consumed by an ML workload, decomposed into data-movement and compute terms.
$E_{\textbf{move}}$	**Energy per Byte Moved**	joules/byte	Energy cost of data movement. Dominates total energy ($E_{\text{move}} \gg E_{\text{compute}}$).
$E_{\textbf{compute}}$	**Energy per Operation**	joules/FLOP	Energy cost of a single arithmetic operation.

Deep Learning Notation

We follow standard deep learning conventions (Goodfellow et al. 2016) with explicit disambiguation for systems variables.

Symbol	Definition	Dimensions/Type
B	**Batch Size**	Integer. The number of samples processed in parallel. *(Never bandwidth.)*
P	**Parameters**	Integer. The total count of trainable weights in a model. *(Never peak FLOP/s.)*
D	**Dataset Size**	Integer. Number of training samples or tokens. *(Never data volume in bytes—use D_{vol}.)*
S	**Sequence Length**	Integer. Number of tokens or time steps.
d	**Hidden Dimension**	Integer. Size of the hidden state vector.
d_{head}	**Attention Head Dimension**	Integer. Per-head hidden dimension in attention layers.
N_L	**Number of Layers**	Integer. Total number of layers in a network.
N_{heads}	**Number of Attention Heads**	Integer. Number of attention heads in a multi-head attention layer.
H_{KV}	**Number of Key-Value Heads**	Integer. Number of key-value heads in grouped-query or multi-query attention.
ℓ	**Layer Index**	Integer. Index for a layer. Use instead of bare L when indexing layers, since L collides with loss and latency.
$\mathcal{L}$	**Loss Function**	Scalar. The objective function minimized during training.
η	**Learning Rate**	Scalar. Step size for the optimizer. *(Never bare for hardware efficiency—use η_{hw}.)*
θ	**Model Weights**	Vector/Matrix. The set of all learnable parameters.
$p(x)$	**Distribution**	Probability mass/density for a random variable. Use lowercase $p(\cdot)$ for generic distributions to avoid collision with P (parameter count).
$p(y \mid x)$	**Conditional Distribution**	Conditional probability/density. Use this form for generic label relationships; reserve P_0 and P_t for the degradation equation's training/current distributions.
$\Pr(E)$	**Event Probability**	Probability of an event E. Use for event statements such as $\Pr(\text{batch} = 0)$; use $p(x)$ and $p(y \mid x)$ for distributions.

Local matrix algebra may use A, B, and C for operands when the equation explicitly introduces matrices (for example, GEMM $C = \alpha AB + \beta C$). This is a local linear-algebra convention, not a global override of B as batch size.

Performance, Serving, and Memory Notation

Reusable performance and serving quantities follow the same collision-avoidance rule as the iron law. Rates use R or descriptive Greek symbols; request counts use Q_{req} rather than overloading N; queue utilization uses a subscripted ρ so bare ρ remains available for other ratio models.

Symbol	Definition	Unit/Type	Notes
I	**Arithmetic Intensity**	FLOP/byte	Workload FLOPs per byte moved. The roofline model uses I as the independent variable.
I_{ridge}	**Roofline Ridge Point**	FLOP/byte	$I_{\text{ridge}} = R_{\text{peak}}/\text{BW}$. Prefer this explicit form over starred shorthand so the meaning remains clear in prose.
R_{attain}	**Attainable Compute Rate**	FLOP/s	Roofline rate: $R_{\text{attain}} = \min(R_{\text{peak}}, I \times \text{BW})$. Uses R, not T, because this quantity is a rate.
MFU	**Model FLOPs Utilization**	Dimensionless	Useful model FLOP/s divided by available peak FLOP/s. Text acronym avoids overloading η.
r_{comp}	**Compression Ratio**	Dimensionless	Uncompressed size divided by compressed size. A compressed payload has size M/r_{comp}; the subscript avoids bare C collisions.

Symbol	Definition	Unit/Type	Notes
Q_{req}	**Request Concurrency**	requests	Average in-flight requests in a stable serving system. Avoids collision with N as device count in distributed settings.
λ_{arr}	**Arrival Rate**	requests/s	Request arrival rate. The subscript avoids collision with λ as degradation sensitivity or failure rate.
T_{lat}	**Request Time in System**	seconds	End-to-end queueing/serving latency for Little's Law. Distinct from L_{lat}, the fixed-latency term in the iron law.
$T_{\text{svc}}(B)$	**Batch Service Time**	seconds	Time to serve a batch of size B.
$\mu_{\text{eff}}(B)$	**Effective Service Rate**	requests/s	Batched service rate, typically $\mu_{\text{eff}}(B) = B/T_{\text{svc}}(B)$.
ρ_{serv}	**Serving Utilization**	Dimensionless	Queue/server utilization. Use instead of bare ρ, which is reserved for communication-computation ratio in distributed contexts.
M_{total}	**Total Memory Footprint**	bytes	Sum of explicit memory components; avoids bare M ambiguity.
M_{weights}	**Weight Memory**	bytes	Memory occupied by model parameters.
$M_{\text{gradients}}$	**Gradient Memory**	bytes	Memory occupied by stored gradients.
$M_{\text{optimizer}}$	**Optimizer-State Memory**	bytes	Momentum, variance, master weights, and related optimizer buffers.
$M_{\text{activations}}$	**Activation Memory**	bytes	Retained activations for the backward pass or serving intermediates.
s_{elem}	**Element Storage Size**	bytes/element	Bytes per stored tensor element.

Units and Precision

- Physical units: This book uses SI (metric) units throughout, including meters, kilograms, seconds, watts, and °C, consistent with standard engineering and scientific practice. Where source data was originally reported in imperial units, we convert to SI and note the original values parenthetically. A space always separates the number from the unit (for example, 100 ms, 2 TB/s).
- Data and memory: We use **decimal SI prefixes only**: KB = 10^3 bytes, MB = 10^6, GB = 10^9, TB = 10^{12}. We do not use binary-prefixed units in prose; all capacities, throughputs, and model sizes are reported in decimal units (for example, 80 GB, 2 TB/s, 102 MB).
- Compute: We distinguish operation counts from rates. Total work uses FLOPs (for example, GFLOPs, TFLOPs), while throughput uses FLOP/s with decimal prefixes (for example, GFLOP/s, TFLOP/s).
  - 1 TFLOP = 10^{12} FLOPs
  - 1 TFLOP/s = 10^{12} FLOPs per second
  - Arithmetic intensity conventionally uses FLOP/byte as a unit ratio (floating-point operations per byte moved). This is a ratio unit, not a total-work symbol or throughput symbol.
- Currency: Dollar amounts use the dollar sign (\$); unless otherwise noted, dollar-denominated costs are U.S. dollars (USD).
- Precision:
  - **FP64**: Double precision (8 bytes)
  - **FP32**: Single precision (4 bytes)
  - **TF32**: TensorFloat-32 (19-bit compute format, commonly stored as FP32)
  - **FP16**: Half precision (2 bytes, standard range)
  - **BF16**: Brain float (2 bytes, wide dynamic range)
  - **FP8**: Quarter precision (1 byte, E4M3 or E5M2 format)
  - **FP4**: 4-bit floating-point format
  - **INT8**: 8-bit integer (1 byte)

- **INT4**: 4-bit integer; lower integer precisions follow the same uppercase `INTn` pattern (for example, INT3, INT2)

Quick Reference: Resolving Collisions

Common collision points in ML Systems literature include:

Symbol	ML Meaning	Systems Meaning	Our Convention
B	Batch Size	Bandwidth	**Batch Size**. Use BW for bandwidth.
P	Parameters	Peak FLOP/s	**Parameters**. Use R_{peak} for peak rate.
D	Dataset Size	Data Volume	**Dataset Size**. Use D_{vol} for bytes moved.
L	Loss	Latency	**Loss** ($\mathcal{L}$). Use L_{lat} for latency.
η	Learning Rate	Efficiency	**Learning Rate**. Use η_{hw} for efficiency.

The general principle: **ML conventions take precedence for single letters**; systems concepts get subscripts or multi-letter symbols. This reflects the primary audience (ML practitioners learning systems) and preserves compatibility with the vast ML literature.

Distributed Systems Notation

Distributed-systems work introduces a second layer of notation for coordination, communication, and reliability. Some symbols below are reserved for synchronization, repair, and gradient-sizing contexts; defining them centrally keeps the canonical form unambiguous when they appear. The central equation is the **Fleet Law** (the distributed step time law):

$$T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}$$

Symbol	Definition	Unit	Notes
N	**Number of Devices**	Integer	Accelerator count in a distributed job. *(Not parameters—use P.)*
$T_{\text{step}}(N)$	**Distributed Step Time**	seconds	Wall-clock time for one training step at scale N.
T_{compute}	**Single-Device Compute Time**	seconds	Forward + backward pass on one device.
$T_{\text{comm}}(N)$	**Communication Time**	seconds	Time for collective operations (AllReduce, AllGather). Grows with N.
T_{overlap}	**Overlapped Time**	seconds	Communication hidden behind computation. Reduces effective overhead.
T_{sync}	**Synchronization Time**	seconds	Total nonoverlapped synchronization or coordination cost per step.
ρ	**Communication-Computation Ratio**	Dimensionless	$\rho = T_{\text{comm}}(N)/(T_{\text{compute}}/N)$. Values near or above 1 indicate communication-bound scaling.

The α-β communication model

Network communication time decomposes into a fixed latency and a bandwidth-dependent transfer:

$$T(n) = \alpha + \frac{n}{\beta}$$

Symbol	Definition	Unit	Notes
$T(n)$	**Communication Time**	seconds	Wall-clock time to transmit a message of size n across one link.
α	**Network Latency**	seconds	Fixed per-message overhead. *(Not learning rate—context disambiguates.)*
β	**Link Bandwidth**	bytes/s	Effective throughput per link.
n	**Message Size**	bytes	Size of the payload (for example, gradient tensor).
n^*	**Crossover Point**	bytes	$n^* = \alpha \cdot \beta$. Below n^*: latency-bound. Above: bandwidth-bound.

The LogGP extension

LogGP refines the α-β model by separating network latency, processor overhead, and long-message injection gaps. The standard names are short, but the book uses descriptive subscripts for the highest-collision term.

Symbol	Definition	Unit	Notes
$T_{\text{long}}(n)$	**LogGP Long-Message Time**	seconds	Transfer time for an n-byte long message. Reuses n from the α-β model.
L_{LogGP}	**LogGP Network Latency**	seconds	Standard LogGP latency parameter; subscript avoids bare L collisions.
o_{send}	**Sender Overhead**	seconds	Processor/NIC overhead to initiate a message.
o_{recv}	**Receiver Overhead**	seconds	Processor/NIC overhead to complete a message.
g	**Message Injection Gap**	seconds	Minimum interval between consecutive message injections.
G	**Long-Message Gap per Byte**	seconds/byte	Per-byte long-message gap; distinct from β link bandwidth.

Scaling efficiency

$$\eta_{\text{scaling}} = \frac{T_1}{N \times T_N} \leq 1$$

Symbol	Definition	Unit	Notes
η_{scaling}	**Scaling Efficiency**	Dimensionless	Fraction of ideal linear speedup achieved. 1.0 is the theoretical limit.
T_1	**Single-Device Time**	seconds	Baseline wall-clock time on one device.
T_N	**N-Device Time**	seconds	Wall-clock time on N devices.

Fault tolerance and reliability

System reliability degrades with scale. Assuming independent components with exponential failure distributions, the system reliability across N components is:

$$R_{\text{system}}(t) = e^{-N\lambda t}$$

The aggregate exponent uses the system-level rate $N\lambda$, where λ is the per-component failure rate under the independent, identical-component assumption. The exponent $N\lambda t$ is dimensionless, so t must use the same time unit as λ (hours, when λ comes from FIT-derived rates below).

The optimal checkpoint interval balances I/O cost against rework cost:

$$\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$$

The formula uses *system-wide* MTBF (the failure interval for the whole job), not *component* MTBF; substituting the per-component value would overestimate τ_{opt} by $\sqrt{N}$. For dimensional consistency, T_{write} and $\text{MTBF}_{\text{system}}$ must be in the same time units. Industry-standard MTBF is reported in hours; to combine with seconds-scale write times, convert: $\text{MTBF}_s = \text{MTBF}_h \times 3600$. The result τ_{opt} then carries the same time unit chosen for the inputs.

Symbol	Definition	Unit	Notes
$R_{\text{system}}(t)$	**Reliability Function**	Probability	Probability of no failure across the whole system before time t.
λ	**Per-Component Failure Rate**	failures/hour	Per-component failure rate; the system-wide rate is $N\lambda$ under independent, identical components. The industry-standard reporting unit is FIT (failures per 10^9 device-hours): $\lambda_{\text{hour}} = \text{FIT} \times 10^{-9}$. *(Not sensitivity—context disambiguates.)*
MTBF	**Mean Time Between Failures**	hours	Average time between consecutive failures. $\text{MTBF}_{\text{system}} = \text{MTBF}_{\text{component}}/N$.
MTTF	**Mean Time To Failure**	hours	Component lifetime metric derived from FIT rates. Distinct from system-level MTBF when repair/replacement cycles are modeled.
MTTR	**Mean Time To Repair**	hours	Average recovery time after a failure.
τ_{opt}	**Optimal Checkpoint Interval**	seconds	Young-Daly formula. Minimizes total wasted time.
T_{write}	**Checkpoint Write Time**	seconds	Time to persist model state to storage.

Parallelism dimensions

3D parallelism partitions the training workload across three orthogonal dimensions:

$$N_{\text{total}} = d \times p \times t$$

Symbol	Definition	Unit	Notes
N_{total}	**Total Device Count**	Integer	Product of the three parallelism dimensions. The total accelerators across the job.
d	**Data Parallelism**	Integer	Number of model replicas. *(Also hidden dimension—context disambiguates.)*
p	**Physical Pipeline Stages**	Integer	Number of physical pipeline stages in model-depth partitioning.
t	**Tensor Parallelism**	Integer	Degree of intra-layer partitioning (model width). *(Also time—context disambiguates.)*
m	**Microbatch Count**	Integer	Number of microbatches used to fill and drain the pipeline.
b_{pipe}	**Pipeline Bubble Fraction**	Dimensionless	Fraction of pipeline GPU-time spent idle; avoids bare b.
V	**Virtual Pipeline Stages per Device**	Integer	Number of interleaved virtual chunks assigned to each physical pipeline stage.
M	**Message Payload Size**	bytes	Total size of a gradient, model-state, activation, or compressed message payload to communicate.

Fleet capacity factors

Fleet-scale capacity calculations compound local model efficiency, scaling efficiency, and operational goodput.

Symbol	Definition	Unit	Notes
f_{compute}	**Useful Compute-Time Fraction**	Dimensionless	$(T_{\text{compute}}/N)/T_{\text{step}}(N)$. Measures how much step time is useful local arithmetic after distribution.
f_{overhead}	**Overhead Fraction**	Dimensionless	Fraction of wall time lost to pipeline bubbles, checkpoints, failure recovery, maintenance, or other nonproductive overheads.
η_{goodput}	**Goodput Ratio**	Dimensionless	Fraction of allocated wall time that produces useful training work; often $\eta_{\text{goodput}} = 1 - f_{\text{overhead}}$.
R_{eff}	**Effective Fleet Throughput**	FLOP/s	$R_{\text{eff}} = (N R_{\text{peak}}) \times \text{MFU} \times \eta_{\text{scaling}} \times \eta_{\text{goodput}}$.

Compression economics

Compression models compare codec overhead against the communication time saved by reducing the payload.

Symbol	Definition	Unit	Notes
$T_{\text{transfer}}(x)$	**Transfer Time for Payload** x	seconds	Communication time for an x-byte payload under the relevant link or collective model.
$T_{\text{compressed}}$	**Compressed Communication Time**	seconds	Total time after encode, transfer of M/r_{comp}, and decode.
T_{encode}	**Codec Encode Time**	seconds	Local compression overhead before transfer.
T_{decode}	**Codec Decode Time**	seconds	Local decompression overhead after transfer.

Serving memory at scale

Symbol	Definition	Unit	Notes
M_{KV}	**KV-Cache Memory**	bytes	Key-value cache footprint for autoregressive serving.

Additional collision points

The distributed context introduces further symbol collisions beyond those noted in the base notation:

Symbol	ML Meaning	Distributed Meaning	Our Convention
N	—	Number of Devices	**Device Count** in distributed context (accelerators, not host nodes).
α	—	Network Latency	**Network latency** in α-β model. Context disambiguates.
λ	Sensitivity	Failure Rate	**Context-dependent**. Sensitivity in degradation; failure rate in reliability.
d	Hidden Dimension	Data Parallelism Degree	**Context-dependent**. Parallelism in 3D notation; hidden dim in architectures.

Symbol	ML Meaning	Distributed Meaning	Our Convention
t	—	Tensor Parallelism/Time	**Context-dependent**. Tensor parallelism in 3D notation; time in reliability.
τ	Drift Threshold	Local Interval/Delay	**Avoid bare reuse**. Use τ_{ckpt} for generic checkpoint intervals, τ_{opt} for the Young-Daly optimum, and τ_{stale} for gradient staleness.

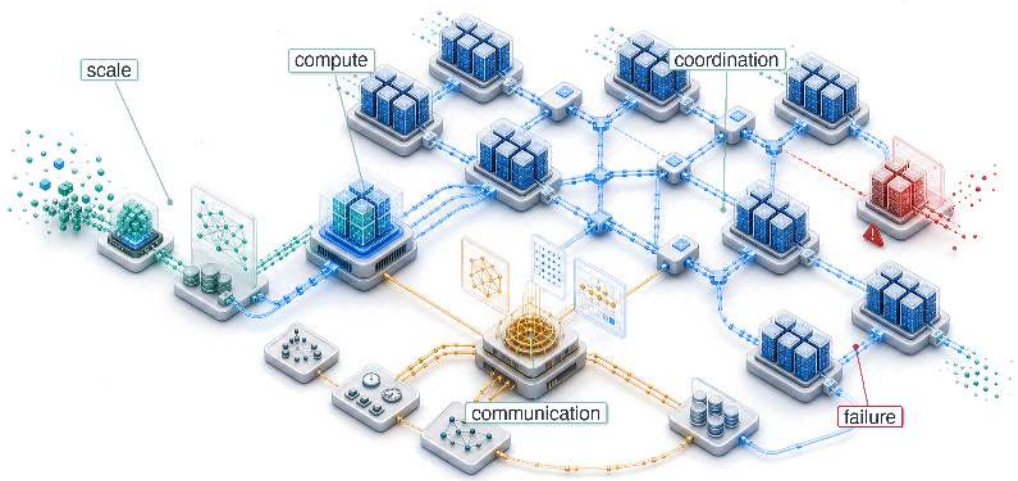
Additional units

- **Network throughput**: Reported in both bytes/s (GB/s, TB/s) and bits/s (Gbps) depending on convention. InfiniBand and Ethernet specifications use Gbps; application-level throughput uses GB/s. We note the convention on first use.
- **Power in sustainability contexts**: Physical power may use explicitly subscripted $P_{\ldots}$ variables (for example, P_{IT}, P_{dynamic}, P_{average}) when the surrounding section is about power or energy. Bare P remains reserved for parameter count.

MAIN MATTER

1

Introduction

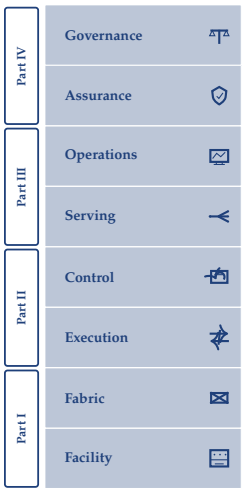


1.1 The Scale Moment
1.2 The Fleet Stack: A Hierarchy of Architecture
1.3 AI Scaling Laws
1.4 Constraints of Scale
1.5 The C^3 Taxonomy: Foundations of Scale
1.6 Foundational Concepts
1.7 The Structure of This Textbook
1.8 Fallacies and Pitfalls
1.9 Summary

Purpose

Why do the engineering principles that work on single machines break down at production scale?

Machine learning at scale has a physics of its own. On one node, performance is governed by the memory wall: data must reach the accelerator fast enough to keep the math busy. In a distributed cluster, that same problem stretches across machines. Data must move through network links between racks, and the slowest shared crossing can limit the entire job. Hardware failures also change character. When a single-accelerator training job fails, it is an inconvenience; when one node in a 10,000-node cluster fails, it can stall the machine learning fleet this chapter defines. This discontinuity explains why mastery of single-machine ML is no longer sufficient: scale is not more of the same, but fundamentally different engineering terrain requiring different principles, different architectures, and different ways of thinking about what makes systems work. That terrain carries a societal dimension that small models do not, because its impact is amplified by the billions of users these systems serve: when a local model exhibits bias, the harm is contained; when a foundation model exhibits bias, it propagates through digital life. In C^3 terms, the discipline this book builds is the physics of distribution, where compute, communication, and coordination turn algorithmic choices into questions of topology, fault tolerance, security, and governance. Each Volume II part introduces the principles for one layer of that progression: fleet foundations, distributed execution, deployment at scale, and responsible operation. Later chapters return to those Volume II principles as the system grows, without collapsing them into the Volume I principle set.



Learning Objectives

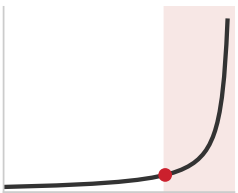
- Explain how Compute, Communication, and Coordination replace single-node Data-Algorithm-Machine constraints in production fleets
- Apply the fleet law to estimate coordination tax and classify scaling regimes
- Analyze scaling-law limits when reliability, network bandwidth, power, or governance constraints dominate growth
- Diagnose fleet-scale distribution risks using reliability gaps, consistency-availability trade-offs, communication intensity, and routine failure rates
- Synthesize fleet-stack design principles across physical infrastructure, operational control, and societal governance

1.1 The Scale Moment

MACHINE learning on a single accelerator is governed by local memory hierarchy and arithmetic intensity: the physics of silicon. The boundary left open by Volume I appears when that local machine is no longer the system. As models move from research prototypes to global services, the binding constraints move outward into racks, networks, power delivery, and recovery machinery. That change is not a smooth extrapolation; it is the **Scale Moment**.

The scale moment is the physical and operational transformation that occurs *when* models cross the three fundamental walls: memory, network, and energy. These three walls are the canonical triad this volume navigates by. On one accelerator the memory wall binds; as we move from a single GPU to a machine learning fleet comprising thousands of nodes, the network wall replaces the local memory wall as the primary performance bottleneck. Its hard limits are bisection bandwidth, the aggregate bandwidth across the narrowest cut that splits the fleet, and speed-of-light latency. Serving billions of users then hits the energy wall, making thermodynamic efficiency a first-order engineering requirement. Crossing those walls opens a reliability gap alongside them, making hardware failure a routine event rather than a rare exception.

Between 2012 and the mid-2020s, public and third-party estimates place training compute growth from roughly 10^{18} FLOPs for AlexNet to approaching 10^{25} FLOPs for leading large models, roughly seven orders of magnitude. The difference is qualitative, not merely quantitative: Compute, Communication, and Coordination now turn algorithmic choices into questions of topology, fault tolerance, security, and governance.



Fleet reliability collapses as node count climbs.

Consider a GPT-4-class training scenario using a hypothetical A100-class cluster of 25,000 GPUs running for 90 days. These values are illustrative rather than disclosed by OpenAI (OpenAI et al. 2023). In a cluster of this size, the probability of at least one failure over an interval t, $\Pr(\text{failure before } t) = 1 - e^{-t/\text{MTBF}_{\text{system}}}$, becomes a binding constraint. The worked example below establishes the arithmetic for one cluster; section E.1.2 formalizes the MTBF cascade that governs failure rates across multi-thousand-GPU fleets, so a reader can predict the interruption cadence for any fleet size.

Napkin Math 1.1: Scale and reliability

Problem: A training run for a GPT-4-class model uses 25,000 GPUs. If each individual GPU has a mean time between failures (MTBF) of 50,000 hours (the canonical data center-grade figure used for fleet-scale reliability examples), how often will the training job be interrupted by hardware failure?

Math:

1. **System MTBF**: The system MTBF is the component MTBF divided by the number of independent GPUs. With this scenario's values, 50,000 hours divided by 25,000 GPUs is about 2 hours.
2. **Daily failure rate**: 24 hours divided by 2 hours is about 12 failures per day.

3. **Total annual failures**: 25,000 GPUs times (8,760 hours divided by 50,000 hours) is about 4,380 failures per year.

Systems insight: In this regime, the system is always in a state of partial failure. Traditional software recovery (manual restart) collapses; the system must be architected for fault tolerance as a first-class citizen. Hardware can no longer be treated as a reliable abstraction; it is a probabilistic resource that requires constant, automated state preservation (checkpointing).

Meta's published Llama 3 training run shows the same reliability arithmetic in a real large-scale system.

War Story 1.1: The training run that failed every few hours (2024)

Context: Meta reports training Llama 3 405B on 16,384 H100 GPUs for a 54-day pretraining run (Dubey et al. 2024).

Failure mode: At that scale, failures were not rare disruptions. The run experienced 419 unexpected interruptions, averaging one interruption every few hours.

Consequence: Training at that scale became an exercise in continuous recovery: checkpointing, automated diagnosis, and restart workflows determined whether the cluster produced useful model progress or burned time waiting for humans.

Systems lesson: The scale moment turns reliability from a hardware specification into a distributed-systems design requirement. A fleet-scale model is trained by the recovery machinery as much as by the optimizer.

The recovery machinery that defined the Llama 3 run exists because cluster sizes exploded; to see why a run now needs hundreds of restart cycles, it helps to trace the compute trajectory that pushed fleets to this scale. The history of machine learning is defined by scale: each major capability leap has emerged from the ability to apply computation at previously impossible scales, making systems engineering central to AI advancement.

Compute requirements have grown exponentially. AlexNet trained on two GTX 580 GPUs for approximately 5–6 days (Krizhevsky et al. 2012). BERT required 64 Tensor Processing Unit (TPU)[1] chips for 4 days, roughly 6,144 chip hours (Devlin et al. 2019).

GPT-3 consumed an estimated 3.14×10^{23} FLOPs during training on V100 GPUs in a Microsoft high-bandwidth cluster (Brown et al. 2020). The surrounding Microsoft infrastructure for this era reached roughly 10,000 GPUs, which is the cluster-size anchor used in figure 1.1. PaLM trained on 6,144 TPU v4 chips for roughly 60 days, consuming approximately 10^{24} FLOPs (Chowdhery et al. 2022). OpenAI's GPT-4 report did not disclose model size, training hardware, or training compute, so the GPT-4-class scenario in this section should be read as illustrative rather than a published configuration (OpenAI et al. 2023). These examples sit within the rapid training-compute growth trend documented by earlier compute-trend studies (Amodei and Hernandez 2018; Sevilla et al. 2022). Table 1.1 makes the scale shift explicit: training budgets move from single-machine experiments into distributed systems where hardware count, wall-clock time, and operational coordination are part of the model design.

Table 1.1: Training Compute Evolution: Growth in hardware scale and training duration across published landmarks and an illustrative GPT-4-class scenario, showing why distributed systems become central as training budgets move from 10^{18}-scale to the largest training runs.

Model	Year	GPUs/TPUs	Training Time	Estimated FLOPs
AlexNet	2012	2 GPUs	5–6 days	~10^{18}
BERT-Large	2018	64 TPUs	4 days	~10^{20}
GPT-3	2020	1,024 V100 GPUs (run est.)	at least 29 days	3.14×10^{23} FLOPs
PaLM	2022	6,144 TPUs	~60 days	~10^{24}

1 **Tensor Processing Unit (TPU)**: Google's custom ASIC, built around a 256×256 systolic array, a grid of multiply-accumulate cells that passes partial sums between neighboring cells, trading GPU flexibility for 15–30× better performance-per-watt on matrix-heavy ML workloads. The fleet-scale consequence is *what* matters here: TPU v4 pods reach 1.1 EFLOP/s aggregate, but their dedicated Inter-Chip Interconnect (ICI), at 4,800 Gb/s per chip, is *what* makes them a single distributed computer rather than a collection of fast chips. Without that interconnect, BERT's 64-chip training would have been communication-bound long before it was compute-bound.

Model	Year	GPUs/TPUs	Training Time	Estimated FLOPs
GPT-4-class scenario	2023	~25,000 GPUs (illustrative)	~90 days (illustrative)	~10^{25} scenario

Training compute is only one dimension. Figure 1.1 traces the related growth in cluster size itself by plotting the number of accelerators used to train landmark models over the past decade.

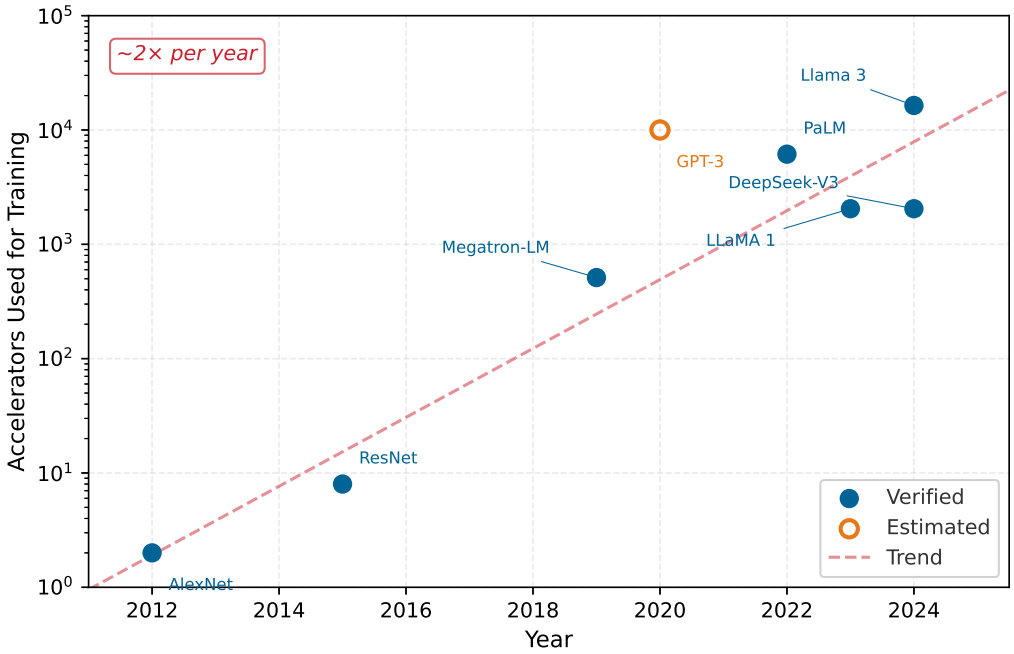


Figure 1.1: The Cluster Size Explosion: Number of accelerators used to train landmark models, 2012–2024. Verified counts from published papers are shown as filled circles; the GPT-3 estimate (hollow marker) reflects approximate cluster size from Microsoft infrastructure announcements rather than a precise published count. The dashed trend line indicates approximately 2× annual growth in cluster size, a rate that outpaces Moore's Law and motivates the infrastructure challenges developed throughout the book.

Figure 1.1 reveals that cluster sizes have grown by roughly four orders of magnitude in just over a decade, from two GPUs for AlexNet to over 16,000 H100s for Llama 3. This empirical trajectory is the foundation of the scale moment: leading models have required far larger accelerator fleets than earlier deep-learning landmarks. The curve alone, however, does not explain *which* constraint breaks first. Large language-model training, recommendation serving, and federated mobile learning all encounter fleet scale, but each stresses a different part of the system.

Lighthouse 1.1: Lighthouse archetypes at scale

Lighthouse Archetypes are the canonical workloads that keep the scale moment concrete throughout this book. At this point, their role is to show that "more machines" is not one engineering problem: the dominant constraint depends on what must cross the fleet boundary. The full roster with the C^3 taxonomy mapping and binding-constraint analysis appears at section 1.6.1.

- **Archetype A (GPT-4/Llama-3)**: Very large language models move from memory bounds on one device to multi-node model parallelism and pipeline parallelism, so communication becomes the bottleneck.

- **Archetype B (DLRM at Scale)**: Deep Learning Recommendation Model (DLRM) workloads move from fitting embedding tables in local memory to **Embedding Sharding** across hundreds of nodes, so sparse all-to-all traffic becomes the bottleneck.
- **Archetype C (Federated MobileNet)**: Mobile inference and on-device adaptation move from one device to federated learning across billions of devices, so unreliable clients, privacy constraints, and stragglers become the bottleneck.

These archetypes turn the growth curve into an engineering question. The same increase in accelerator count can produce a network wall, an embedding-shard bottleneck, or a fleet of unreliable edge participants. Where the archetypes focus attention on the workload, a parallel set of systems lenses does the same for the substrate, directing attention to data-center physics, network topology, and distributed consistency. Combined with the rise of federated learning[2], that diversity has transformed ML from a discipline where algorithms dominate to one where systems engineering determines success. A sophisticated algorithm that cannot scale often provides less practical value than a simpler algorithm deployed efficiently across scalable infrastructure.

The transition from single-machine to distributed training introduces qualitative changes in system behavior. Figure 1.2 contrasts the two regimes: the single-machine world governed by the memory wall vs. the fleet-scale world governed by communication dominance, routine failure, and governance complexity.

2 **Federated Learning**: A distributed learning paradigm where models are trained across millions of decentralized edge devices holding local data. Chapter 11 analyzes the privacy and coordination challenges of federated fleets.

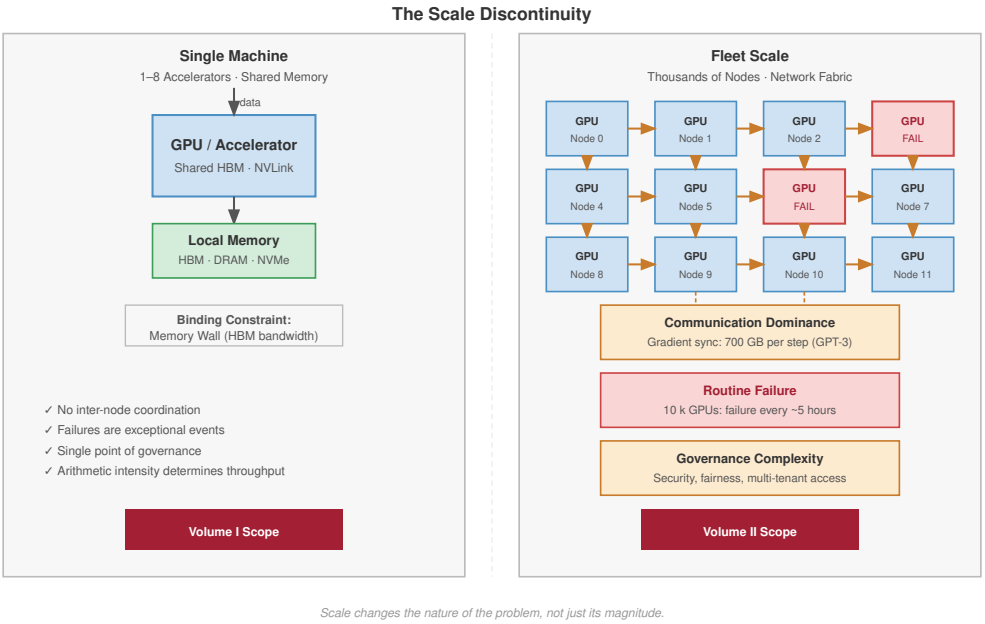


Figure 1.2: The Scale Discontinuity: Single-machine systems (left) are governed by local memory bandwidth and operate with no inter-node coordination, exceptional failures, and a single point of governance. Fleet-scale systems (right) introduce communication dominance, routine hardware failure, and governance complexity as first-order engineering constraints.

The discontinuity captured in figure 1.2 is qualitative, not merely quantitative: at fleet scale, the binding constraint shifts from silicon to the network fabric, and the engineering discipline shifts from optimization to resilience. The unit of compute is no longer a single server but a **Machine Learning Fleet**: a massive, interconnected distributed system that must act as a single coherent engine.

Definition 1.1: Machine learning fleet

Machine Learning Fleet is a distributed system of thousands of interconnected accelerators, storage arrays, and network fabrics designed to operate as a single coherent computer.

1. **Significance**: It coordinates synchronous state across all nodes, where the total time T is governed by the Slowest Worker (Straggler). It requires Bisection Bandwidth ($\text{BW}_{\text{bisect}}$) that scales with the aggregate compute capacity (R_{peak}) of the fleet.
2. **Distinction**: Unlike Traditional Clusters (for example, Spark, MapReduce) that manage independent, asynchronous jobs, an ML Fleet operates under Synchronous Tight Coupling: all workers must reach the training-step barrier before any can advance, so near-perfect reliability is required to maintain throughput.
3. **Common pitfall**: A frequent misconception is that an ML Fleet is "just more servers." In reality, it is a Warehouse-Scale Computer (WSC) where the network is the system bus and the orchestrator is the operating system.

As systems scale beyond a single node, a fundamental physical constraint emerges: the **bisection bandwidth wall**[3], which limits how fast data can cross the network midpoint. At fleet scale, networking often determines model throughput more than compute does.

On a single GPU, training proceeds deterministically: the same code, data, and random seed produce identical results. At the scale of thousands of GPUs, new phenomena emerge. Network partitions can split clusters into groups that train independently, causing model divergence. Stragglers (workers that process data slower than peers due to hardware variation or thermal throttling) can bottleneck entire training runs. Hardware failures that occur once per machine-year become daily events when operating 10,000 machines[4]. Systems must checkpoint frequently enough that losing a day's progress becomes acceptable rather than catastrophic.

These scale-induced challenges have driven infrastructure investment by large AI organizations. Meta's Research SuperCluster (RSC), announced in 2022, contained 16,000 NVIDIA A100 GPUs connected by 200 Gb/s InfiniBand[5] networking (AI 2022). Google's TPU v4 pods contain 4,096 chips with 1.1 EFLOP/s of aggregate compute capacity (Zu et al. 2024). Microsoft's Azure supercomputer for OpenAI reached more than 10,000 GPUs in 2020, and later Azure AI data center announcements describe tens-of-thousands-GPU training fabrics (Langston 2020; Microsoft 2025). The scale of the models dictates the scale of the infrastructure. Section D.1 catalogs these interconnect bandwidth constants and feeds them into the α-β communication model, which lets a reader turn a link's bandwidth and latency into a predicted transfer cost.

The scale moment establishes the *why*: exponential growth in compute demand forces ML systems beyond any single machine, creating communication dominance, routine failure, and governance obligations. The next question is *how* to organize the engineering response. Distributed systems frameworks designed for independent tasks cannot satisfy the tight coupling that ML training demands; a different architectural hierarchy is needed.

1.2 The Fleet Stack: A Hierarchy of Architecture

Apache Spark (Zaharia et al. 2016) processes independent data partitions; a web microservice handles isolated requests. The Machine Learning Fleet does neither. Its workload requires synchronous state updates across thousands of accelerators every few hundred milliseconds, a coupling intensity that existing distributed frameworks were never designed to sustain. The workload characteristics of ML systems differ fundamentally from traditional distributed systems, even though the underlying hardware (network, compute, storage) is identical. To reason about these differences systematically, this book formalizes the engineering crux of scale as a four-layer stack, the fleet stack, which transforms raw cluster resources into global-scale AI applications.

Figure 1.3 visualizes the transition from single-node to fleet. The left side of the diagram summarizes the single-node regime: 1–8 accelerators connected by shared memory, where the binding constraint is the memory wall. The scaling arrow crosses into the distributed fleet regime, where thousands of nodes coordinate across a high-speed switch fabric and the bottleneck shifts to the bisection bandwidth wall: network congestion and message-passing latency dominate.

[3] **Bisection Bandwidth** (from graph theory): The minimum aggregate bandwidth of all links that, if cut, would partition the network into two equal-sized sets of nodes. In distributed ML, this "worst-case" cut determines the cluster's synchronization bottleneck: an AllReduce synchronization can move no faster than the bisection bandwidth, regardless of how many GPUs are added.

[4] **Hardware Failure Rates at Scale**: Individual GPUs fail at 1–2 percent annually under typical conditions, but rates exceed 9 percent under intensive training workloads. Multiply by fleet size and failure becomes routine: Meta reported 419 unexpected interruptions during Llama 3's 54-day training on 16,384 H100s, roughly one every three hours, with GPU and HBM3 faults causing over half. Automated checkpointing and recovery maintained over 90 percent effective training time, illustrating that at fleet scale the engineering challenge shifts from preventing failure to minimizing recovery latency.

[5] **InfiniBand (IB)**: Born in 1999 from the merger of Intel's NGIO and the Compaq/IBM Future I/O initiatives, InfiniBand was originally designed to replace the PCI bus. Its defining feature for ML is RDMA (Remote Direct Memory Access), which bypasses the OS kernel to transfer data directly between application memory on different machines at microsecond-scale latency. HDR IB provides about 25 GB/s line-rate bandwidth per link; NDR reaches about 50 GB/s. This bandwidth gap compared with common Ethernet links determines whether large-model training is compute-bound or communication-bound.

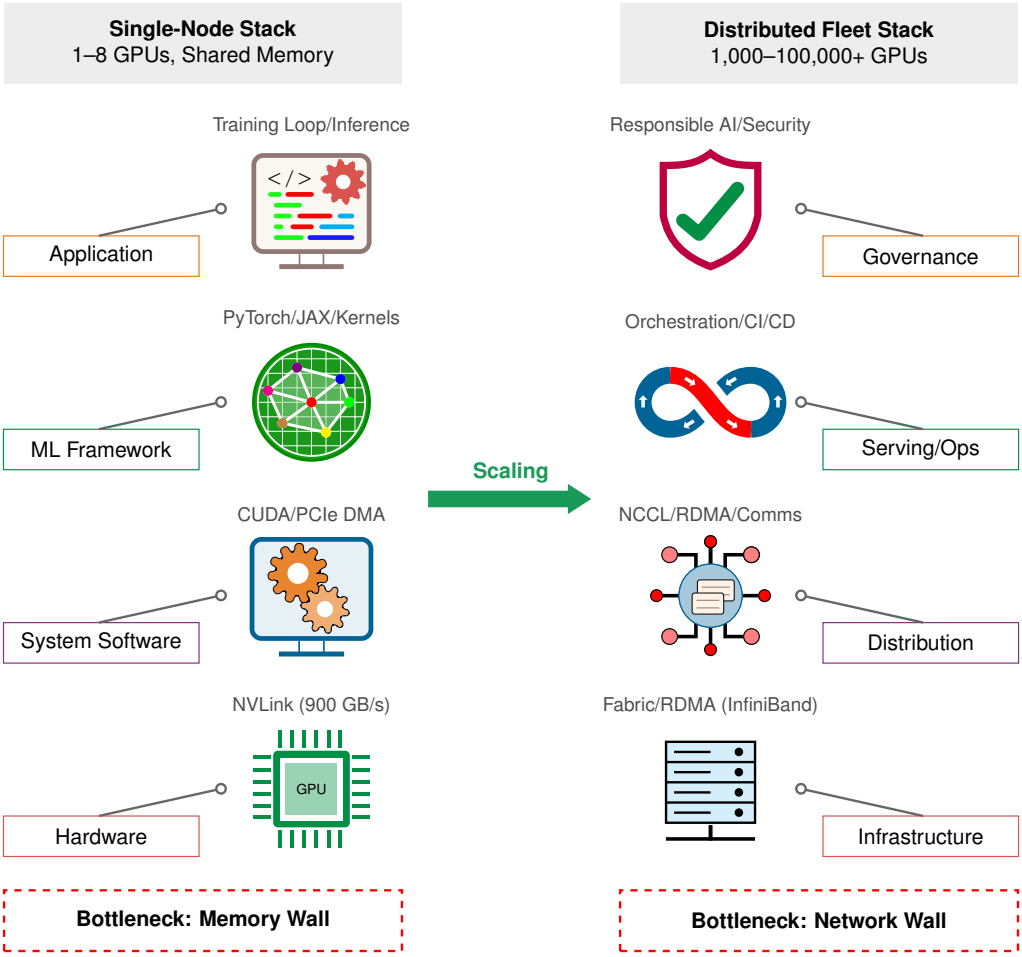


Figure 1.3: The Scaling Regimes of ML Systems: Machine learning engineering is partitioned into two distinct physical regimes. Single-node systems are limited by local memory bandwidth (memory wall), while distributed fleets are limited by network communication (bisection bandwidth wall). Mastery of intra-node data movement is the prerequisite for distributed scaling.

The stack architecture in figure 1.3 does not change when we scale: every ML system still has hardware, a system envelope, a workload, and a mission. *What* changes is the physics at each layer. Read the figure from bottom to top. At the bottom row, Hardware (NVLink at 900 GB/s within one node) becomes Infrastructure (InfiniBand RDMA fabric spanning racks at 400 Gb/s per link), and the bottleneck shifts from the memory wall to the bisection bandwidth wall. One row up, System Software (a single CUDA runtime managing PCIe DMA) becomes Distribution (NCCL and RDMA libraries coordinating thousands of processes across the fabric).

The upper two layers undergo an equally profound transformation. ML Framework (PyTorch or JAX executing a training loop on one node) becomes Serving/Ops (orchestration and continuous integration/continuous deployment (CI/CD) pipelines that schedule distributed jobs and manage rolling deployments). At the top, Application (a single training script or inference service) becomes Governance (responsible AI policy, security auditing, and multi-tenant access control), because fleet-scale deployment introduces organizational concerns absent from a single machine. Four layers compose the fleet stack:

1. **Infrastructure** (Hardware, the Engine): The physical foundation. This layer defines the fleet's raw capabilities: per-node R_{peak} and BW, interconnected by InfiniBand RDMA fabric. The

recurring hardware anchor for this introduction is the NVIDIA H100, which serves as a concrete reference accelerator rather than an abstract placeholder.

2. **Distribution** (Systems, the Car): The communication substrate. This layer defines the cluster envelope: NCCL, the collective library that implements reductions and broadcasts on GPU fabrics; RDMA collectives that coordinate thousands of accelerators; bisection bandwidth; power usage effectiveness (PUE), the facility-energy overhead above IT load; and failure rates (MTBF).
3. **Serving/Ops** (Workloads, the Route): The orchestration layer. This layer manages the mathematical workload sharded across the cluster $(O, D_{\text{vol}}, \text{CI})$ through deployment pipelines and scheduling, where O is local operation count, D_{vol} is data volume, and CI previews the communication-intensity ratio: network bytes per local FLOP. We use Lighthouse Workloads like GPT-4 and DLRM.
4. **Governance** (Missions, the Destination): The mission context. This is the top of the stack, where responsible AI policy, security, and multi-tenant access control shape fleet-wide behavior. A mission (such as Frontier Model Training) introduces high-level requirements (for example, "99.99 percent service availability") that dictate the configuration of every layer below.

This hierarchy ensures that every distributed engineering decision is grounded in its "Mission Context." For example, the Frontier Training mission instantiates Archetype A (GPT-4/Llama-3) from the volume's canonical roster (section 1.6.1), and operates on a cluster of H100 hardware. By standardizing these protagonists, we ensure that the "Physics of Scale" remains traceable across every chapter.

1.2.1 Traditional vs. ML fleet dynamics

Traditional systems (for example, a search engine or a banking database) optimize for independent, asynchronous tasks. A web server handles millions of requests, each isolated from the other. When one request fails, the others continue. This model, exemplified by systems like **MapReduce** (Dean and Ghemawat 2004), achieves scale by partitioning data into independent chunks that require minimal coordination.

The Machine Learning Fleet, by contrast, operates under **Synchronous Tight Coupling**. While the parameter server architecture (Li et al. 2014) introduced ways to manage distributed state, large synchronous models often require even tighter synchronization to maintain performance. Iterative statefulness makes ML training repeat the same math millions of times while updating a massive shared state, the model weights, rather than processing independent one-and-done jobs. Barrier synchronization means that 10,000 GPUs in a synchronous training step must wait for the slowest worker before any can proceed, so a 10 percent performance drop on one node can reduce the entire cluster's throughput by 10 percent. Bisection bandwidth dominance makes ML training bandwidth-bound rather than user-latency-bound, because gigabytes of gradient data must cross the network every second and require non-blocking topologies that traditional data centers rarely implement.

Figure 1.4 illustrates this contrast directly: in MapReduce, workers write independently to shared storage and a straggler delays only its own partition, whereas in the ML Fleet, every worker must arrive at an AllReduce barrier before the training step can proceed.

The barrier synchronization pattern in figure 1.4 explains *why* ML fleets cannot borrow fault-tolerance strategies from MapReduce: in a barrier-coupled system, every worker's progress depends on every other worker's health.

Checkpoint 1.1: The fleet mindset

These questions check whether the fleet mindset is clear:

- ☐ **Straggler impact**: Explain why a single slow worker (straggler) has a disproportionate impact on a synchronous ML training job compared to a MapReduce job.
- ☐ **Bisection bottleneck**: Identify whether traditional web serving or ML training is more likely to make bisection bandwidth the primary performance bottleneck.

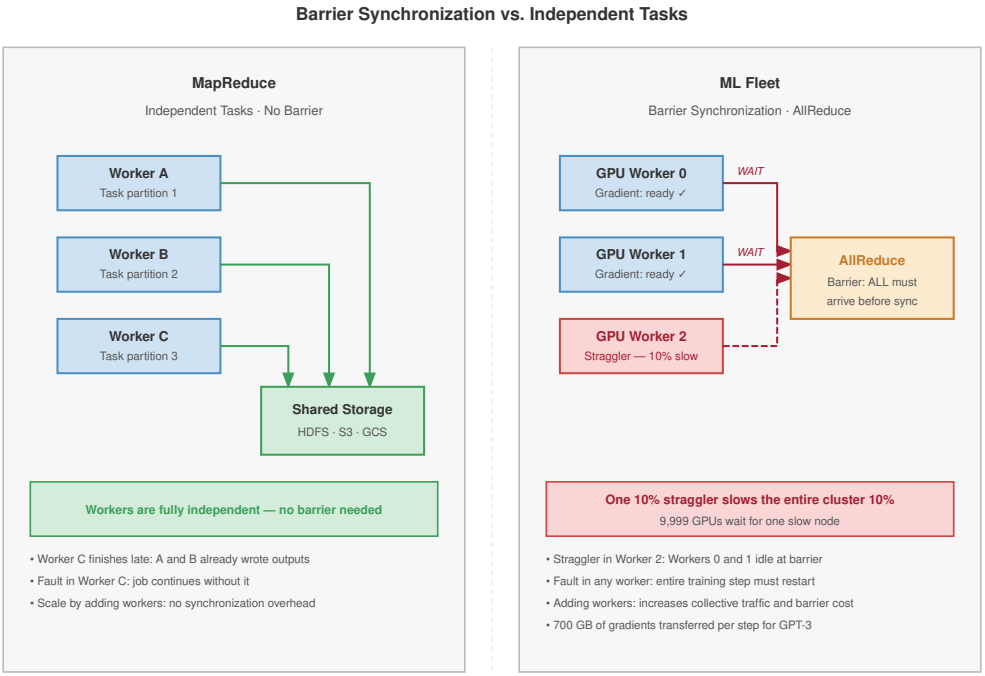


Figure 1.4: Barrier Synchronization vs. Independent Tasks: MapReduce workers (left) operate independently, writing results to shared storage without coordination. A straggler delays only its own output. ML Fleet workers (right) must synchronize gradients at an AllReduce barrier every training step, meaning a single 10% straggler slows the entire cluster by 10%.

- ☐ **Tight coupling**: Explain Synchronous Tight Coupling and connect it to the global barrier at the end of each training step.
- ☐ **Linear speedup myth**: Classify the claim as true or false: adding more GPUs to a training job always results in a linear speedup of the training process.

1.2.2 The shift to the warehouse-scale computer

The ML Fleet demands the **Warehouse-Scale Computer (WSC)**[6] perspective. In traditional computing, the data center is a building that *houses* many computers. In the ML Fleet, the data center *is* the computer.

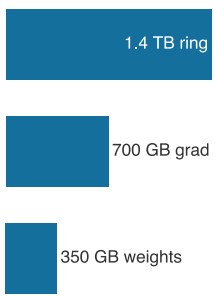
- **Network Fabric**: The system bus.
- **Distributed Storage**: The local disk.
- **Fleet Orchestrator**: The operating system.

Mastering this material requires making this mental shift: the engineer is no longer writing code for a CPU but writing logic for a 100-Megawatt computer spanning thousands of racks. The warehouse-scale computer is a common paradigm for large models, while alternative architectures like **wafer-scale engines** attempt to collapse this entire hierarchy back into a single piece of silicon, trading the modularity of a distributed cluster for the extreme bandwidth of on-chip communication.

1.2.3 Communication becomes dominant

These workload characteristics produce two further consequences at scale, both expressions of the canonical triad introduced at the scale moment: communication becomes the dominant cost (the network wall), and failure becomes routine (the reliability gap that accompanies it). At small scale,

6 **Warehouse-Scale Computer (WSC)**: Barroso and Hölzle, later with Clidaras, framed the data center itself as the computer. For ML fleets, power delivery, cooling topology, optical network layout, and bisection bandwidth become first-order constraints; a 100 MW facility can limit training throughput more than any single accelerator's TFLOP/s.



Gradient synchronization becomes the defining cost of distributed training.

computation dominates. Training a model on a single GPU spends most of its time performing matrix multiplications. Communication overhead is a small fraction of total time.

At large scale, *communication dominates*. Distributed training requires synchronizing gradients across workers after each batch. For a model with 175B parameters, FP32 gradients occupy about 700 GB before any collective algorithm is applied. In Ring All-Reduce, each worker sends and receives roughly $2(N-1)/N$ times the gradient tensor size, because each gradient byte makes about two trips around the ring, one to reduce and one to broadcast back; Chapter 6 derives this factor. Network traffic therefore depends on precision, worker count, and collective implementation, and on slower interconnects communication can consume a large fraction of each iteration rather than a negligible one.

This ratio explains *why* distributed training systems optimize communication so aggressively. **Horovod** uses Ring All-Reduce, NCCL integration, and Tensor Fusion to improve collective communication (Sergeev and Balso 2018); **Megatron-LM** applies model parallelism (Shoeybi et al. 2019); and **ZeRO** reduces memory redundancy (Rajbhandari et al. 2020). For the present argument, the important mapping is role-based: Horovod represents the collective runtime, NCCL the GPU communication backend, Tensor Fusion the gradient-fusion path, Megatron-LM model partitioning, and ZeRO optimizer-state sharding. At fleet scale, these techniques are requirements for viability, not optional performance improvements.

1.2.4 Failure becomes routine

The reliability arithmetic established at the scale moment already showed the second consequence: with thousands of GPUs, hardware fails every few hours, manual intervention becomes impossible, and the system must self-heal through frequent checkpointing, redundant workers, and automated recovery. The formal availability model and its architectural consequences follow in section 1.4.1.

Communication dominance and routine failure are consequences of the scale moment, but they do not explain *what* drives the relentless growth in fleet size. The answer lies in a set of empirical relationships that connect model quality to resource investment, relationships that have made warehouse-scale infrastructure an economic necessity rather than an engineering luxury.

1.3 AI Scaling Laws

Training GPT-3 consumed roughly 3×10^{23} floating-point operations (Brown et al. 2020). Public GPT-4-class training estimates are often on the order of 10^{25} FLOPs, although OpenAI did not disclose the actual training compute or hardware configuration (OpenAI et al. 2023; SemiAnalysis 2023). These estimates matter here because each order-of-magnitude increase in compute demanded a corresponding expansion of the Machine Learning Fleet.

This pattern is not coincidental. Rich Sutton's "bitter lesson" articulated the underlying principle: performance in machine learning is primarily driven by applying general methods at massive scale rather than encoding human knowledge into algorithms (Sutton 2019). Scaling laws formalize this observation quantitatively: model loss improves sublinearly as a power-law function of compute, dataset size, and parameters, $\mathcal{L}(X) \propto X^{-\alpha_{\text{scale}}}$ (Kaplan et al. 2020; Hoffmann et al. 2022). Each equal-sized loss improvement therefore requires disproportionately more resources, a systems consequence this volume later names as the universal scaling law. Each scaling dimension (parameters, data, and compute) interacts with infrastructure constraints differently, making multi-dimensional efficiency optimization essential at production scale.

1.3.1 Empirical evidence for scaling laws

The rapid evolution in AI capabilities since the late 2010s exemplifies this scaling trajectory. GPT-1 (2018) contained 117 million parameters and performed basic sentence completion. GPT-2 (2019) scaled to 1.5 billion parameters and achieved coherent paragraph generation.

GPT-3 (2020) expanded to 175B parameters and achieved sophisticated text generation across diverse domains. Each increase in model size brought substantially improved capabilities at exponentially increasing costs.

The pattern extends beyond language models. In computer vision, parameter counts climbed roughly an order of magnitude from AlexNet (2012) to large vision transformers, and each generation

traded better accuracy for proportionally more compute and training data, the same coupling that forced vision training onto multi-accelerator infrastructure.

The scaling hypothesis underlies this progress: larger models capture more intricate data patterns, yielding improved accuracy and generalization. This trajectory, however, introduces critical resource constraints. Training GPT-3 required approximately 3.14×10^{23} floating-point operations, equivalent to running a consumer gaming PC continuously for hundreds of years, at substantial financial and environmental costs.

These resource demands reveal *why* scaling laws are necessary for efficient resource allocation. Figure 1.5 traces *how* computational demands of training large models have escalated at an unsustainable rate, growing faster than Moore's Law improvements in hardware.

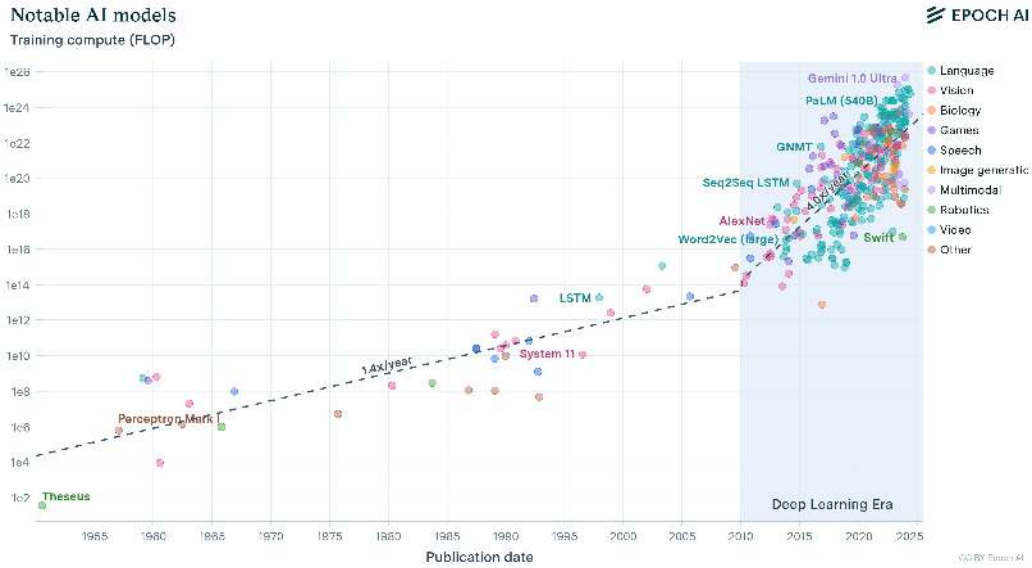


Figure 1.5: Model Training Compute Trends: Training compute has grown exponentially, accelerating dramatically in the deep learning era. The fitted trends shift from roughly 1.4× per year before 2010 to roughly 4× per year inside the deep learning era, both far steeper than Moore's Law (~2 years). The fastest stretch was steeper still: within the 2012–2019 frontier subset, compute doubled approximately every 3.4 months. This trajectory explains *why* efficiency has become a strategic imperative rather than an optional optimization.

These scaling relationships provide a quantitative framework for navigating the trade-offs. Model performance follows power-law relationships where improvements are consistent but exhibit diminishing returns. Optimal resource allocation therefore requires coordinating model size, dataset size, and computational budget rather than scaling any single dimension in isolation. The computational characteristics that drive these workloads' resource demands determine *how* they stress distributed infrastructure.

> **Systems Perspective 1.1: Transformer compute refresher**
>
> Transformers process sequences using self-attention mechanisms that compute relationships between all token pairs. This architecture's computational cost scales quadratically with sequence length ($\mathcal{O}(S^2)$ where S is sequence length), making resource allocation particularly critical for language models. The term "FLOPs" (floating-point operations) quantifies total computational work, while "tokens" represent the individual text units (typically subwords) that models process during training.

1.3.2 Compute-optimal resource allocation

Empirical studies of large language models (LLMs) reveal a key insight. For any fixed computational budget, there exists an optimal balance between model size and dataset size (measured in tokens[7])

[7] **Tokens**: Subword units produced by algorithms like Byte-Pair Encoding (BPE), which iteratively merges the most frequent character pairs in a corpus. Token vocabulary size creates a direct systems trade-off: larger vocabularies can reduce sequence length for some languages and domains, but they expand the embedding table roughly in proportion to vocabulary size. GPT-3 used a 50,257-token vocabulary; moving to 100K+ tokens in newer models therefore trades tokenization efficiency against additional embedding memory.

that minimizes training loss.

Figure 1.6 illustrates this principle through three related views. The left panel shows IsoFLOP curves where each curve corresponds to a constant number of floating-point operations (FLOPs[8]) during transformer training. The valleys in these curves identify the most efficient model size for each computational budget. The center and right panels reveal how the optimal number of parameters and tokens scales predictably as computational budgets increase, confirming that compute-optimal training requires coordinated scaling. The annotated markers extrapolate those fits to a frontier-scale budget near 10^{24} FLOPs, where the compute-optimal point lands at roughly 63 billion parameters trained on roughly 1.4 trillion tokens. That pairing is the roughly 20 tokens per parameter Chinchilla ratio in action: parameters and tokens grow together rather than either dimension racing ahead.

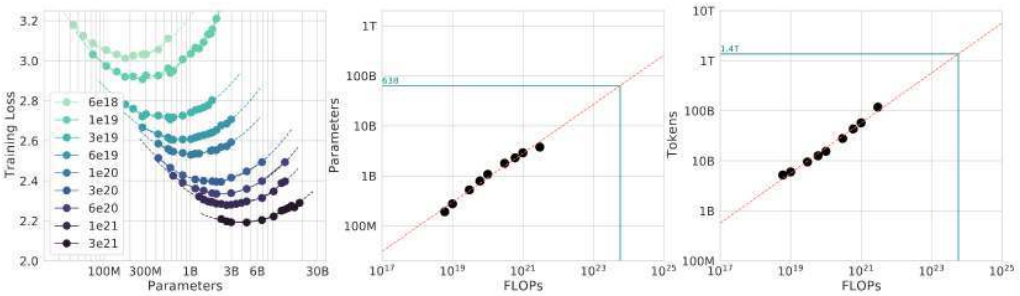


Figure 1.6: Optimal Compute Allocation: For fixed computational budgets, language model performance depends on balancing model size and training data size; the left panel maps training loss across parameter counts, identifying an efficiency sweet spot for each FLOP level. The center and right panels quantify how optimal parameter counts and token requirements scale predictably with increasing compute, demonstrating the need for coordinated scaling of both model and data to maximize resource utilization in large language models.

Kaplan et al. (2020) demonstrated that transformer-based language-model loss follows predictable power-law relationships with model parameters, dataset size (measured in tokens), and total computational budget. The result was not a license to increase every dimension blindly; it showed that model quality depends on coordinated allocation across those resources, a point later sharpened by Chinchilla's compute-optimal scaling analysis (Hoffmann et al. 2022).

Figure 1.7 presents test loss curves for models spanning from 10^3 to 10^9 parameters, revealing two insights. Larger models achieve superior sample efficiency, reaching target performance levels with fewer training tokens. As computational resources increase, the optimal model size grows correspondingly, with loss decreasing predictably when compute is allocated efficiently.

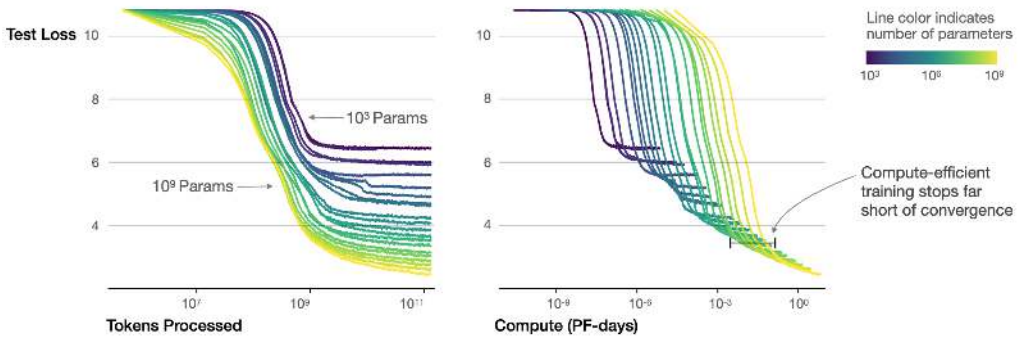


Figure 1.7: Scaling Laws & Compute Optimality: Larger models consistently achieve better performance with increased training data and compute, but diminishing returns necessitate careful resource allocation during training. Optimal model size and training duration depend on the available compute budget, as evidenced by the convergence of loss curves at different parameter scales and training token counts.

Optimal compute allocation follows the Chinchilla result that model size *and* training tokens should scale approximately proportionally for compute-optimal training, with a ratio on the order

[8] **FLOPs vs. FLOP/s**: A critical distinction: FLOPs denotes total computational *work* (operations performed), while FLOP/s denotes hardware *throughput* (operations per second). Confusing the two leads to incorrect cost estimates. GPT-3 required 3.14×10^{23} FLOPs of work; an A100 delivers 312 TFLOP/s of throughput. Dividing work by peak throughput gives ideal wall-clock time; actual time is approximately work$/(\eta_{\text{hw}} R_{\text{peak}})$, so utilization below one multiplies ideal time by $1/\eta_{\text{hw}}$.

of 20 training tokens per parameter (Hoffmann et al. 2022). This corrected earlier guidance that favored making models larger while training them on comparatively fewer tokens.

These predictions assume perfect compute utilization, an assumption that breaks down in distributed training. Communication overhead scales unfavorably with system size, creating bandwidth bottlenecks that reduce effective utilization. At multi-thousand-GPU scale, utilization depends on partitioning and interconnect: Narayanan et al. (2021) report Megatron-LM runs on 3,072 A100 GPUs at 52 percent of peak device throughput and note that slower interconnects or communication-heavy partitions hinder scaling.

1.3.3 Mathematical foundations and operational regimes

Power-law relationships express scaling behavior mathematically, though the intuition behind these patterns matters more for system design than precise formulation. The equations in this section explain why the same fleet can enter different operational regimes governed by how compute and data are allocated, by where diminishing returns set in, and by how much computation a system spends before answering.

Theorem 1.1: Power-law scaling formulation

Scaling laws are expressed formally as power-law relationships. The general formulation is:

$$\mathcal{L}(X) = A_{\mathcal{L}} X^{-\alpha_{\text{scale}}} + C_{\mathcal{L}}$$

where loss $\mathcal{L}$ decreases as resource quantity X increases, following a power-law decay with rate α_{scale}, plus a baseline constant $C_{\mathcal{L}}$. Here, $\mathcal{L}(X)$ represents the loss achieved with a resource quantity such as parameters, tokens, or compute; $A_{\mathcal{L}}$ and $C_{\mathcal{L}}$ are task-dependent constants; and α_{scale} is the scaling exponent that characterizes the rate of performance improvement. A larger value of α_{scale} signifies more efficient performance improvements with respect to scaling.

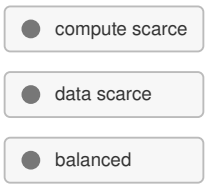
These predictions find strong empirical support across multiple model configurations. Figure 1.8 shows *how* early-stopped test loss varies predictably with both dataset size and model size, confirming that learning curves across configurations align through appropriate parameterization.

1.3.3.1 Resource-constrained scaling regimes

Applying scaling laws in practice is a scarcity diagnosis: the useful question is which resource prevents balanced growth first. Compute budget, data availability, and model size create three regimes with different optimal responses.

Three regimes determine the appropriate scaling response:

- **Compute-limited regime**: Compute scarcity restricts scaling potential despite abundant training data, so academic institutions, startups, and time-constrained teams often train smaller models for longer periods to maximize utilization.
- **Data-limited regime**: Data scarcity appears when computational resources exceed what the available dataset can support, so teams in specialized, proprietary, or privacy-constrained domains often train larger models for fewer optimization steps to extract more information from limited examples.
- **Optimal regime**: Balanced compute and data follow compute-optimal scaling laws, as DeepMind's Chinchilla model demonstrated by outperforming much larger models through proportional scaling of model size and training data (Hoffmann et al. 2022).



Scaling regimes are compute-scarce, data-scarce, or balanced.

Recognizing which regime governs a given project prevents common inefficiencies: over-parameterized models with insufficient training data, or under-parameterized models that waste available compute.

Performance improvements follow predictable patterns, but the relevant design action changes with resource availability and with the point in the ML lifecycle where compute is spent. Two further distinctions matter for system design: how performance changes with dataset size, and when in the lifecycle additional compute is applied.

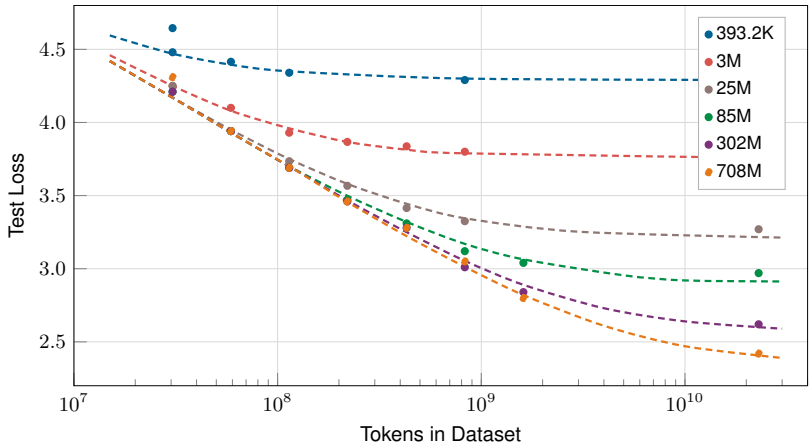


Figure 1.8: Loss vs. Dataset Size Across Model Scales: Test loss curves showing how models of different sizes (393K to 708M parameters) benefit from increased training data. Larger models achieve lower loss but all curves exhibit diminishing returns at high token counts.

Dataset size passes through three phases of its own (figure 1.9). With few examples, generalization error is high and erratic; as data grows it falls predictably along a power law, the regime where additional data buys the most; eventually it saturates against a noise floor where more data yields negligible gain. The systems consequence is that finite high-quality data, not compute, becomes the binding constraint once a model enters the saturation regime, which is why data efficiency complements brute-force scaling. Chapter 5 develops where this saturation reshapes training-data pipelines and budgets.

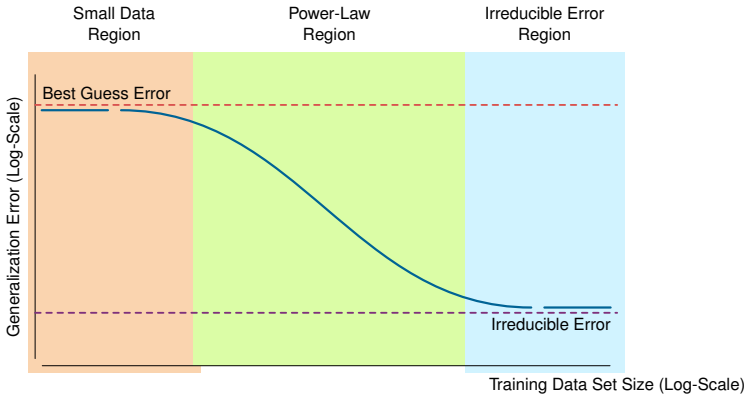


Figure 1.9: Data Scaling Regimes: Generalization error as a function of dataset size exhibits three distinct phases. The Small Data regime shows high variance; the Power-Law regime demonstrates predictable error reduction; the Irreducible Error regime represents the fundamental noise floor.

The lifecycle distinction names three points where a fleet can invest compute, as figure 1.10 traces: pretraining, the long, failure-sensitive run that realizes the scaling law in wall-clock time; posttraining adaptation, where fine-tuning and preference optimization replace one massive run with many smaller, coordination-heavy jobs; and test-time computation, where extra reasoning steps per query trade serving latency and throughput for accuracy. Each shifts the binding constraint to a different part of the fleet, so the full treatments belong with their workloads: Chapter 5 for the pretraining and posttraining runs, and Chapter 10 for test-time compute at serving. For resource-constrained deployments, posttraining and test-time scaling often prove more practical than retraining a model from scratch.

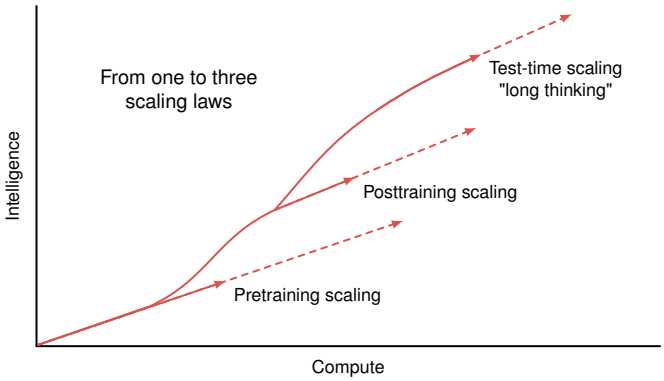


Figure 1.10: The Three Scaling Phases: Intelligence (capability) improves through successive scaling regimes. Pretraining scales with compute and data. Posttraining (reinforcement learning from human feedback, instruction tuning) refines behavior. Test-time scaling (chain-of-thought, search) extracts additional capability at inference.

1.3.4 Practical applications in system design

When OpenAI sized GPT-3, it did not run an exhaustive architecture search; it extrapolated scaling curves from smaller experiments to pick a model size and token budget in advance. That decision is the clearest illustration of how scaling laws inform practical system design and resource planning. Within well-defined operational regimes, model performance depends predominantly on scale rather than idiosyncratic architectural innovations, though diminishing returns mean each additional improvement demands exponentially increased resources for progressively smaller benefits.

Concretely, the authors used scaling-law extrapolations to choose model size and training data under the guidance available at the time (Brown et al. 2020; Kaplan et al. 2020). They scaled an established transformer architecture to 175B parameters and approximately 300 billion tokens, enabling advance prediction of model performance and resource requirements; later Chinchilla results showed that this point was undertrained relative to compute-optimal token allocation (Hoffmann et al. 2022).

Scaling laws serve multiple practical functions. During resource budgeting, empirical scaling curves help estimate returns on investment across model size, dataset expansion, and training duration under fixed computational budgets. Scaling trends also reveal when architectural changes yield significant improvements relative to gains from scaling alone, reducing the need for exhaustive architecture search. When a model family exhibits favorable scaling behavior, scaling the existing architecture is often more effective than transitioning to unvalidated designs.

The same principles apply in reverse for resource-constrained settings. In edge and embedded environments, scaling laws enable designers to select smaller configurations that deliver acceptable accuracy within deployment constraints. By quantifying scale-performance trade-offs, these laws identify when brute-force scaling becomes inefficient and point toward model compression, knowledge transfer, and hardware-aware design.

Scaling laws also function as diagnostic instruments. A performance plateau despite increased resources may indicate dimensional saturation (inadequate data relative to model size) or inefficient compute utilization. This diagnostic capability makes scaling laws both predictive and prescriptive: they forecast what resources a target capability requires and reveal *why* a system underperforms its predicted trajectory.

1.3.5 Sustainability and cost implications

Scaling laws reveal performance pathways while exposing rapidly escalating resource demands. As models expand, training and deployment costs grow disproportionately, creating tension between performance gains and system efficiency. Training the largest models demands distributed infrastructures comprising hundreds or thousands of accelerators, consuming tens of thousands of GPU-days and millions of kilowatt-hours of electricity. Chapter 5 examines how distributed training introduces additional complexity around communication overhead, synchronization, and scaling efficiency.

Large models also require extensive, high-quality datasets to reach their potential. As models approach saturation of available high-quality data, particularly in natural language processing, performance gains from data scaling can become marginal. Data efficiency is therefore a necessary complement to brute-force scaling.

The sharpest of these costs is environmental, and it turns on a decision the fleet operator actually makes: where to site the cluster. The same training run can differ by roughly 40× in CO_2 emissions depending on grid location, from Quebec hydropower at ~20 g CO_2/kWh to Poland coal at ~800 g CO_2/kWh.[9] That single placement choice, together with job scheduling that shifts workloads to low-carbon hours, makes data-center siting a first-order infrastructure decision rather than an afterthought. Financial cost moves in parallel: training a large foundation model runs into the millions of dollars, which limits who can do large-scale research at all. Chapter 15 develops the full energy and carbon accounting that turns these pressures into design constraints.

Scaling laws do not guarantee unbounded improvement. Each incremental performance gain must be weighed against its resource cost. As systems approach practical scaling limits, the emphasis shifts from *scaling alone* to *efficient scaling*: balancing performance, cost, energy consumption, and environmental impact.[10]

1.3.6 Scaling law breakdown conditions

Scaling laws hold within specific operational regimes but break down at their boundaries. As systems expand, they encounter conditions where the assumptions of smooth, predictable scaling cease to apply. These breakdown points expose inefficiencies that demand refined design approaches.

Most of these breakdowns share one root cause: growth in one dimension that outpaces the others. Hoffmann et al. (2022) show that many large language models were undertrained because model size grew faster than training-token budgets, and that compute-optimal training requires scaling parameters and tokens together. The same imbalance appears in other guises. Schedules and learning rates that are not retuned to a larger model leave it short of its potential despite the infrastructure spent on it; finite high-quality data eventually yields diminishing marginal utility, so larger models begin to memorize rather than generalize; and hardware ceilings on memory bandwidth, interconnect capacity, and I/O throughput cap how far a trillion-parameter model can be distributed at all. At the extreme, even balanced scaling reaches the limits of what a training distribution can teach: benchmark numbers keep rising while generalization does not, and models grow brittle to adversarial or out-of-distribution inputs. Table 1.2 organizes these failure modes, mapping each breakdown type to its underlying cause and a representative scenario, so practitioners can anticipate the inefficiency before committing the budget.

[9] **Carbon Intensity of Training**: GPT-3 training emitted an estimated 552 tons of CO_2, and published estimates place large-language-model training in the 10^2 to 10^3 ton range, though figures vary widely with hardware, utilization, and electricity mix (Patterson et al. 2021).

[10] **Scaling Law Saturation**: While power-law relationships suggest unbounded gain, empirical evidence identifies **Semantic Saturation** points where adding data or parameters yields negligible improvement in downstream utility. This "diminishing returns" regime forces systems engineers to pivot from brute-force scale to data efficiency and model compression to extract further value within fixed power budgets.

Table 1.2: Scaling Breakdown Types: Unbalanced scaling across model size, training data size, and compute resources leads to specific failure modes, such as overfitting or diminishing returns, impacting system performance and efficiency. The table categorizes these breakdowns, identifies their root causes, and provides representative scenarios to guide more effective system design and resource allocation.

Dimension Scaled	Type of Breakdown	Underlying Cause	Example Scenario
Model Size	Overfitting	Model capacity exceeds available data	Billion-parameter model on limited dataset
Training Data Size	Diminishing Returns	Saturation of new or diverse information	Scaling web text beyond useful threshold
Compute Budget	Underutilized Resources	Insufficient training steps or inefficient use	Large model with truncated training duration
Imbalanced Scaling	Inefficiency	Uncoordinated increase in model/data/compute	Doubling model size without more data or time
All Dimensions	Semantic Saturation	Exhaustion of learnable patterns in the domain	No further gains despite scaling all inputs

Checkpoint 1.2: Applying scaling laws

These questions check whether scaling laws are being used as resource-allocation tools:

- ☐ **Compute vs. data regime**: Choose between training a larger model for fewer steps and training a smaller model for more steps when a team has fixed compute but limited training data.
- ☐ **Scaling breakdown**: Identify the scaling breakdown that follows from doubling model parameters (P) while keeping dataset size (D) constant.
- ☐ **Chinchilla optimality**: Explain why Chinchilla optimality serves as a resource-allocation target and identify the minimized quantity.
- ☐ **Compute vs. capability**: Classify the claim as true or false: scaling laws predict that doubling compute always yields a doubling of model capability.

These breakdown points demonstrate that scaling laws describe empirical regularities under specific conditions, conditions that become difficult to maintain at scale. Discerning *where* and *why* scaling ceases to be effective drives the development of strategies that enhance performance without relying solely on scale.

1.3.7 Integrating efficiency with scaling

Scaling laws reveal the walls; efficiency engineering builds the paths around them. Data saturation, infrastructure bottlenecks, and diminishing returns set hard limits on what brute-force scaling can achieve. The same constraints, however, point toward solutions along three interconnected dimensions:

- **Algorithmic efficiency**: Techniques such as sparsity and distillation extract more capability per FLOP.
- **Data efficiency**: Curriculum learning and active selection extract more information per training example.
- **Systems efficiency**: Communication hiding and pipeline overlap extract more utilization per accelerator.

Each dimension addresses a different breakdown condition from table 1.2, and each receives detailed treatment in the chapters that follow.

Scaling laws and efficiency techniques determine *how much* computation large models require and *how well* the fleet uses it. They do not, however, address the physical and logical constraints that the fleet itself imposes. Hardware fails, bandwidth saturates, distributed systems face fundamental impossibility results, and societal impact at scale demands governance. These constraints *cannot* be optimized away; they must be engineered around.

1.4 Constraints of Scale

The three walls describe where performance binds; the constraints behind them are broader. The Machine Learning Fleet operates under three categories of irreducible constraint: physical (the memory, network, and energy walls themselves, plus the reliability gap that opens as fleet size grows), logical (distributed systems face impossibility results like the CAP theorem), and societal (scale amplifies the impact of every technical decision). Understanding these constraints is a prerequisite for the diagnostic framework that follows.

1.4.1 The reliability gap

In traditional software, we treat hardware as a reliable abstraction. A single server typically has an availability of "four nines" (99.99 percent), meaning it fails for roughly 53 minutes a year. The Machine Learning Fleet, however, operates at a scale where this abstraction collapses, opening the **reliability gap**: hardware that is reliable in isolation becomes unreliable as a fleet. When a fleet coordinates 25,000 GPUs, equation 1.1 multiplies the individual component probabilities to give the probability that the entire system is healthy ($R_{\text{system}}(t)$):

$$R_{\text{system}}(t) = (R_{\text{component}}(t))^N = e^{-N\lambda t} \tag{1.1}$$

Here, N is the number of independent components in the fleet, λ is the per-component failure rate per unit time, t is the time window, and $R_{\text{component}}(t)$ is the probability that one component survives that window. The engineering implication is the exponent: every added component multiplies the failure opportunity, so reliability engineering becomes a scaling problem rather than a cleanup task.

If each node in a cluster is 99.9 percent reliable, a 1,000-node cluster is healthy only 36.8 percent of the time. Scale that to a 10,000-node fleet, and the probability of the entire system being healthy at any given second drops to 0.0045 percent. Figure 1.11 traces this exponential decay for two per-node reliability levels across fleet sizes from 1 to 10,000 GPUs.

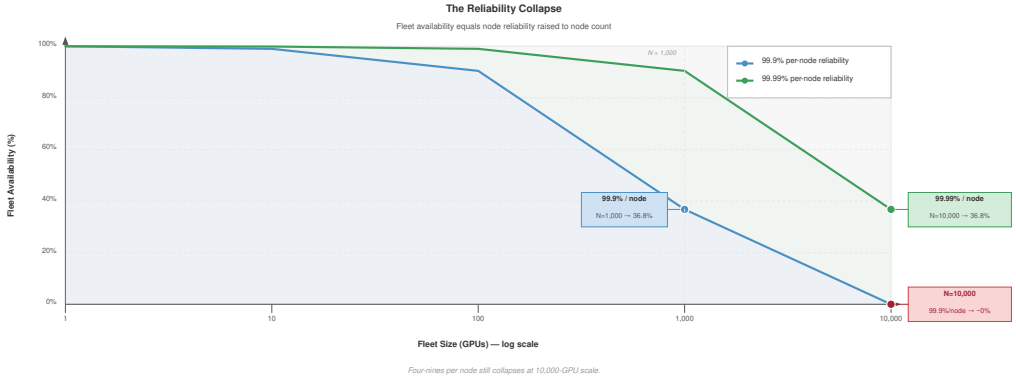


Figure 1.11: The Reliability Collapse: Fleet availability as a function of fleet size for two per-node reliability levels. At 99.9% per-node reliability (blue), a 1,000-GPU cluster retains only 36.8% fleet availability; at 10,000 GPUs, availability collapses to near zero. Even 99.99% per-node reliability (green) drops to 36.8% at 10,000-GPU scale.

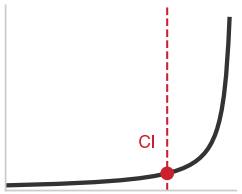
The exponential decay in figure 1.11 makes the architectural consequence inescapable: no amount of per-node improvement can sustain fleet-wide availability at scale. Here we establish the idea with a single per-node reliability figure; section E.1 derives the full failure-probability cascade and the fleet-wide availability formula a reader needs to size checkpoint intervals and redundancy. The engineering lesson is that *failure is the common case*. At scale, resilience replaces prevention as the governing objective: the system trades absolute uptime for recovery speed, checkpoint quality, and automated repair. The defining challenge of Chapter 7 is this shift from keeping every component running to ensuring that the fleet self-heals when components fail.

1.4.2 Communication intensity (the CI ratio)

If the single-machine iron law governs *how* a system executes, **communication intensity (CI)** governs *where* it stalls. On a single accelerator, the roofline model determines whether a kernel is compute-bound or memory-bound. At fleet scale, this analysis elevates to the network.

Equation 1.2 defines communication intensity as the ratio of data moved across the network to the operations performed locally:

$$\text{CI} = \frac{\text{Bytes Transferred (Network)}}{\text{FLOPs Executed (Local)}} \tag{1.2}$$



Past the communication-intensity cliff, adding GPUs stops helping.

Low communication intensity, $\text{CI} < 0.01$, describes compute-heavy workloads whose GPUs spend most of their time doing math, so scaling is relatively easy. High communication intensity, $\text{CI} > 0.1$, describes network-bound workloads limited by bisection bandwidth, where adding more GPUs can slow training rather than speed it up.

Communication and scaling optimizations, from gradient sparsification (sending fewer gradient bytes) to 3D parallelism (partitioning data, tensor, and pipeline dimensions across different fabric tiers), try to lower exposed communication intensity so that the Machine Learning Fleet acts as a single, massive computer rather than a collection of idling processors waiting for the wire. We

define the ratio here and apply it qualitatively; section C.3.3 works communication intensity through concrete regimes and shows the bandwidth-saturation point at which adding accelerators stops helping.

Checkpoint 1.3: The scale mandate

These questions check whether the scale mindset has shifted from local execution to distributed constraints:

- ☐ **Communication intensity**: Explain in CI-ratio terms why adding more nodes (N) to a high-CI workload can slow training rather than speed it up.
- ☐ **Reliability gap**: Explain the reliability gap that prevents a 10,000-GPU cluster from remaining "perfectly healthy."
- ☐ **Arithmetic vs. communication**: Distinguish arithmetic intensity (local memory) from communication intensity (global network).

1.4.3 Why distribution is hard

Scale forces distribution: no single machine provides the compute required for the largest models, and no centralized system can collect every data source that global user bases generate. Coordinating computation across physically separated machines connected by finite-bandwidth networks, however, creates constraints that no engineering can eliminate.

The first stressor appears inside data centers, where synchronous training must choose how much consistency to preserve under partitions and failures. The second appears at the edge, where the devices themselves are unreliable, power-limited, and policy-constrained.

1.4.3.1 The CAP theorem reality

The **CAP Theorem**[11] establishes that distributed systems can provide at most two of three properties: **Consistency**, **Availability**, and **Partition Tolerance**. The ML fleet encounters all three constraints, forcing explicit trade-offs.

Distributed ML systems make different trade-offs depending on their requirements. Synchronous training chooses consistency: all workers see the same model state, but training halts if any worker becomes unreachable. Asynchronous training chooses availability: training continues even with stragglers, but workers may operate on stale model versions. Federated learning often chooses availability with eventual consistency, accepting temporary divergence for continuous operation on edge devices.

[11] **CAP Theorem** (Brewer's Theorem): Brewer formulated the conjecture (Brewer 2000), and Gilbert and Lynch gave the formal proof that in the face of a network partition (P), a system must choose between absolute Consistency (C) and high Availability (A) (Gilbert and Lynch 2002). For the ML Fleet, this manifests as a choice between **Synchronous Training** (Consistency prioritized; the job stalls if a node fails) and **Asynchronous Training** (Availability prioritized; training continues but with stale gradients).

1.4.3.2 Edge distribution complexity

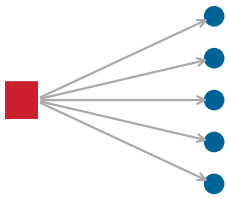
The coordination challenges discussed so far assume data-center distribution, where machines run in managed facilities. At the network edge, these challenges intensify along every dimension.

Billions of smartphones and IoT devices operate in uncontrolled environments with unreliable connectivity and limited power. Privacy, consent, or policy constraints may prevent raw data from leaving the device, making federated learning a natural architecture (McMahan et al. 2017). Intermittent connectivity forces the system to tolerate asynchronous updates spanning days. These constraints require architectural approaches (differential privacy, on-device inference, and model compression) that differ fundamentally from data-center ML.

When ML systems operate at the scale of billions of users, their societal impact demands consideration beyond technical excellence. Governance becomes an engineering requirement.

1.4.4 Why governance matters at scale

Scale and distribution amplify impact beyond engineering. When a system serves billions of users, a technical bug becomes a societal risk. This amplification creates governance requirements that small-scale systems can safely ignore. We frame governance as the **Control Plane** of the Machine Learning Fleet, not a set of external rules.



At fleet scale, a technical bug becomes societal risk.

That control plane first has to defend the fleet. ML systems face unique security threats that intensify at production scale. Model extraction attacks steal proprietary intellectual property through API queries. Data poisoning injects backdoors into models that remain dormant until triggered by a specific input. At fleet scale, these threats become economically attractive targets for sophisticated attackers. Defending the fleet requires systematic approaches: access controls, differential privacy[12], and continuous behavioral monitoring that go far beyond traditional perimeter security.

12 **Differential Privacy (DP)**: A mathematical framework for adding calibrated noise to computations to bound information leakage about individual data points. Chapter 13 examines how DP protects user privacy in large-scale ML systems.

The same control plane must make the fleet auditable. Systems operating at scale attract regulatory attention. Regimes such as the **EU AI Act** and local privacy laws make **Auditability** an architectural requirement for many deployments. Proving that a model trained on 10,000 GPUs did not ingest prohibited data requires end-to-end data lineage tracking. Generating a human-interpretable explanation for a sub-millisecond recommendation demands techniques designed into the serving architecture. Meeting these requirements calls for technical capabilities (audit trails, bias testing, and consent management) built into the infrastructure from day one.

Beyond legal compliance, the Machine Learning Fleet carries ethical obligations. Recommendation algorithms shape public discourse; hiring algorithms affect livelihoods. These systems do not fail loudly with a crash log; they fail silently through bias and polarization. Responsible AI is the engineering practice of treating fairness, transparency, and accountability as invariants: hard constraints that, if violated, should trigger a system-wide halt.

1.5 The C^3 Taxonomy: Foundations of Scale

The physical, logical, and societal constraints just surveyed all manifest as wasted wall-clock time, so navigating them demands a diagnostic that says where that time actually goes. The single-machine foundation for that analysis is the **Data · Algorithm · Machine (D·A·M) taxonomy**, which Appendix A develops as a complete diagnostic framework. Data is the information the system learns from, and performance becomes data-bound when the I/O pipeline cannot feed the processor fast enough. The algorithm is the mathematical logic being executed, and performance becomes compute-bound when arithmetic units are the limiting factor. The machine is the physical hardware substrate, and performance becomes memory-bound when memory bandwidth or capacity limits throughput.

Scaling from one machine to a fleet introduces three new fundamental resources that compete for wall-clock time. Where the canonical triad of walls names where the fleet binds, these three resources name where its time is spent. The **C^3 Taxonomy** extends the D·A·M lens to the fleet, identifying the physical and logical boundaries of the Machine Learning Fleet. Compute (C_1, the math) is the local execution of matrix operations on individual accelerators, where the goal is to keep the math engine running at peak utilization. Communication (C_2, the wire) is the movement of data across the network fabric, governed by bisection bandwidth and the speed of light. Coordination (C_3, the logic) is the synchronous management of state across thousands of nodes, where collective algorithms, fault tolerance, and distributed consensus determine how efficiently N independent nodes can act as a single computer.

These three dimensions form the **Triad of Distributed Efficiency**. If the fleet spends too much time on communication or coordination, the expensive compute capacity sits idle. The central challenge is engineering the fleet to minimize the C^3 Gap: the difference between theoretical hardware peak and actual distributed throughput. We introduce the three dimensions here as a framing device; Appendix B develops the complete taxonomy and the diagnostic structure that lets a reader localize a bottleneck to one of the three C's.

1.5.1 The projection: From D·A·M to C^3

The D·A·M taxonomy describes the *workload*: the nouns of our system (the data we have, the algorithm we want to run, the machine we bought). The C^3 taxonomy describes the *execution tax*: the verbs of the fleet (computing the math, communicating the results, coordinating the state). Appendix A reviews the single-node D·A·M lens; the C^3 taxonomy projects that workload onto a distributed fleet. Stretching an algorithm across nodes creates communication, such as AllReduce operations to synchronize gradients. Stretching data across nodes requires coordination through distributed samplers and checkpointing. Machine failures in a fleet require coordination through fault tolerance and straggler mitigation.

The intersection of these two taxonomies defines the entire design space of fleet-scale ML systems engineering. We do not solve D·A·M or C^3 *in isolation*; we solve their *cross-products*.

1.5.2 The fleet law

The C^3 taxonomy yields a diagnostic equation for every distributed training step. To reason about performance at scale, we must distinguish the efficiency of a single component from the efficiency of the entire system. On a single machine, execution time is often governed by the iron law of performance, $T \approx D_{\text{vol}}/\text{BW} + O/(R_{\text{peak}}\eta_{\text{hw}}) + L_{\text{lat}}$, decomposing into data movement, computation, and overhead. At fleet scale, that single-machine view no longer suffices: we care about the efficiency of the fleet across time and energy, and a parallel decomposition emerges. Equation 1.3 states the **Fleet Law** (the distributed step time law), which decomposes distributed execution into its canonical distributed-step components:

$$T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}} \tag{1.3}$$

where T_{compute}/N is the idealized local arithmetic time after distributing work across N devices, $T_{\text{comm}}(N)$ is the time moving data across the network (gradient synchronization, parameter broadcasts), $T_{\text{sync}}(N)$ is time consumed by synchronization logic (barriers, scheduling decisions, fault recovery), and T_{overlap} is the communication hidden behind useful compute. Equation 1.4 rewrites this decomposition as the compute-time fraction, defined as the fraction of wall-clock time spent on useful math:

$$f_{\text{compute}} = \frac{T_{\text{compute}}/N}{T_{\text{step}}(N)} \tag{1.4}$$

This decomposition is a diagnostic instrument. When f_{compute} drops below 0.5, more than half the fleet's expensive silicon sits idle. The C^3 taxonomy identifies *where* the time goes: if $T_{\text{comm}}(N)$ dominates, upgrade interconnects or overlap communication with computation; if $T_{\text{sync}}(N)$ dominates, consider asynchronous methods or reduce barrier frequency; if T_{compute}/N dominates, the fleet is doing its job. Crucially, the **Displacement of Overhead** dictates that overhead *cannot* be eliminated, only relocated among the three C's. Asynchronous training eliminates coordination barriers but introduces gradient staleness; pipeline parallelism reduces communication volume but adds bubble time. There is no free lunch in distributed systems.

Google's **ML Productivity Goodput** metric provides the production-scale instantiation of this framework. Goodput decomposes end-to-end training productivity into three multiplicative factors: **Program Goodput** (η_{program}), measuring how efficiently code uses hardware (Compute); **Runtime Goodput** (η_{runtime}), capturing losses from communication stalls and failures (Communication); and **Scheduling Goodput** ($\eta_{\text{scheduling}}$), capturing wasted time from preemptions and reconfigurations (Coordination). The C^3 taxonomy is the theoretical framework; Goodput is how production teams measure it.

Figure 1.12 visualizes the C^3 framework as a three-column layout, with Compute, Communication, and Coordination each occupying their own column of subtopics and the inter-column gaps highlighting the bottlenecks that arise where the dimensions meet. The same framework can be drawn as a triangle in which the vertices are the three resources, the edges are the trade-offs, and the center embodies the displacement of overhead—the unavoidable consequence of distributing computation across independent machines.

The fleet law is the time-side diagnostic of fleet efficiency across time and energy: it asks how much of each distributed step remains useful compute after communication, synchronization, and overlap are accounted for. Section B.3 develops the full derivation and attributes each term to a physical cause.

1.5.3 The energy-scale invariant

While the fleet law governs time, the **Energy-Scale Invariant** governs the sustainability and economic viability of the fleet. At scale, every training step is a thermodynamic event. Equation 1.5 defines the

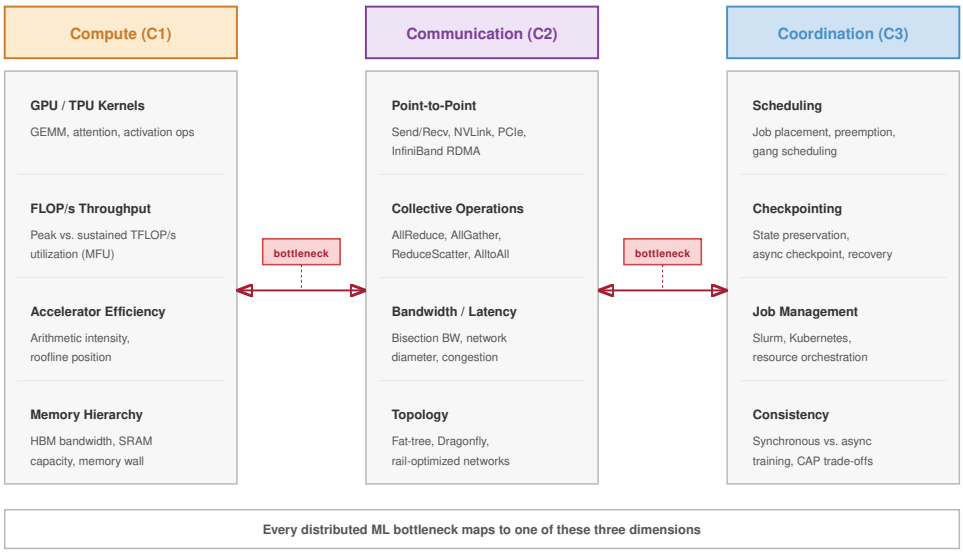


Figure 1.12: The C^3 Taxonomy: Every distributed ML system partitions wall-clock time among three irreducible dimensions: Compute (the useful math), Communication (data movement across the network), and Coordination (synchronization logic). The figure lays out each dimension as a column of subtopics, with the bottlenecks that arise between adjacent dimensions called out in the inter-column gaps. The framework's underlying claim—displacement of overhead—asserts that overhead cannot be eliminated, only relocated; this is the diagnostic lens for every performance analysis in this book.

fleet energy productivity (ρ_{energy}), measured in FLOP/J, as the ratio of useful work to total energy consumed:

$$\rho_{\text{energy}} = \frac{O_{\text{useful}}}{E_{\text{compute}} + E_{\text{cooling}} + E_{\text{network}}} \tag{1.5}$$

where O_{useful} is useful work in FLOPs and E_{network} often becomes a nonnegligible fraction of the total budget as we move terabytes across optical fabrics. Mastery of scale requires optimizing for the Pareto frontier of both laws: minimizing T_{step} while maximizing ρ_{energy}. We introduce the invariant here; section C.4.2 details the energy hierarchy at scale and the thermodynamic constraints that bound how far ρ_{energy} can be pushed.

The energy-side metric is only half of the scaling diagnosis. A production team also needs a time-side efficiency scalar that says whether extra devices are shortening the training step or merely increasing coordination overhead.

The scaling efficiency (η_{scaling}) of the fleet is the ratio of ideal parallel step time to actual step time:

$$\eta_{\text{scaling}} = \frac{T_1}{N \times T_N}$$

where T_1 is the single-device time for the same work and $T_N = T_{\text{step}}(N)$ is the step time on N devices. Section B.3.1 provides the component-decomposition technique and worked examples for splitting $T_{\text{step}}(N)$ into its terms, equipping a reader to compute η_{scaling} numerically.

> **Systems Perspective 1.2: Amdahl's distributed pitfall**
>
> The fleet law is a specialized form of Amdahl's Law. The maximum speedup of a distributed system is limited by its most tightly coupled component, usually the network synchronization. If a model spends 20 percent of its time waiting for the network ($T_{\text{comm}}(N)$), no amount of

faster GPUs can make it more than 5× faster, regardless of how many are added. Scale is limited by coordination, not compute alone.

Applying both laws to a GPT-3-class synchronization estimate makes the coordination tax and its thermodynamic counterpart visible in the same terms as the scaling equation: the same Ring AllReduce that starves the accelerators of time also burns energy on every step.



At scale only about 4 percent of step time is compute; sync dominates.

Napkin Math 1.2: The coordination and energy tax

Problem: What scaling efficiency and energy cost result when training GPT-3 (175B params) on a cluster connected by 100 Gb/s Ethernet versus 200 Gb/s InfiniBand?

Physics: To synchronize 175B parameters (FP16), a Ring AllReduce must move approximately 700 GB of data through each participating accelerator endpoint per iteration.

Analysis: The fleet law exposes the communication tax.

- **InfiniBand** (200 Gb/s): High bandwidth and low latency yield a scaling efficiency of only 4.1 percent, meaning the accelerators compute for just 4.1 seconds out of every 100 seconds while the rest is consumed by synchronization.
- **Ethernet** (100 Gb/s): Lower bandwidth and higher overhead collapse the scaling efficiency further to 2.1 percent, leaving the accelerators productive for roughly 2.1 seconds out of every 100 seconds.

Energy: The energy wall is the thermodynamic cost of the same synchronization. Each endpoint's gradient synchronization consumes approximately 84 J across the network fabric (at 15 pJ/bit). In a training run of 1,000,000 steps, the endpoint network movement alone accounts for about 84 MJ of energy before multiplying by the number of participating accelerators.

Systems insight: Scale makes the network the primary "processor." If the network is inefficient, the GPUs sit idle, wasting millions of dollars in compute capacity. Furthermore, the thermodynamic cost of data movement (the energy wall) means that "more GPUs" is not a sustainable scaling strategy without communication-hiding and compression.

1.6 Foundational Concepts

The fleet stack is the organizing spine for this book. The C^3 taxonomy introduced earlier (figure 1.12) gives the local diagnostic lens for one distributed step: *where* wall-clock time goes among compute, communication, and coordination. Scaling laws predict *how much* computation a target capability level demands, and governance constraints define *what* the fleet must never do. Those lenses support the stack rather than competing with it. When a chapter needs to trace how a change propagates, we will use the AI Triad at Scale; when it needs to assign ownership, we will use the Five-Pillar map. The dependency order comes from the fleet stack. Figure 1.13 organizes the complexity of this book into **The Fleet Stack**, a four-layer framework where engineering decisions at the bottom constrain possibilities at the top.

This layered progression structures the textbook's four parts: the physical substrate, the logic of distribution, deployment at scale, and the responsible fleet. The detailed chapter map appears in section 1.7; here the stack establishes the dependency order.

The **AI Triad at Scale** is the dependency reminder inside that stack. Every machine learning system comprises three interdependent components: data that guides behavior, algorithms that learn patterns, and computational infrastructure that enables both training and inference. At production scale, these interdependencies intensify. A GPT-4-class scenario at roughly 10^{25} FLOPs would demand infrastructure capable of efficient gradient synchronization across tens of thousands of GPUs. Training data at this scale requires distributed storage systems with access patterns optimized for ML workloads. Figure 1.14 visualizes these dependencies between data, algorithms, and infrastructure, revealing the optimization landscape that ML systems engineers must address.

The **Five-Pillar Framework** turns those layer-level constraints into ownership domains. Data engineering defines what the fleet must ingest, transform, and serve; model development defines the

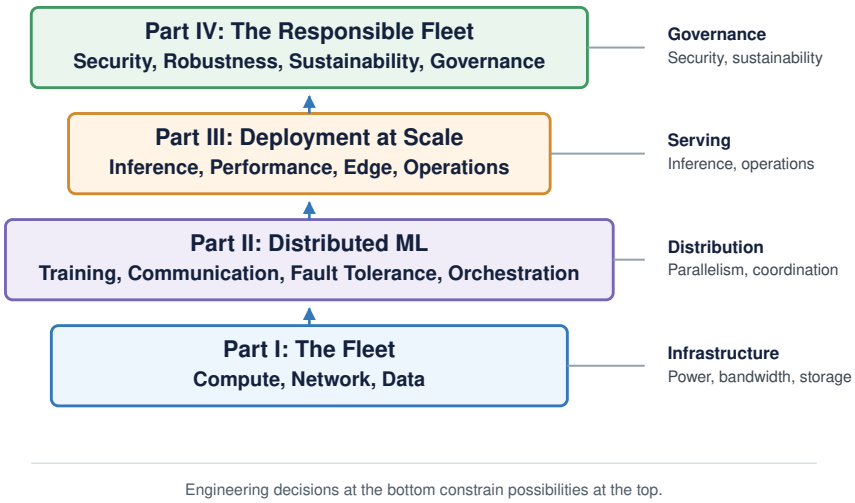


Figure 1.13: The Fleet Stack: The organizing framework for this book. We build from the Infrastructure Layer (compute, network, data) up through the Distribution Layer (parallelism, communication, fault tolerance) and Serving Layer (inference, performance, edge, operations) to the Governance Layer (security, robustness, sustainability, responsible engineering). Engineering decisions at the bottom constrain possibilities at the top.

architectures and training procedures that consume that data at scale. Optimization then bridges the model to its deployment constraints, compressing or accelerating the workload when latency, memory, or power budgets become binding. Deployment infrastructure supplies the cloud platforms, network fabrics, and storage hierarchies that make the fleet runnable. Operations (MLOps) closes the loop by measuring whether the deployed system remains reliable, secure, and effective as production data and hardware conditions change.

Production frameworks matter because each implements the same abstract primitives with different names and ownership boundaries. Table 1.3 provides a cross-framework mapping as a translation aid for the primitives developed across the fleet stack, not as a tool ranking.

Table 1.3: The Framework Rosetta Stone: A mapping of abstract distributed ML primitives to their implementations across major frameworks. The primitives are developed where they become binding constraints, including distributed training (Chapter 5), collective communication (Chapter 6), inference (Chapter 10), and operations (Chapter 12).

Abstract Primitive	PyTorch (Native)	DeepSpeed	Megatron-LM	JAX/XLA	Ray
Data Parallel	DDP	DeepSpeedEngine	DistributedDataParallel	pmap/jit(sharded)	Ray Train
Sharded DP	FSDP	ZeRO-1/2/3	FullyShardedDataParallel	sharding.Mesh	FSDP Strategy
Tensor Parallel	DTensor	InferenceTP	Column/RowParallel	xmap/spmd	Ray Train TP
Pipeline Parallel	PiPPy	PipelineModule	PipelineParallel	GSPMD	Ray Train PP
Grad Accumulation	backward(accumulate)	grad_accum_steps	grad_acc_steps	lax.scan	Ray Train Config
Checkpointing	torch.save/DCP	save_checkpoint	save_checkpoint	orbax	ray.checkpoint
Orchestration	torchrun	deepspeed CLI	megatron_main.py	jax.distributed	Ray Core

These primitives recur throughout the book; the table is a reference to return to as each one is introduced, not a syllabus to memorize now. The **Six Systems Engineering Principles** provide the standard for individual design decisions: instrument first, design for 10× headroom, expect

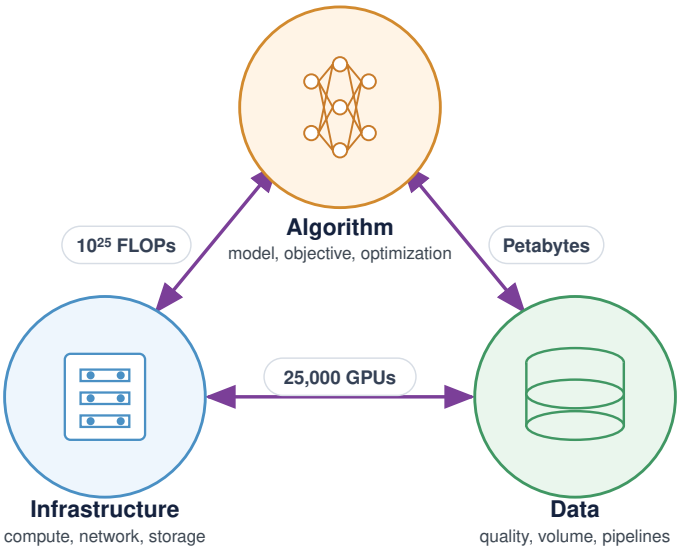


Figure 1.14: The AI Triad at Scale: The three interdependent components of every ML system. At production scale, each component's requirements intensify: data pipelines must handle petabytes with consistent quality; algorithms can demand 10^{25} FLOPs for large-model training; and infrastructure must coordinate thousands of accelerators while maintaining fault tolerance. Changes to any vertex cascade through the others, creating the multi-dimensional optimization challenge that defines ML systems engineering.

bottlenecks to migrate, treat failure as steady state, compound small efficiency gains across the fleet, and co-design hardware with algorithms. At fleet scale, a claim about performance or reliability is only useful when it is instrumented, because the cause may sit in the model, the scheduler, the network, the storage tier, or the serving path. That measurement discipline supports $10\times$ design: a prototype is not yet an architecture until it shows where headroom remains. As scale rises, bottlenecks move from local compute toward communication and coordination, hardware failures become routine, and small efficiency gains become material because they repeat across the fleet. Hardware co-design follows from the same logic, since network topology, storage hierarchy, and accelerator choice shape which algorithms remain practical. These are the practices a fleet engineer applies; the volume closes by distilling what they ultimately serve into the durable principles of distributed ML systems.

Together, these frameworks assume familiarity with single-machine ML systems: *how* models are trained, optimized, and deployed on individual devices. This book teaches engineers to scale, distribute, and govern them across the global Machine Learning Fleet.

1.6.1 Three systems archetypes

Abstract frameworks become concrete through specific workloads. This book traces three **Lighthouses at Scale** through every chapter. Each archetype occupies a distinct corner of the C^3 taxonomy, stressing communication, coordination, or compute in fundamentally different ways, ensuring that every principle is tested against the diversity of real-world fleet engineering.

1.6.1.1 Archetype A (GPT-4/Llama-3)

GPT-4 and Llama-3 are autoregressive transformer architectures that generate text one token at a time. At single-accelerator scale, autoregressive language models are memory-bandwidth probes; at the scale of hundreds of billions of parameters, and in some proprietary systems potentially more, these models define the *throughput-bound* regime: training demands ExaFLOP/s of sustained compute distributed across thousands to tens of thousands of accelerators, while serving requires memory bandwidth sufficient to load billions of weights for every generated token. The fleet

challenge is partitioning very large models across accelerator fleets using 3D Parallelism (data, tensor, and pipeline) without the network fabric becoming the binding bottleneck. In the C^3 taxonomy, Archetype A is dominated by communication—gradient synchronization and activation transfers across the cluster consume more wall-clock time than the arithmetic itself.

1.6.1.2 Archetype B (DLRM at scale)

DLRM is Meta's production architecture for personalized content ranking. Unlike LLMs, which are dominated by dense matrix multiplications, recommendation models derive their capacity from massive embedding tables—sparse lookup structures that map billions of user and item features into dense vectors. A single production DLRM instance may contain 10 TB or more of embedding parameters, far exceeding any single accelerator's memory. The fleet challenge is sharding these embedding tables across hundreds of nodes while processing millions of queries per second (QPS) with <100 ms tail latency, all while managing $\mathcal{O}(N^2)$ all-to-all communication contention between embedding shards and dense layers. In the C^3 taxonomy, Archetype B stresses coordination: sparse feature routing, shard placement, request scheduling, and embedding-update consistency determine whether the all-to-all traffic can be served within tail-latency limits.

1.6.1.3 Archetype C (federated MobileNet)

MobileNet is a family of efficient convolutional neural networks designed for on-device inference under tight power and memory budgets. In a federated learning setting, MobileNet instances train locally on millions of heterogeneous edge devices (smartphones, IoT sensors, medical wearables) without raw data ever leaving the device. The constraint regime is qualitatively different from the data-center archetypes: compute budgets are measured in *milliwatts* rather than *megawatts*, and privacy or policy constraints can prohibit centralized data aggregation. In the C^3 taxonomy, Archetype C is dominated by compute at the per-device level: each local training step is bound by the milliwatt-scale silicon envelope of the edge device, not by fleet-level network bandwidth or coordination protocols. The fleet challenge is coordinating model updates across millions of such compute-constrained, unreliable devices using Federated Averaging, which periodically averages device updates at a coordinator, while maintaining convergence guarantees despite data distributions that are not independent and identically distributed (i.i.d.) and intermittent connectivity.

Read the three archetypes as constraint probes, not as a catalog of model families. The LLM case asks whether dense synchronization can keep thousands of accelerators useful, the DLRM case asks whether sparse feature routing can meet tail-latency budgets, and the federated MobileNet case asks whether local device constraints can be coordinated into a statistically coherent update. Table 1.4 summarizes the three canonical workloads tracked throughout this book.

Table 1.4: Lighthouse Archetypes at Scale: The three canonical workloads tracked throughout this book, with their dominant constraint and the fleet-scale challenge each raises.

Archetype	C^3 Dominant Constraint	Fleet Challenge
Archetype A (GPT-4/Llama-3)	Communication (fleet-wide)	Partition hundreds of billions of parameters, and potentially larger proprietary models, across thousands to tens of thousands of GPUs using 3D Parallelism without the network becoming the bottleneck.
Archetype B (DLRM at Scale)	Coordination (routing and placement)	Shard 10 TB+ embedding tables across hundreds of nodes; process millions of QPS with <100 ms tail latency while managing sparse feature routing, shard placement, and $\mathcal{O}(N^2)$ all-to-all contention.
Archetype C (Federated MobileNet)	Compute (per-device envelope)	Coordinate learning across millions of compute-constrained, unreliable edge devices using Federated updates; raw data cannot leave the device.

1.7 The Structure of This Textbook

This textbook organizes around the fleet stack, progressing from the physical substrate through the logic of distribution to societal governance. The chapter sequence is a dependency map rather than a topic inventory: the physical constraints of silicon and cooling dictate how we wire the network; the network limits dictate how we partition training algorithms; partitioned algorithms dictate how we

handle fault tolerance and orchestration; and fleet orchestration dictates how we operate, secure, and govern systems in production. Each part addresses a fundamental scale impediment that prevents a single-machine solution from working at production scale.

The infrastructure arc begins with the impediment no algorithm can ignore: no single server has enough memory, power, cooling, or I/O to train the largest models. Chapter 2 therefore starts with high-density silicon and facility physics, Chapter 3 turns isolated machines into a cluster-scale "Gradient Bus," and Chapter 4 completes the substrate by keeping accelerators fed without letting storage become the hidden bottleneck.

The distribution arc begins once the physical fleet exists and the coordination tax becomes unavoidable. Splitting math across machines creates two new problems: replicas must exchange state fast enough to behave like one optimizer, and failures become routine rather than exceptional. Chapter 5 develops partitioning strategies for models with hundreds of billions to trillions of parameters, Chapter 6 explains the communication primitives that bind independent nodes into a coherent computer, Chapter 7 treats recovery speed as more important than individual-node uptime, and Chapter 8 manages the multi-tenant cluster where those jobs must be placed, isolated, and rescheduled.

The deployment arc shifts from training a model to serving it economically. Once a model reaches production, the binding question is no longer only time-to-train; it is whether latency, utilization, and lifetime serving cost stay inside the operating envelope. Chapter 9 closes the gap between hardware peak and actual throughput through kernel fusion and compilation, Chapter 10 turns those local optimizations into serving systems for millions of users, Chapter 11 moves intelligence toward devices with milliwatt power budgets, and Chapter 12 develops the control plane for monitoring health, drift, and performance across global deployments.

The governance arc treats responsibility as an engineering layer rather than an afterword. At global scale, technical bugs become societal hazards: adversaries can poison data or extract weights, open-world inputs can break brittle models, energy use becomes a fleet constraint, and governance failures can turn technical capability into social harm. Chapter 13, Chapter 14, Chapter 15, and Chapter 16 develop those obligations as system properties that must be designed, measured, and operated.

1.8 Fallacies and Pitfalls

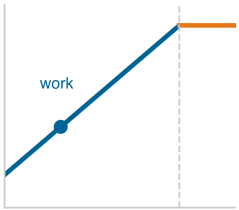
The following fallacies and pitfalls capture architectural mistakes that waste development resources, miss performance targets, or deploy systems critically mismatched to their operating constraints. Each represents a pattern we have seen repeatedly in the transition from single-machine ML to the Machine Learning Fleet.

Fallacy: *Focusing on algorithmic efficiency while ignoring hardware-system alignment.*

Engineers often optimize FLOPs and parameter counts assuming these metrics predict deployment performance. Real efficiency depends on *how* well the math aligns with the underlying hardware. For example, unstructured pruning achieves 80 percent sparsity but delivers no speedup on dense hardware (like NVIDIA Tensor Cores), while structured pruning at 50 percent sparsity can enable up to 2× sparse-math throughput on supported hardware and kernels, but does not guarantee end-to-end speedup. A model reduced from 10B to 3B parameters (70 percent FLOPs reduction) might achieve only 20 percent latency improvement because memory bandwidth bottlenecks dominate and the pruning pattern lacks hardware-friendly structure.

Pitfall: *Optimizing one metric without checking system-wide trade-offs.*

The **Universal Efficiency Fallacy** assumes that optimizations like quantization or distillation are "free wins." In production, each optimization introduces specific trade-offs. INT8 quantization achieves 4× memory reduction but typically incurs 1–2 percent accuracy loss. Knowledge distillation enables 2–4× compression but demands expensive teacher model training. The optimal architecture requires balancing accuracy, latency, and power, not merely minimizing resource consumption.



Cutting FLOPs alone leaves latency memory-bound.

Fallacy: *Edge deployment efficiency requirements are simply scaled-down versions of cloud requirements.*

This **"Cloud-Lite" Fallacy** treats edge systems as resource-constrained cloud systems. Edge devices face qualitatively different constraints. For example, autonomous vehicles at 120 km/h convert every 100 ms of processing delay into 3.33 m of positional uncertainty. While cloud deployments scale to kilowatts, edge systems operate under 5–15 W power budgets. A cloud-optimized model

with 95 percent accuracy and 50 ms latency might be unusable on an edge device where thermal throttling increases latency to 200 ms and drains the battery in under an hour.

Pitfall: *Assuming scaling laws predict efficiency requirements linearly across all scales.*

The **Linear Scaling Fallacy** occurs when teams extrapolate resource requirements using power-law relationships without accounting for coordination overhead. $\mathcal{L}(X) = A_{\mathcal{L}} X^{-\alpha_{\text{scale}}} + C_{\mathcal{L}}$ works within validated ranges but fails at the boundaries. A team training 100B-parameter models by extrapolating from 10B-parameter experiments might predict a 3× improvement but achieve only 1.8× if coordination and communication overhead consumes 40 percent of the scaled step time. Production systems designed assuming linear scaling have experienced 2–3× cost overruns when empirical performance deviated from power-law predictions beyond validated thresholds.

1.9 Summary

This volume opens with a fundamental challenge: the principles that enable success on single machines become the obstacles that prevent success at scale. We have moved from the laboratory to the Machine Learning Fleet, where communication costs dominate computation, failures are a statistical certainty, and societal impact demands rigorous governance.

The transition from building systems that *work* to building systems that *scale* represents the next regime of engineering. The core principles of machine learning systems (measure everything, optimize the bottleneck, design for failure) remain essential, but their application changes fundamentally when the system spans thousands of nodes. Network topology becomes as important as memory hierarchy, and fleet coordination replaces local synchronization: updates, failures, and shared state must be managed across many machines.

These principles collectively reshape *how* engineers reason about system design. When failure is routine rather than exceptional, and when communication cost eclipses computation, the diagnostic instincts developed on single machines must be recalibrated. The engineer who internalizes these constraints moves from local speedup questions to fleet-governance questions: where coordination overhead appears, what fails when a node disappears, and which consistency guarantee can be relaxed. That shift, from optimizing a component to governing a fleet, is what separates practitioners who can architect at scale from those who can only prototype on one machine.

Key Takeaways: Scale makes a new machine

- **More GPUs change the problem**: Techniques that work on one node stop being merely slower at fleet scale; they can become wrong. Bisection bandwidth, power delivery, reliability, and governance create constraints that do not appear in single-accelerator experiments.
- **The network spends every speedup**: Adding accelerators divides local compute but adds synchronization, gradient traffic, and tail-latency exposure. A 175B-parameter model can move hundreds of gigabytes of gradients per step, so useful FLOPs depend on communication, not peak arithmetic alone.
- **Failure becomes steady state**: A 25,000-GPU training run with per-GPU MTBF measured in tens of thousands of hours still sees cluster-level failures every few hours. Checkpointing, recovery, and observability are baseline design requirements, not cleanup work.
- **C^3 replaces local intuition**: The single-machine Data, Algorithm, Machine lens projects to Compute, Communication, and Coordination at fleet scale. The fleet law makes this shift explicit: scaling depends on synchronization, overlap, and consistency as much as hardware count.
- **Impact needs a control plane**: Foundation-scale systems amplify security, privacy, fairness, and policy failures across large user populations. Governance is not an appendix to infrastructure; it is the mechanism that decides what the fleet is permitted to optimize and deploy.

Everything this introduction has argued converges on one reversal. At scale, the techniques that made a single machine fast become the very things that hold a fleet back, because the binding constraint moves outside the box. On one accelerator, computation is the cost and the memory hierarchy sets the limit; across ten thousand, computation is the easy part, and the limit is everything between the machines, the communication that carries every gradient and the coordination that keeps thousands of nodes agreeing on one state. That is the physics of distribution, and it answers to its own three-letter law: where a single node is governed by Data, Algorithm, and Machine, the fleet is governed by Compute, Communication, and Coordination. The rest of this book works out that shift, from the silicon upward to the governance that decides what the fleet is permitted to do.

What's Next: From requirements to silicon

The requirements of scale are established: we know *why* we must distribute, *what* physical costs arise, and which principles govern the Machine Learning Fleet. Requirements alone, however, do not compute gradients. The fleet needs a physical foundation (silicon, power, and cooling) capable of sustaining the workloads we have described. Chapter 2 begins building that foundation, examining the accelerator architectures, memory hierarchies, and power delivery systems that form the Infrastructure Layer of the fleet stack.

Research Questions: For further inquiry

- How should the single-machine D·A·M lens change when a workload crosses into the fleet-scale C^3 regime?
- When can adding accelerators reduce useful training progress rather than improve it?
- How should engineers use scaling laws as planning tools without mistaking them for infrastructure guarantees?
- Why should governance be treated as a control plane of the fleet rather than an external policy layer?

I

THE FLEET

Part I

Fleet Principles

The Machine Learning Fleet is the warehouse-scale computer where the network is the bus, power density is the speed limit, and failure is a statistical certainty. Continuing the curriculum's focus on the physics of AI engineering, Part I builds the physics of scale: the silicon, the wires, the cooling systems, and the storage hierarchies that make distributed ML possible. We shift from the single accelerator to the data-center-scale machine, where the engineering question is no longer how to compute on one device but how to move energy and information at a scale that challenges the limits of the physical infrastructure.

This transition requires a fundamental shift in perspective. At scale, the individual GPU is merely a component in a larger, tightly coupled system. The principles of the Fleet are not best practices for cluster management; they are the physical invariants that dictate what kind of models can be trained and how they can be served. From the thermodynamic limits of heat dissipation to the bisection bandwidth of the network fabric, these constraints define the boundaries of the fleet stack. The governing pattern parallels the single node: where the node is bound by the conservation of complexity across Data, Algorithm, and Machine, the Fleet is bound by the displacement of overhead across Compute, Communication, and Coordination (the C^3 taxonomy). The execution tax of scale cannot be eliminated, only relocated.

Those boundaries appear first in the facility itself: every operation eventually becomes heat that must be removed.

Principle 1: The Thermodynamic Limit

Invariant: Irreversible computation and data movement dissipate electrical energy as heat; at data-center scale, the practical limit is the rate at which that heat can be removed.

$$\dot{Q}_{\text{limit}} = \frac{\Delta Q}{\Delta t}$$

where $\dot{Q}_{\text{limit}}$ is the maximum heat-removal rate, ΔQ is heat energy, and Δt is the removal time window.

Implication: The bottleneck of the modern AI data center is not FLOP/s, but watts per square foot. When a rack generates 100 kW of heat, air cooling fails. The physical design of the fleet is dictated by the ability to move heat away from silicon.

At the rack level, the same heat budget becomes a density constraint.

Principle 2: The Power Density Wall

Invariant: Cluster scaling is constrained by thermal dissipation limits (watts per rack) and cooling capacity, not just silicon area or floor space.

Implication: Modern AI accelerators generate heat densities that exceed air cooling capabilities. **Liquid cooling** becomes a facility requirement, not an option, for large-scale training clusters.

Thermal capacity sets the outer envelope; inside it, memory capacity determines how frontier models must be divided.

Principle 3: The Memory Capacity Gap

Invariant: Frontier model memory requirements have grown faster and more irregularly than single-accelerator HBM capacity.

Implication: Models no longer fit on single devices. Architectures must embrace 3D parallelism (splitting the model itself via tensor and pipeline parallelism) as the default state, breaking the abstraction of the "single device."

Once a model is divided across devices, the network becomes part of the execution path.

Principle 4: The Bisection Bandwidth Theorem

Invariant: The performance of a distributed application is limited by the minimum bisection bandwidth of the network topology.

Implication: A cluster with 10,000 GPUs cannot support synchronized distributed training if the accelerators are connected by a 1 Gbps Ethernet tree. **Non-blocking topologies** (fat-tree, Dragonfly) are required to ensure that the network does not become the bottleneck for collective operations like AllReduce.

The fleet also fixes where efficiency comes from: the choice between general-purpose flexibility and hardware specialization.

Principle 5: The Generality Tax

Invariant: For workloads with stable computation patterns, specialization can improve performance per watt by reducing general-purpose overhead, while also reducing programmability and workload flexibility.

$$\rho_{\text{energy,specific}} \gg \rho_{\text{energy,general}} \quad \text{for a fixed power budget}$$

Implication: The trajectory from CPU to GPU to Tensor Processing Unit (TPU) to fixed-function ASIC is a workload-dependent architecture trade-off, not a universal physical law. Each step can trade programmability for efficiency, and the fleet architect must choose the right point on this curve for each workload.

Compute efficiency helps only if the training pipeline can keep the accelerators fed.

Principle 6: The I/O Wall

Invariant: When storage throughput cannot deliver training data as fast as accelerators consume it, GPUs idle regardless of their computational power.

$$\text{Utilization} = \min\left(1, \frac{\text{BW}_{\text{storage}}}{N_{\text{GPU}} \times R_{\text{consumption}}}\right)$$

where $\text{BW}_{\text{storage}}$ is aggregate storage bandwidth, N_{GPU} is the number of accelerators, and $R_{\text{consumption}}$ is each accelerator's data-consumption rate.

Implication: A storage system that was adequate for 8 GPUs becomes the bottleneck at 64. The I/O wall scales with the number of accelerators: every GPU added to the cluster raises the throughput floor that storage must sustain, making the data pipeline, rather than the model, the limiting factor.

Meeting that demand requires a hierarchy rather than a single undifferentiated storage pool.

Principle 7: The Storage Hierarchy Principle

Invariant: Storage performance decreases and capacity increases as data moves further from the accelerator.

Implication: The systems engineer's task is to ensure that the *right data* is on the *right tier* at the *right time*. Data format choices, caching strategies, prefetch buffer sizing, and tiering policies all exist to manage movement upward through the hierarchy so that accelerators do not starve.

Even after data reaches the right tier, coordination limits how efficiently more nodes translate into more throughput.

Principle 8: The Scaling Efficiency Bound

Invariant: Adding nodes to a distributed training job yields diminishing returns because communication overhead grows with cluster size while per-node computation remains constant.

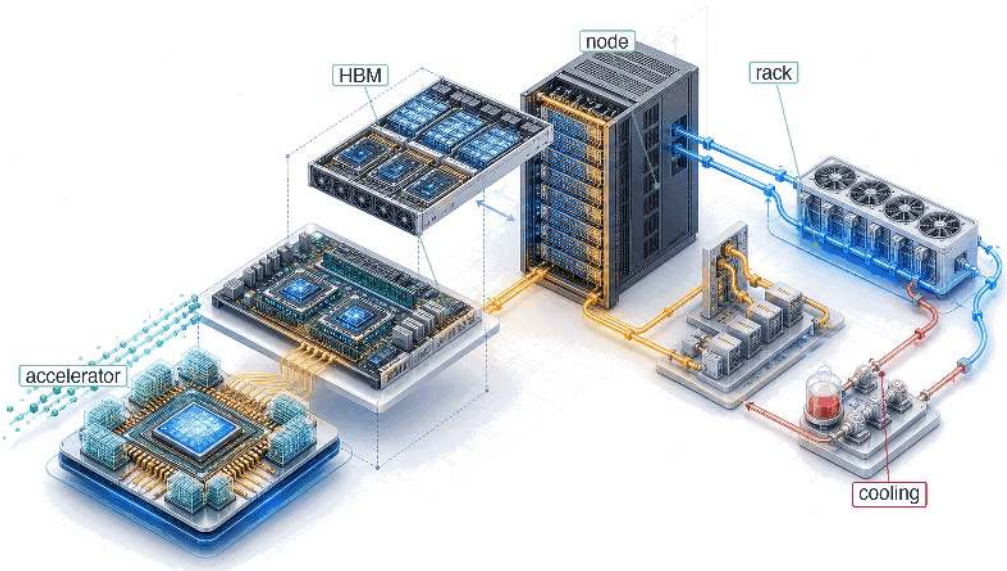
$$\eta_{\text{scaling}} = \frac{T_1}{N \times T_N} \leq 1$$

Implication: Perfect linear scaling ($\eta_{\text{scaling}} = 1.0$) is a theoretical limit, not a practical target. Well-tuned dense-training jobs commonly reach $\eta_{\text{scaling}} \approx 0.85$–$0.95$ at moderate scale on modern fabrics, and degrade further as N grows. The gap between $\eta_{\text{scaling}} = 1.0$ and the achieved efficiency is the **Communication Tax**—the price of coordination.

These invariants establish the Fleet as a first-class engineering object. Part I builds this machine from the ground up: from the landscape of distributed ML systems and why single machines no longer suffice, through the silicon, power, and cooling systems of the AI data center, the network fabrics that connect the fleet, and the storage hierarchy that feeds the training pipeline. Together, these chapters form the foundation for the lighthouse archetypes (section 1.6.1) that we will track throughout this volume.

2

Compute Infrastructure



2.1 The Infrastructure Walls
2.2 Accelerator Spectrum
2.3 The Memory Wall
2.4 The Node
2.5 The Rack
2.6 The Pod
2.7 Infrastructure Technologies That Move the Walls
2.8 Fallacies and Pitfalls
2.9 Summary

Purpose

Why does infrastructure, not algorithms alone, determine who can participate in advancing machine learning at scale?

Machine learning systems have a physical reality that transcends code. While the algorithms for training large models may be public, the ability to execute them at scale is gated by the physics of infrastructure. Every training run is a race against thermal limits, power delivery stability, and the cost of data movement. When algorithms are the primary bottleneck, progress can happen anywhere. At large scale, the constraint shifts toward who can build and operate the physical systems that make scale possible. Building an ML fleet requires engineering across four physical levels: the accelerator (silicon physics), the node (interconnect density), the rack (thermal management), and the pod (warehouse-scale networking). At each level, the goal is the same: to minimize the time data spends in transit and maximize the time it spends in computation. The economics are equally unforgiving: a large accelerator cluster represents major capital expenditure and megawatts of continuous power. Infrastructure becomes a participation boundary for organizations working at the largest scales. This chapter maps that physical stack, from the stacking of memory dies to the stabilization of grid-scale power ramps. In C^3 terms, this substrate is where the fleet's limits originate: every bound on compute and communication in the chapters that follow traces back to it.

Learning Objectives

- Compare GPU, TPU, and ASIC dataflows to explain specialization across the accelerator spectrum
- Apply roofline analysis to identify compute-, memory-, or bandwidth-bound limits for training and inference
- Calculate token latency or memory budgets from HBM capacity, bandwidth, precision, and model state size
- Map tensor, pipeline, and data parallelism onto NVLink, rack, and pod bandwidth tiers
- Evaluate rack and pod designs using power density, cooling limits, topology, and facility reliability constraints
- Select accelerator infrastructure using sustained throughput, utilization threshold, energy cost, and fleet ownership cost

2.1 The Infrastructure Walls

Scale forces distribution: Compute, Communication, and Coordination replace the single-machine instincts of Data, Algorithm, and Machine. Those fleet-level limits now have to be paid for in hardware. Four physical constraints recur as the system grows from one accelerator to a warehouse-scale pod: memory bandwidth, power delivery, communication bandwidth, and reliability. Figure 2.1 shows where each wall becomes dominant along that path.

The physical stack begins at the silicon die and expands outward through four levels. At each level, the same engineering pattern appears: a constraint becomes intolerable, and the solution creates the next layer of infrastructure. Nodes aggregate accelerators to overcome memory capacity limits. Racks concentrate nodes and confront power delivery and cooling. Pods wire racks into a warehouse-scale computer and face the communication wall at full force. The path from transistor to data center is anchored by the 175B model, which serves as the thread connecting each level.

2.2 Accelerator Spectrum

The foundation starts with the silicon, where a specialized chip earns its place by doing one thing a general-purpose processor cannot do efficiently: run the same dense arithmetic, over and over, at maximum throughput. Consider what happens when a CPU executes a matrix multiplication. The processor fetches an instruction, decodes it, checks for data hazards, routes operands through a deep pipeline, and writes the result back to a register file. For each multiply-accumulate operation, the chip expends energy on branch prediction, speculative execution, out-of-order scheduling, and cache coherence.

These mechanisms exist because a CPU must handle any instruction sequence efficiently, from pointer-chasing linked-list traversals to system calls. For neural network workloads, however, the operation is almost always the same: multiply two matrices of known dimensions, add a bias, and apply a nonlinear function. The CPU's elaborate control logic represents a tax on every operation, one that buys flexibility the workload does not need. This concept mirrors the classic reduced instruction set computer (RISC) vs. complex instruction set computer (CISC) debate in computer architecture: just as RISC processors achieved higher throughput by simplifying the instruction set, ML accelerators achieve higher throughput by simplifying the computational model to match the dominant workload pattern.

Systems Perspective 2.1: The generality tax

A modern server CPU devotes roughly 30–40 percent of its die area to caches, 20–30 percent to control logic (branch predictors, reorder buffers, instruction decoders), and only 5–10 percent to arithmetic units. This allocation makes sense for general-purpose code, where branches are unpredictable and data access patterns are irregular. For matrix multiplication, however, the access pattern is perfectly regular and the control flow is trivially predictable. Every transistor

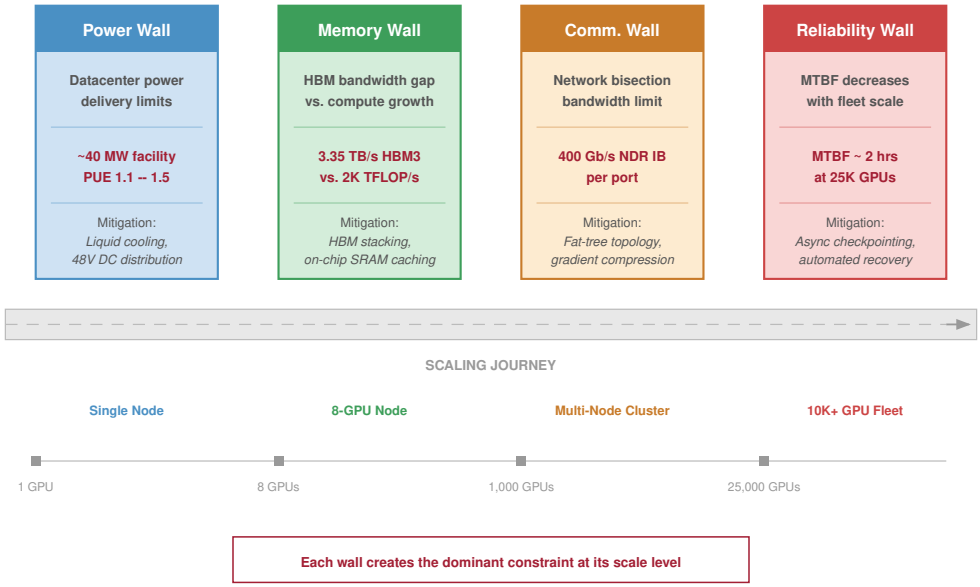


Figure 2.1: The Four Infrastructure Walls: Four dominant constraints arranged across a Scaling Journey: Power wall (data center delivery limits, ~40 MW facility), memory wall (HBM bandwidth gap, 3.35 TB/s HBM3 vs. ~2K TFLOP/s), communication wall (network bisection limit, 400 Gb/s NDR IB per port), and reliability wall (MTBF ~2 hours at 25K GPUs). Each wall becomes the dominant constraint at a different scale, from a single GPU through 8-GPU nodes and 1K-GPU clusters to 25K+ GPU fleets.

spent on branch prediction or speculative execution is a transistor that could have been a multiplier. This is the **generality tax** (principle 5): the silicon area that a processor wastes on capabilities irrelevant to the dominant workload.

The accelerator revolution is, at its core, an exercise in eliminating this tax. Each step along the spectrum from CPU to custom ASIC reclaims more die area for arithmetic by removing another layer of general-purpose control logic. The progression is quantifiable: a CPU devotes roughly 5–10 percent of its die to arithmetic, a GPU devotes roughly 50–60 percent, a Tensor Processing Unit (TPU) devotes roughly 70–80 percent, and a purpose-built ASIC can devote over 90 percent of its die to the target computation.

The transition from general-purpose to specialized silicon follows a logical progression. At one end of the spectrum, GPUs retain substantial programmability while providing 10–100× the matrix throughput of CPUs. At the other end, custom ASICs hardwire a specific dataflow for maximum efficiency at the cost of flexibility.

Understanding *where* each architecture falls on this spectrum, and *why*, is essential for selecting hardware that matches a given workload. The spectrum is not a ranking from "bad" (general-purpose CPU) to "good" (custom ASIC), but a continuum of trade-offs where each position offers the best match for a different combination of workload characteristics and deployment constraints.

GPUs represent the first major step away from general-purpose computing toward massive parallelism while retaining programmability. An NVIDIA H100, for example, contains 16,896 CUDA cores organized into 132 Streaming Multiprocessors (SMs). Each SM can execute thousands of threads simultaneously using the **Single Instruction, Multiple Threads (SIMT)**[1] execution model. Unlike a CPU, which optimizes for single-thread latency, a GPU hides memory latency

[1] **SIMT (Single Instruction, Multiple Thread)**: Coined by NVIDIA to distinguish their GPU model from classical single instruction, multiple data (SIMD). In SIMT, 32 threads form a *warp* that shares an instruction stream but can diverge at branches; divergence forces serial execution of both paths, halving throughput per branch. For ML workloads, where matrix operations have uniform control flow, warp divergence is rare, which is why GPUs achieve near-peak utilization on general matrix multiply (GEMM) operations but degrade on irregular operations like sparse attention or dynamic routing.

by maintaining thousands of threads in flight and switching between them when one stalls on a memory access.

The programmer writes a single function (a *kernel*), and the hardware maps it across a massive grid of threads. This model is flexible enough to run any mathematical kernel, from convolutions to attention mechanisms to custom loss functions, while providing orders of magnitude more throughput than a CPU for data-parallel computation.

The trade-off is that irregular, branch-heavy code runs poorly because divergent threads within a warp (a group of 32 threads that execute in lockstep) must execute serially. When threads in the same warp take different branches of an if-else statement, the hardware must execute both branches sequentially, disabling the threads that took the other path during each branch. For ML workloads, this divergence penalty is usually small because neural network operations have highly uniform control flow: every element in a matrix multiplication follows the same computation path.

TPUs go a step further, sacrificing some programmability for maximum efficiency on a single operation type by hardwiring the dataflow itself. Google's Tensor Processing Units use a **systolic array**[2] architecture: a fixed grid of multiply-accumulate (MAC) units where data flows between neighboring cells in a regular, wave-like pattern. Instead of fetching and decoding instructions for each operation, the systolic array receives a matrix at one edge and pulses it through the grid, with each cell performing one MAC and passing the result to its neighbor. This eliminates the instruction fetch and decode overhead entirely, and it avoids writing intermediate results to memory because each partial sum flows directly to the next computation.

The cost of this efficiency is reduced programmability. Models must be compiled through Accelerated Linear Algebra (XLA), which maps high-level operations onto the fixed dataflow. Workloads with irregular control flow or dynamic shapes may not map well to this architecture, and the compilation process itself can be time-consuming (minutes to hours for complex models), which slows the iteration cycle during model development.

The programming model distinction between GPUs and TPUs has practical implications for organizational decisions. Research teams that frequently modify model architectures (adding custom attention patterns, experimenting with new activation functions, prototyping novel training algorithms) generally prefer the GPU's CUDA ecosystem because new operations can be implemented as custom kernels without waiting for compiler support. Teams running established architectures at scale (standard Transformer training, large-scale fine-tuning) may prefer TPUs because the XLA compiler can optimize the entire computation graph, often achieving better hardware utilization than hand-written CUDA kernels for standard operations.

[2] **Systolic Array**: From Greek *systole* (contraction), the systolic-array model was introduced by H. T. Kung and Charles E. Leiserson's VLSI work and later framed by Kung as a class of architectures where data pulses through a grid of processing elements in a heart-like rhythm (Kung and Leiserson 1979; Kung 1982). The metaphor explains the design: each cell performs one MAC and passes the result to its neighbor, eliminating register-file round-trips. Google's TPU v1 deployed a 256×256 array (65,536 MACs), achieving 92 TOPS within 75 W by hardwiring this dataflow – the architectural bet that made data center-scale inference economically viable (Jouppi et al. 2017).

Example 2.1: The TPU origin story

Context: In 2013, Google engineers projected that if users spoke to their Android phones for just three minutes per day using voice search, the company would need to double its data center compute capacity to handle the inference load (Jouppi et al. 2017). The cost was prohibitive.

Insight: Google's TPU analysis framed production neural-network inference as a dense-linear-algebra workload large enough that voice-search growth alone could have doubled serving capacity needs. This finding led directly to the TPU v1, a purpose-built inference accelerator with a 256×256 systolic array that could deliver 92 TOPS (INT8) within a 75 W power envelope. The first TPUs were deployed in 2015 and served production inference workloads including RankBrain and portions of Google Neural Machine Translation (Jouppi et al. 2017); the AlphaGo match-play system is documented separately (Silver et al. 2016).

Systems lesson: Specialization became economically justified because the workload was stable, massive, and dominated by one operation class. The TPU story is a generality-tax case, not merely a faster-chip story.

Custom ASICs represent the extreme end of the spectrum, where the economics of silicon justify abandoning general-purpose programmability entirely. When an organization runs a single model architecture at enormous scale, designing a chip specifically for that workload becomes economically rational. By stripping away every feature not required by the target computation, custom ASICs

achieve the lowest energy per operation and the highest sustained utilization of any architecture on the spectrum.

Tesla's Dojo D1 chip, for example, is optimized for the spatial and temporal structure of video-based vision models. Its hundreds of tightly coupled processing nodes are designed around the dataflow needs of Tesla's vision pipeline, with on-chip SRAM sized to keep spatial tiles close to the compute units. This reduces the repeated round-trips to off-chip memory that a general-purpose GPU would require for the same computation.

AWS Trainium takes a different approach, targeting the broad category of Transformer training rather than a single model. Trainium systems combine compiler-managed memory scheduling with hardware-assisted collective communication, so common training patterns such as tensor-parallel synchronization and data-parallel reductions can be optimized in the accelerator fabric rather than handled entirely by host software.

The risk of custom silicon is equally clear: if the dominant model architecture shifts, as it did from convolutional neural networks (CNNs) to Transformers between 2017 and 2020, a custom ASIC designed for the old architecture becomes a stranded asset. The design cycle for a new ASIC is 2–3 years from conception to deployment, which means the architecture decision must anticipate workload trends several years into the future. This prediction challenge is nontrivial: in 2015, few would have predicted that attention-based Transformers would replace CNNs as the dominant architecture within five years, and organizations that committed to CNN-optimized ASICs during that period found their hardware stranded by the architectural shift.

Custom silicon is therefore a bet on workload stability, and the organizations that make this bet are typically those with enough scale to justify the $50–$200 million development cost and enough workload volume to amortize the per-chip NRE (nonrecurring engineering) cost across millions of chips. Google can justify the TPU's development cost because high-volume services such as Search, YouTube recommendations, and Gmail spam filtering can amortize a shared accelerator platform. A research lab running a few hundred GPUs cannot justify the same investment.

The bet pays off handsomely when workloads are stable: a purpose-built ASIC can deliver 5–10$\times$ the energy efficiency of a general-purpose GPU for its target operation. However, the consequences of a wrong bet are severe, as the chip's fixed dataflow cannot be reprogrammed to accommodate a fundamentally new computational pattern. Pushing the locality argument further leads to a deeper question of physical scale: whether a single die can be expanded to the size of the entire wafer.

2.2.1 Wafer-scale engines

Wafer-Scale Engines (WSE) represent the ultimate pursuit of data locality. While every other architecture on the spectrum relies on chiplets or discrete dies connected by relatively slow PCB-level or package-level interconnects, a wafer-scale engine (like the Cerebras WSE-3) is a single, continuous piece of silicon the size of a dinner plate. By avoiding the need to "dice" the wafer into individual chips, a WSE can maintain a single, massive on-chip interconnect across its entire surface.

The WSE-3 contains 900,000 AI-optimized cores and 44 GB of on-chip SRAM, all connected by a silicon fabric that delivers 21 PB/s of memory bandwidth. To put this in perspective, a single WSE-3 has compute comparable to a large multi-H100 cluster and on-chip memory bandwidth comparable to thousands of H100 HBM links, but because the entire system resides on a single piece of silicon, the communication latency between any two cores is measured in nanoseconds rather than microseconds.

As figure 2.2 shows, the challenge of wafer-scale integration is physical: manufacturing yield, power delivery, and thermal expansion. A single defect on a standard chip might render it useless, but on a wafer-scale engine, the software must be "defect-aware," routing around local manufacturing flaws in the silicon fabric. Delivering 23 kW of power to a single piece of silicon and cooling it requires specialized manifold-level liquid cooling that is closer to industrial plumbing than traditional computer engineering.

Wafer-scale engines sit at a unique point on the spectrum: they are highly specialized in their *physical* architecture but flexible in their *computational* model, as the underlying cores are often general-purpose enough to execute diverse ML kernels. They represent a "Scale Up" philosophy that attempts to eliminate the communication wall by making the cluster the chip.

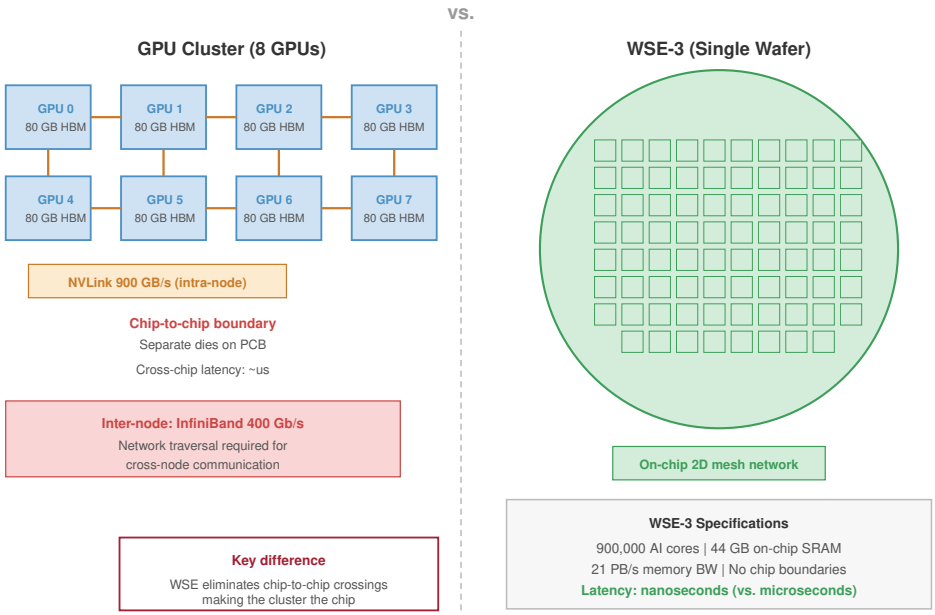


Figure 2.2: Wafer-Scale Engine (WSE) Architecture: Unlike traditional processors, a WSE uses the entire 300 mm silicon wafer as a single continuous compute fabric. This eliminates the 'Dicing' process and replaces PCB-level interconnects with a high-bandwidth on-chip mesh. The software must be defect-aware, routing data through a unified fabric that treats the entire wafer as a single logical processor with uniform memory access (UMA) characteristics across 900,000 cores.

The key wafer-scale trade-off is manufacturing complexity and defect-aware routing in exchange for eliminating the inter-chip communication bottleneck entirely, keeping all 900,000 cores within nanoseconds of each other on a single silicon fabric. Figure 2.3 places these architectures on a continuum, revealing the fundamental trade-off between programmability and efficiency that governs every accelerator design choice. Table 2.1 turns the same continuum into the architectural features an engineer must compare.

Table 2.1: The Accelerator Spectrum: As we move from left to right, we trade general-purpose programmability for compute density and power efficiency. Wafer-scale engines occupy a distinct niche, providing cluster-scale performance on a single piece of silicon.

Feature	**CPU**	**GPU** (H100)	**TPU** (v5p)	**Wafer-Scale**	**Custom ASIC**
Arithmetic Core	Scalar/Vector	Tensor Core	systolic array	RISC-style Core	Fixed Dataflow
Execution	Instruction	SIMT	Data-Driven	Dataflow	Hardwired
Memory Control	Cache	L1/L2 + HBM	Scratchpad	On-Chip SRAM	Explicit Mesh
Flexibility	Extreme	High	Moderate	High	Low
Efficiency	Low	High	Very High	High	Extreme

As table 2.1 summarizes, for our 175B model, the choice is not purely about peak FLOP/s. If we are a research lab experimenting with novel architectures weekly, the GPU's flexibility justifies its generality tax. If we are deploying a fixed Transformer at scale for years, the TPU's dataflow efficiency or a custom ASIC's power advantage may dominate total cost. The accelerator spectrum is ultimately an economic question of *how much flexibility can be surrendered, given the stability of the workload*.

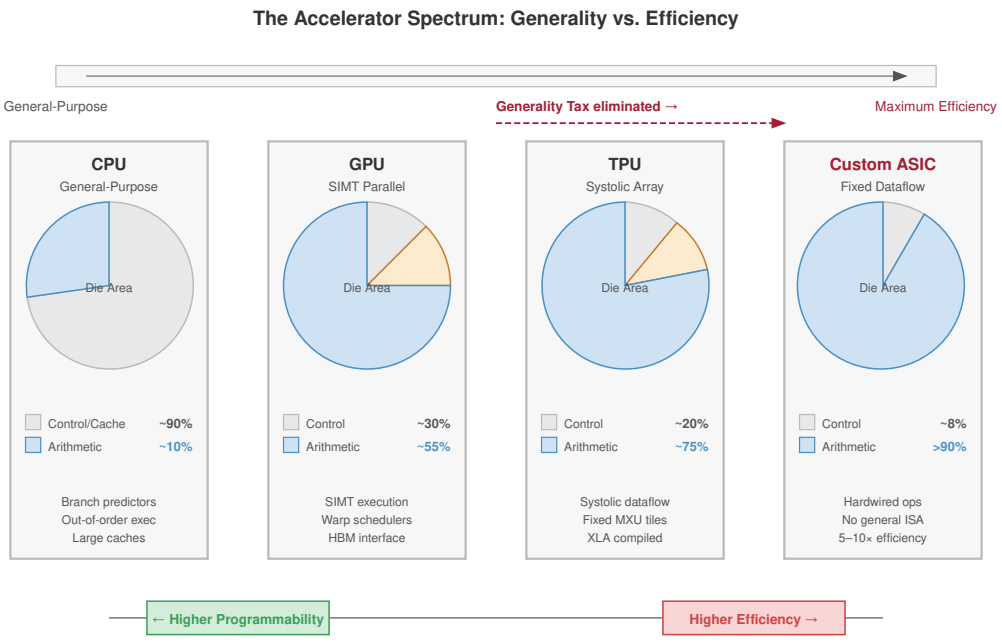


Figure 2.3: The Accelerator Spectrum: The landscape of ML hardware involves a fundamental trade-off between programmability and efficiency. General-purpose CPUs provide maximum flexibility but low compute density. Moving toward custom ASICs and systolic arrays increases throughput per watt by hardwiring common dataflows, at the cost of supporting a narrower range of model architectures.

2.2.2 Chiplet-based accelerators

A major accelerator design pattern is the **chiplet** architecture, exemplified by NVIDIA's Blackwell and AMD's Instinct MI300 series. Rather than fabricating a single monolithic die, chiplet-based designs partition the processor into multiple smaller dies connected by a high-bandwidth die-to-die interconnect on a common package substrate. This approach addresses two physical limitations that constrain monolithic designs.

First, the maximum die size is capped by the photolithographic reticle limit (the 858 mm² single-exposure ceiling examined in section 2.2.3), which limits the number of Tensor Cores, SMs, and HBM stacks a monolithic GPU can integrate. Chiplet designs bypass this limit by placing multiple dies on a single package, with the B200's dual-die design effectively creating a 1,600 mm² equivalent processor.

Second, manufacturing yield decreases exponentially with die area because a single defect anywhere on the die renders the entire chip unusable. Smaller chiplets have higher individual yield, and the package-level integration allows partial yields (a defective chiplet can be replaced with a good one). This yield advantage translates directly to lower manufacturing cost per unit of compute, which is particularly important as transistor densities continue to increase and process nodes become more expensive.

The trade-off is the die-to-die interconnect. Communication between chiplets within a package is faster than communication between packages (NVLink) but slower than communication within a single monolithic die (the on-die mesh network). Workloads that generate frequent, fine-grained communication between processing elements (such as operations that share data between nonadjacent SMs) may experience a latency penalty when those SMs reside on different chiplets. GPU vendors mitigate this by making the die-to-die link transparent to the programmer, so the software sees a single logical GPU regardless of the underlying chiplet topology.

2.2.3 Evolution of GPU architectures

The rapid evolution of NVIDIA's GPU architectures over the past decade illustrates how accelerator design has adapted to the changing demands of ML workloads. Each generation has introduced architectural innovations targeted at the specific bottlenecks revealed by production ML workloads, rather than simply increasing transistor counts.

The Volta architecture (2017) introduced the first-generation Tensor Core, recognizing that neural network training spends the majority of its time in matrix multiplications. By adding dedicated matrix-multiply-accumulate (MMA) hardware alongside the existing CUDA cores, Volta could accelerate the dominant workload pattern without sacrificing the general-purpose programmability that made GPUs attractive for research. The V100, Volta's flagship, delivered 125 TFLOP/s of FP16 Tensor Core throughput at 300 W.

The Ampere architecture (2020) expanded Tensor Core support to additional data types (TF32, BF16, INT8, INT4), reflecting the growing importance of mixed-precision training and quantized inference. The A100 also introduced Multi-Instance GPU (MIG), which partitions a single GPU into up to seven isolated instances, enabling efficient sharing of expensive hardware across multiple inference workloads. The most consequential change for infrastructure was Ampere's third-generation NVLink, which doubled bidirectional bandwidth to 600 GB/s per GPU from Volta's 300 GB/s, directly addressing the communication wall for tensor parallelism, which splits each layer's matrices across GPUs.

The Hopper architecture (2022) added the Transformer Engine, which dynamically selects between FP8 and FP16 precision on a per-layer basis, doubling the effective throughput for Transformer models without requiring manual precision tuning. Hopper also introduced NVLink 4.0 at 900 GB/s per GPU and the NVLink Switch, enabling NVLink connectivity beyond the 8-GPU boundary.

The H100's 1979 TFLOP/s of FP8 Tensor Core throughput at 700 W represents a 6.8× efficiency improvement over V100, achieved through a combination of process technology (TSMC 4N), architectural innovation (Transformer Engine), and precision engineering (FP8 support).

The Blackwell architecture (2024) continued this trajectory with the B200, which pairs two GPU dies in a single package via a high-bandwidth chip-to-chip link, effectively creating a "dual-die GPU" that delivers about 4,500 TFLOP/s of dense FP8 Tensor Core throughput at 1000 W. FP16/BF16 comparisons should use the lower precision-specific vendor figure rather than the FP8 peak.

The dual-die approach acknowledges that single-die GPU sizes are approaching the reticle limit of lithographic equipment, and further scaling requires chiplet-based designs. The reticle limit is a fundamental constraint of photolithography: the lens system in the EUV scanner can only project a pattern onto an area of approximately 26×33 mm (858 mm^2) in a single exposure. A die larger than this area would require multiple exposures with stitching, which is technically possible but dramatically increases cost and reduces yield.

Blackwell also introduced fifth-generation NVLink at 1800 GB/s per GPU, doubling the intra-node bandwidth again. The die-to-die link within the B200 package operates at 10 TB/s, fast enough that the two dies can appear as a single logical GPU to the software for many operations.

The progression reveals a consistent pattern: each generation addresses a bottleneck exposed by the previous generation. Volta added dedicated matrix hardware for the compute bottleneck. Ampere expanded mixed-precision support. Hopper targeted Transformer-specific precision with FP8. Blackwell pushed against the die-size limit with multi-die packaging.

At each step, the accelerator's design reflects an important workload of its era, illustrating *how* the physics of the workload drives the physics of the silicon. This co-evolution between workloads and hardware is not accidental: accelerator architects profile ML workloads to identify bottlenecks, and then design subsequent generations to address them. The result is a hardware evolution that is tightly coupled to model architecture evolution, which is *why* Transformer workloads have strongly shaped accelerator design since 2017.

The evolution also reveals a sobering procurement pattern for infrastructure planners: high-end accelerator generations can turn over faster than the facility that houses them. The V100 remained the flagship training GPU for approximately three years (2017–2020). The A100 held that position for roughly two years (2020–2022). Hopper and Blackwell continued that short-cadence pattern in the mid-2020s.

The planning implication is that a data center built for accelerators must anticipate refresh cycles before the first rack arrives. Hardware refresh planning is therefore an integral part of the initial procurement decision, not an afterthought. Table 2.2 compresses that cadence into the major NVIDIA GPU generations and the interconnect and power changes that arrived with them.

A subtlety that affects fleet consistency is the **silicon lottery** – the manufacturing reality where microscopic variance in 4nm lithography produces a distribution of chip quality across each wafer. NVIDIA manages this yield curve through aggressive binning: dies capable of sustaining high clock frequencies at strictly controlled voltages under the full TDP are designated as premium H100 SXM modules, while those with higher leakage currents or minor defects become H100 PCIe cards or are fused down to lower-tier products. Even within the top-tier SXM bin, the achievable boost clock varies based on silicon characteristics. In a synchronous training cluster, the collective communication primitives are blocked by the slowest participant. A single chip running 50 MHz below the fleet average can degrade the effective throughput of the entire cluster by 5–10 percent, which is *why* sophisticated fleet management systems track per-GPU performance metrics and quarantine underperforming silicon to inference pools where the impact of individual chip variance is less pronounced.

Table 2.2: NVIDIA GPU Architecture Evolution: Each generation introduces architectural features targeted at the dominant ML workload pattern of its era. Efficiency rises substantially but unevenly across generations, while NVLink bandwidth increases from 300 GB/s on V100 to 1,800 GB/s on B200, with a smaller 1.5× A100-to-H100 step between the larger doubling jumps.

Architecture	Year	Key Innovation	Peak Tensor	TDP	NVLink BW
Volta (V100)	2017	First Tensor Core	125 TFLOP/s	300 W	300 GB/s
Ampere (A100)	2020	Multi-precision, MIG	312 TFLOP/s	400 W	600 GB/s
Hopper (H100)	2022	Transformer Engine, FP8	1979 TFLOP/s	700 W	900 GB/s
Blackwell (B200)	2024	Dual-die, NVLink 5	4,500 TFLOP/s FP8	1000 W	1,800 GB/s

Table 2.2 compresses four hardware generations into a few columns. Figure 2.4 unpacks two of those columns, raw throughput and power efficiency, to reveal a divergence that shapes fleet-scale infrastructure decisions.

Figure 2.4 quantifies a critical asymmetry: while each GPU generation delivers dramatically more raw throughput, the power required to sustain that throughput grows nearly as fast. The advertised low-precision Tensor Core peak from P100 to B200 has grown by more than 200×, while TFLOP/s per watt has grown far less. The precision modes differ across generations, but that is itself part of the hardware trend: each generation gains compute density partly by introducing lower-precision formats. This divergence is the physical root of the power wall discussed in the next sections and explains *why* liquid cooling, megawatt-scale power delivery, and thermal management dominate dense accelerator facility design.

2.2.4 Evolution of TPU architectures

Google's TPU trajectory follows a different path, focusing on distributed efficiency and XLA compiler integration. While GPUs emphasize peak per-chip throughput and software flexibility (CUDA), the TPU is designed from the beginning as a "Pod-scale" resource. Every TPU chip is born as part of a larger cluster, with dedicated inter-chip interconnect (ICI) links that bypass the host CPU entirely.

The generation sequence shows how that pod-first design moved from inference to training. The TPU v1 (2015) was a dedicated inference chip with a 256×256 systolic array, delivering 92 TOPS (INT8) for the matrix operations that dominated Google's production inference workloads. TPU v2 (2017) and TPU v3 (2018) then shifted the architecture toward training by adding bfloat16 (BF16), avoiding the dynamic range problems of standard FP16, and scaling the pod concept to 1,024 chips

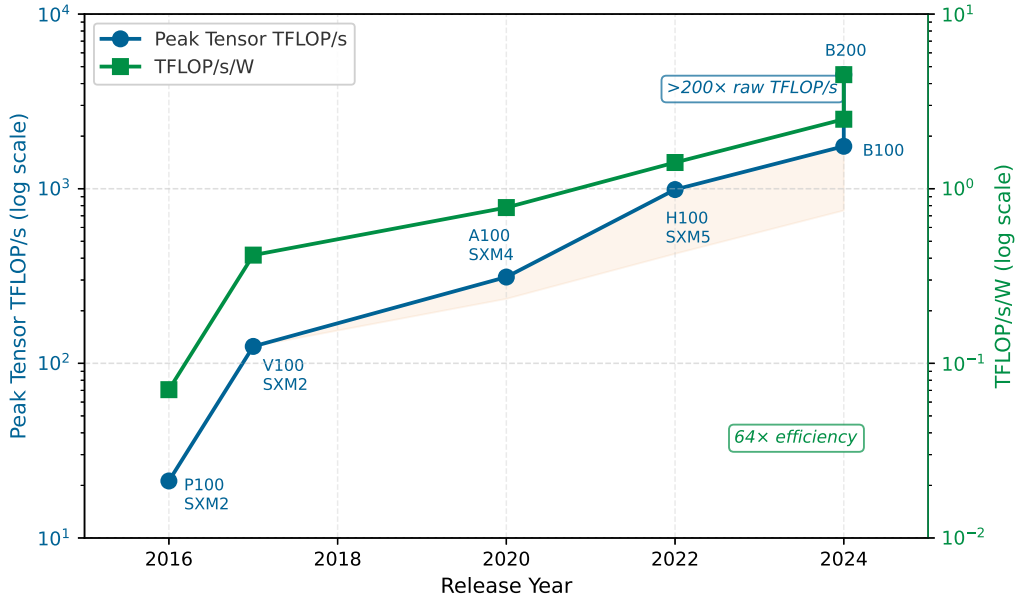


Figure 2.4: The Accelerator Efficiency Wall: Peak Tensor Core throughput (blue, left axis) and power efficiency (green, right axis) for six generations of NVIDIA data center GPUs, both on logarithmic scales. The series follows the advertised low-precision Tensor Core mode for each generation, so precision changes are part of the hardware trend rather than an apples-to-apples FP16 comparison. Raw throughput has grown over 200× from the P100 to the B200, but efficiency (TFLOP/s per watt) has grown far less. The shaded region highlights the widening gap that the power grid, cooling systems, and data center infrastructure must absorb.

and 100+ PFLOP/s of aggregate compute. TPU v4 (2021) extended the same logic into the network itself, adding an optical circuit switch (OCS) so the physical topology could be reconfigured for different workloads while raising per-chip BF16 throughput to 275 TFLOP/s.

The TPU v5p (2023) continued this pod-scale design for high-end training. It features 459 TFLOP/s of BF16 compute, 95 GB of HBM2e with 2,760 GB/s memory bandwidth, and 1,200 GB/s bidirectional ICI bandwidth per chip, optimized for the massive AllReduce operations required by billion-parameter models. Table 2.3 shows how that pod-first design evolved across TPU generations. Section D.2.1 develops the collective communication complexity model and the algorithm choices behind AllReduce, which explain why inter-chip bandwidth, not per-chip arithmetic, sets the ceiling on training throughput as model size grows.

Table 2.3: Google TPU Architecture Evolution: The TPU's development highlights a "Pod-first" design philosophy where inter-chip interconnect bandwidth and compiler-level optimization (XLA) are as important as peak per-chip arithmetic. Note: TPU v1 was INT8 only.

Generation	Year	Key Innovation	Peak BF16	HBM	ICI BW
TPU v1	2015	systolic array (Inference)	92 TOPS*	—	—
TPU v2	2017	High-Bandwidth Memory	45 TFLOP/s	16 GB	600 GB/s
TPU v3	2018	Liquid Cooling, Pod Scale	105 TFLOP/s	32 GB	650 GB/s
TPU v4	2021	Optical Circuit Switching	275 TFLOP/s	32 GB	1,200 GB/s
TPU v5p	2023	SparseCore, HBM2e	459 TFLOP/s	95 GB	1,200 GB/s

As table 2.3 illustrates, this comparison reveals a divergence in philosophy. GPU evolution is a race to pack more arithmetic and higher-precision Tensor Cores onto a single package, using chiplets (Blackwell) to overcome physical die-size limits. TPU evolution is a race to build more efficient warehouse-scale computers, where the individual chip's performance is secondary to the pod's collective bandwidth and reconfigurability (Barroso et al. 2019).

The accelerator's arithmetic engine performs nearly 2,000 TFLOP/s on the H100. Yet raw arithmetic throughput is meaningless if the data cannot reach the compute units fast enough. The fundamental bottleneck that limits every accelerator's effective performance is the memory wall.

2.3 The Memory Wall

With the accelerator's arithmetic engine selected, we confront a paradox: the faster we make our logic, the more it idles waiting for data. This diverging trajectory between processor throughput and data access speed is formally known as the **Memory Wall** (Wulf and McKee 1995). While transistor scaling has driven logic performance up by orders of magnitude, the physical interconnects that feed data to these cores have failed to keep pace. This bottleneck is existential for machine learning: unlike traditional software that benefits heavily from caching and data reuse, neural networks must stream billions of weights from memory for every inference pass, often making bandwidth – not compute – the governing constraint on performance. Section B.2 develops the diagnostic framework that classifies this memory-bound condition alongside the other bottleneck patterns a fleet encounters, so that a measured symptom maps to a named constraint rather than a guess.

The implications are concrete and perceptible in our running example. Our 175B model's weights occupy 350 GB in FP16. During autoregressive decoding, this entire 350 GB tensor must be streamed from off-chip memory into the processor's registers for every single token generated. A single H100 cannot hold that tensor, so this is a bandwidth-floor calculation after sharding or quantization has solved the capacity problem. At H100-class HBM bandwidths (~3.35 TB/s per accelerator), the data movement alone dictates a latency floor of over 100 ms per token if one accelerator-equivalent bandwidth path must stream the weights. The memory wall is not an abstract architectural concept: it is the physical reason a chatbot takes a perceptible pause between words. Three engineering responses address this constraint: high bandwidth memory (HBM) widens the data pipe, the roofline model rigorously diagnoses whether a workload is starving for data or for compute, and Tensor Cores maximize the arithmetic value of every byte fetched. As figure 2.5 shows, memory bandwidth is also a placement trade-off: moving the working set closer to compute raises bandwidth, but it usually reduces the amount of state that can stay there.

Figure 2.5 is not a ranking of accelerators; it is a map of where each design chooses to place bytes. Edge devices keep memory cheap and compact, but bandwidth stays low. HBM moves memory onto the accelerator package, buying enough capacity and enough bandwidth for active model state. Local-SRAM designs push the working set still closer to arithmetic, buying extraordinary bandwidth at the cost of much smaller resident capacity. That tier map explains why HBM is the next engineering response: it is not the fastest memory in the system, but it is the highest-bandwidth tier that can hold tens to hundreds of gigabytes next to the compute die.

2.3.1 HBM: Breaking the memory wall

An H100 can execute nearly 2,000 TFLOP/s of low-precision matrix arithmetic, yet during autoregressive decoding of our 175B model, its Tensor Cores would sit idle for over 99 percent of each cycle in a bandwidth-floor model. The bottleneck is not the speed of *multiplication* but the speed of *delivery*: once the model has been sharded or compressed so the weights fit, the serving path must still stream the active weight shards through the arithmetic units for every output token. No amount of additional Tensor Cores can help when the existing ones are already starved for data. The accelerator response to this fundamental limitation is High Bandwidth Memory[3].

The memory hierarchy within a single accelerator spans orders of magnitude in both capacity and bandwidth. At the top sits the **register file**[4] – approximately 20–30 MB distributed across all SMs – with effectively infinite bandwidth (hundreds of TB/s) but minuscule capacity. Below this lies the L1 cache and **shared memory** (SRAM)[5], offering roughly 256 KB per SM (approximately 33 MB total) with an aggregate bandwidth of ~19 TB/s. Further down sits the 50 MB L2 cache (~12 TB/s), and finally the 80 GB of HBM3 at 3.35 TB/s.

3 **HBM (High Bandwidth Memory)**: Standardized by JEDEC in 2013 as a joint development between AMD and SK Hynix, originally for graphics cards. ML accelerators adopted HBM because neural networks exhibit the same bandwidth-hungry, capacity-moderate access pattern as high-end rendering. Each HBM generation has roughly doubled bandwidth (128 GB/s in HBM1 to 1.2 TB/s per stack in HBM3e), yet the gap between memory bandwidth and arithmetic throughput continues to widen – making HBM a necessary but never sufficient response to the memory wall.

4 **Register File Bandwidth**: In an H100 GPU, the register file provides over 300 TB/s of aggregate bandwidth to the CUDA cores. This is ~100× higher than HBM3 bandwidth, explaining why tiling is mandatory: any operand not in a register during execution forces the arithmetic units to wait 100+ clock cycles for data, collapsing utilization.

5 **SRAM Energy (pJ/bit)**: Accessing a bit in SRAM (shared memory) consumes approximately 0.1–0.5 pJ, while fetching from HBM consumes 2–5 pJ. This 10× energy difference is the physical driver behind kernel fusion: every intermediate result kept in SRAM instead of written to HBM saves both time and significant power and thermal headroom.

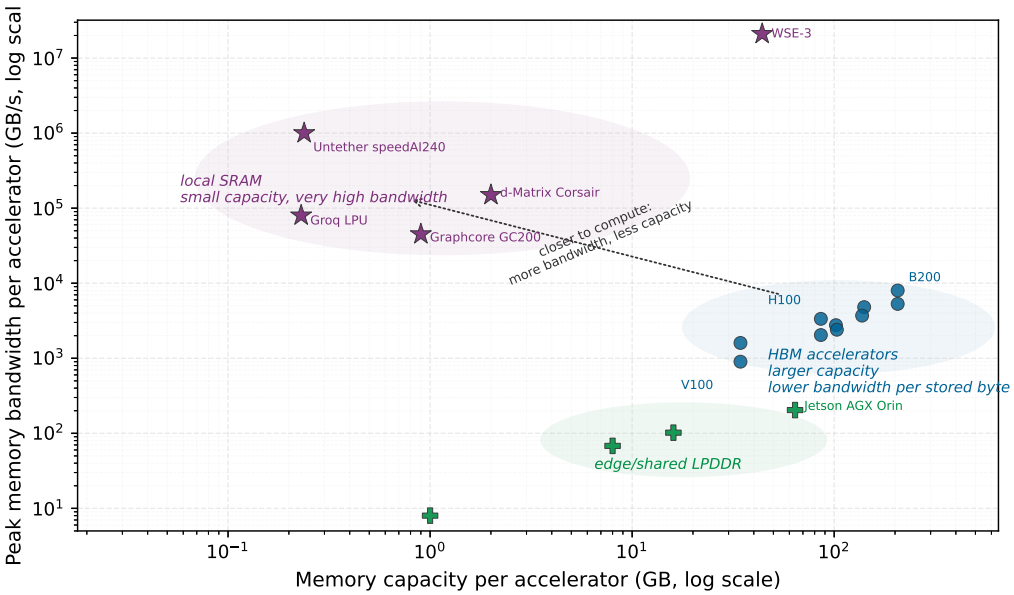


Figure 2.5: Accelerator Memory Tiers: Peak memory bandwidth and capacity separate local-SRAM designs, HBM accelerators, and edge/shared-LPDDR devices. The dotted arrow marks the design trade-off: moving data closer to compute raises bandwidth but reduces capacity.

Lighthouse 2.1: Archetype B (DLRM at Scale): The capacity wall

While Archetype A (GPT-4) is primarily throughput-bound (demanding more TFLOP/s), Archetype B (DLRM at Scale), the DLRM workload, is primarily capacity-bound. A 10 TB embedding table cannot fit into the 80 GB HBM of a single H100, nor can it fit into the aggregate HBM of a single 8-GPU node (640 GB). Capacity alone requires at least 16 such nodes before accounting for replication, hot-table headroom, and bandwidth balance; production-scale recommenders often grow to much larger multi-node shards, transforming a memory-access problem into a massive **All-to-All** network coordination problem.

The bandwidth gap between registers and HBM is approximately 100×. If an operand must be fetched from HBM for a single operation, the arithmetic unit spends about 99 percent of its time stalling. High Model FLOPs Utilization (MFU), the fraction of peak hardware FLOP/s spent on useful model computation, is only possible through aggressive **tiling**: breaking the massive weight matrices into small tiles that fit entirely within shared memory and registers, then performing as many multiply-accumulate operations on each tile as possible before evicting it. As figure 2.6 illustrates, this strategy loads data once into the SM's fast SRAM and reuses it across multiple Tensor Core operations, effectively multiplying the arithmetic value of every byte fetched from HBM.

Despite the 175B model's massive total memory footprint, the active working set at any given microsecond must be meticulously managed to reside in that top 30 MB of register space, or the chip's theoretical performance becomes a mirage. HBM supplies the bulk capacity, but tiling decides whether that capacity feeds the arithmetic units fast enough to matter.

Definition 2.1: High bandwidth memory (HBM)

High Bandwidth Memory (HBM) is the 3D-stacked DRAM architecture used by ML accelerators, in which multiple memory dies are vertically bonded and connected to the processor through thousands of Through-Silicon Vias (TSVs) on a shared silicon interposer, eliminating the

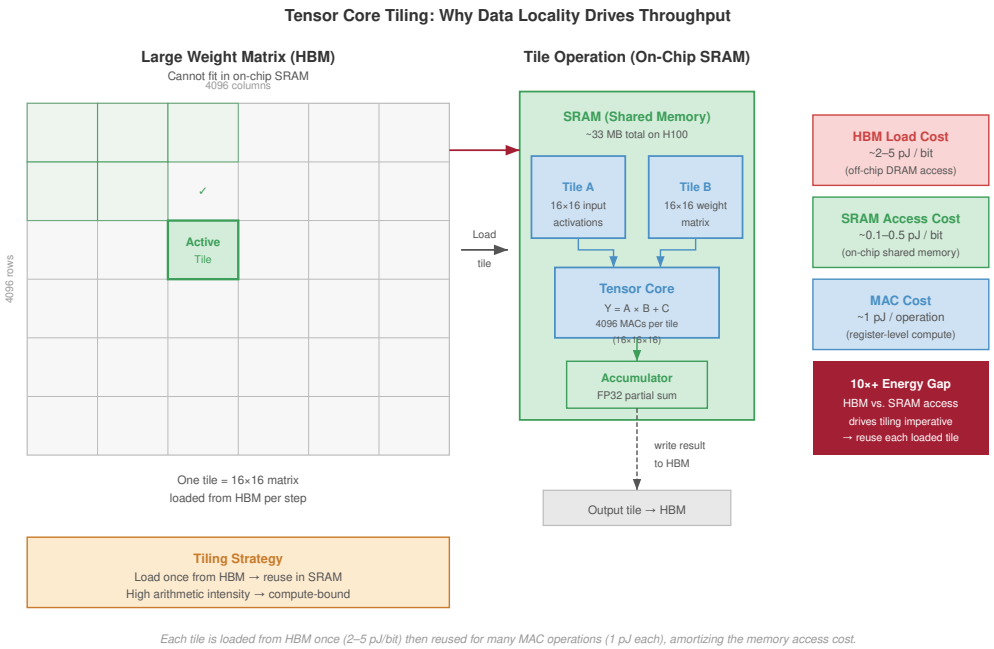


Figure 2.6: Tensor Core Tiling Strategy: To overcome the HBM bandwidth bottleneck, GPUs decompose large matrix operations into small tiles that fit within the Streaming Multiprocessor's (SM) fast SRAM. Data is loaded once into SRAM and reused across multiple Tensor Core operations, effectively multiplying the arithmetic value of every byte fetched from HBM.

centimeter-scale PCB traces of conventional DRAM and replacing them with micrometer-scale vertical paths.

1. **Significance**: HBM3 in the H100 delivers approximately 3.35 TB/s of memory bandwidth—roughly 16× DDR5's ~200 GB/s—at 2 pJ/bit vs. DDR5's ~20 pJ/bit. This bandwidth sets the BW ceiling in the iron law: the H100's FP16 ridge point is approximately $989\,\text{TFLOP/s}/3.35\,\text{TB/s} \approx 295$ FLOP/byte, so any operator with arithmetic intensity below 295 FLOP/byte (for example, attention at 5–20 FLOP/byte) remains memory-bound even with HBM.
2. **Distinction**: Unlike DDR memory, which connects through centimeters of PCB trace (introducing capacitance, attenuation, and high driving current), HBM uses 1,024-bit-wide TSV buses with path lengths measured in micrometers, a 1,000× reduction in signal distance that enables the wider bus without proportionally higher power.
3. **Common pitfall**: A frequent misconception is that HBM solves the memory wall. HBM moves the wall rather than eliminates it: Tensor Core throughput (R_{peak}) has scaled faster than HBM bandwidth across generations (from 900 GB/s with 125 TFLOP/s FP16 in V100 to 3.35 TB/s with 989 TFLOP/s FP16 in H100), so the ridge point continues rising and more operations remain memory-bound despite HBM improvements.

Traditional DDR memory connects to the processor through pins on the edge of a printed circuit board (PCB). Each DIMM communicates over a 64-bit bus, and even with multiple channels (8 channels is typical for a high-end server CPU), a modern server tops out at roughly 200 GB/s of aggregate memory bandwidth.

The physical distance from the DIMM slot to the processor die is measured in centimeters, and every centimeter of copper trace introduces capacitance, signal attenuation, and energy loss. At DDR5 data rates (4,800–6,400 MT/s per pin), the signal conditioning circuits must compensate for

significant channel impairment, consuming substantial power per bit transferred. Increasing the data rate on these long traces requires progressively more power for signal conditioning, creating a diminishing-returns curve that DDR5 is already approaching.

HBM solves this problem by changing the physical topology entirely, as figure 2.7 shows. Instead of routing signals *horizontally* across a PCB, HBM stacks multiple DRAM dies *vertically*, one on top of another, and connects them with **Through-Silicon Vias (TSVs)**[6]: microscopic copper pillars etched through the silicon substrate itself.

6 **TSV (Through-Silicon Via)**: A vertical copper pillar, 5–10 micrometers in diameter, etched through a silicon die to connect stacked layers. Originally developed for CMOS image sensors in smartphone cameras, TSVs enabled HBM by replacing centimeters of PCB trace with tens of micrometers of vertical silicon—a 1,000× reduction in signal path that drops energy per bit from ~20 pJ (DDR5) to ~2 pJ (HBM), making terabyte-per-second bandwidth economically feasible within an accelerator's power budget.

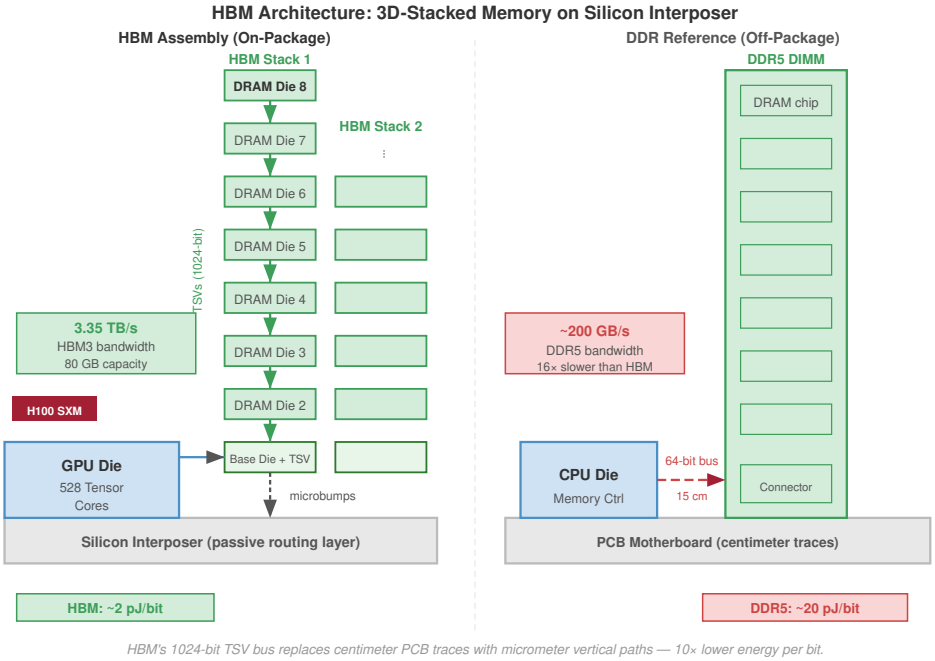


Figure 2.7: HBM 3D-Stacked Architecture: High Bandwidth Memory achieves its performance by vertically stacking DRAM dies and connecting them via Through-Silicon Vias (TSVs). The entire stack sits on a silicon interposer alongside the GPU die, enabling a 1024-bit-wide bus and reducing signal travel distance from centimeters to micrometers. This physical proximity is the key to breaking the memory wall.

The vertical stacking represents a fundamental change in memory architecture: rather than increasing bandwidth by pushing signals faster through long copper traces (the DDR approach, which has diminishing returns), HBM increases bandwidth by multiplying the number of parallel signal paths through extremely short vertical connections. A single HBM stack bonds 8–12 DRAM dies vertically, threading thousands of TSVs through each die to form a 1024-bit-wide interface per stack, compared to 64 bits for a DDR5 channel. This 16× wider interface, combined with higher per-pin signaling rates, is the source of HBM's bandwidth advantage. Because the vias travel *through* the silicon rather than *across* a PCB, the signal path shrinks from centimeters to tens of micrometers, roughly a 1000× reduction, and the whole stack sits on the same silicon interposer as the processor die so data travels from DRAM cell to arithmetic unit in nanoseconds rather than the tens of nanoseconds required by off-package DDR.

The shortened distance has three direct physical benefits:

- **Energy per bit**: The transfer cost drops by an order of magnitude, from approximately 20 pJ/bit for DDR5 to approximately 2 pJ/bit for HBM, because shorter traces have lower capacitance and require less driving current.
- **Latency**: Electrical signals propagate through silicon at roughly two thirds the speed of light, so the shorter path reduces propagation time.

- **Signaling rate**: Lower capacitance allows higher signaling frequencies without the sophisticated equalization circuits required by long PCB traces, enabling the high per-pin data rates that complement the wide bus width.

Together, these effects explain why HBM widens the interface and shortens the path instead of simply driving off-package pins faster.

Checkpoint 2.1: HBM and the memory wall

These questions check whether the physical benefits of 3D-stacked memory are clear:

- ☐ **HBM vs. DDR5**: Explain why HBM achieves higher bandwidth than DDR5 despite having a lower clock frequency.
- ☐ **Off-package signaling cost**: Explain why off-package DDR memory has higher energy per bit and how vertical stacking (TSVs) reduces that cost.
- ☐ **Capacity overflow**: Locate where the remaining 60 GB must reside when an H100 has 80 GB of HBM and a 70B parameter model requires 140 GB for FP16 weights.
- ☐ **Bandwidth not capacity**: Classify the claim as true or false: HBM's primary benefit is increasing the total memory capacity available to the GPU.

Table 2.4 summarizes the physical trade-off: HBM wins bandwidth by moving memory onto the package and widening the interface, not by making each off-package signal faster.

Table 2.4: HBM vs. Standard DRAM Comparison: HBM achieves its bandwidth advantage through three simultaneous innovations: 3D die stacking (more bits per package), TSV interconnects (shorter signal paths), and on-package placement (proximity to the processor).

Metric	**Host DRAM** (DDR5)	**Accelerator HBM** (HBM3e)	Scaling Factor
Mechanism	2D PCB Traces	3D Die Stacking	-
Placement	Socketed DIMMs	On-package (Substrate)	Physical Proximity
Bandwidth	~200 GB/s	~3.35 TB/s	~17× Faster
Interface Width	64-bit	1024-bit per stack	16× Wider
Energy	~20 pJ/bit	~2–5 pJ/bit	4–10× Efficiency

As table 2.4 shows, this bandwidth advantage comes at a price. HBM costs approximately $10–15 per GB, compared to roughly $3 per GB for DDR5 server memory. For an H100 with 80 GB of HBM3, the memory alone represents approximately $800–1,200 of the accelerator's manufacturing cost. For a B200 with 192 GB of HBM3e, the memory cost rises to $1,920–2,880 per accelerator, making HBM one of the most expensive components in the system.

The advanced packaging process, which requires precise alignment of thousands of TSVs across multiple die layers, has lower manufacturing yields and higher complexity. Each step in the stacking process (die thinning, alignment, bonding, and TSV etching) can introduce defects. The cumulative yield across 12 stacking steps means that the overall yield for a complete HBM3e stack is substantially lower than the yield for a single DRAM die. The silicon interposer itself must be large enough to accommodate both the processor die and multiple HBM stacks (often exceeding 1,000 mm^2 in total area), and any defect in a TSV can render an entire stack unusable.

The supply chain dynamics of HBM production further affect its cost and availability. The HBM supply chain is highly concentrated among a small number of manufacturers, and the production processes (die thinning, TSV etching, die-to-die bonding) require specialized capital equipment that is fundamentally different from standard DRAM manufacturing. Expanding HBM production capacity requires 12–18 months of equipment procurement and qualification, which means that production cannot rapidly scale in response to demand surges. When demand outpaces supply, as it

did following the explosion of interest in large language models in 2023, lead times stretch to 12–18 months and prices can double.

For infrastructure planners, this supply chain concentration means that HBM availability, not just its specifications, can determine the timeline for building a training cluster. Organizations planning large deployments must secure HBM allocations 12–18 months in advance, committing capital before the rest of the system is designed. This procurement lead time is longer than for any other component in the stack, making HBM the pacing element for fleet expansion.

For our 175B model, the HBM alone in a cluster of 1,000 accelerators represents roughly $0.8–1.2 million at 80 GB/accelerator (or $2–3 million at 192 GB/accelerator) under the $10–15/GB cost model above. This cost-capacity trade-off explains *why* accelerators typically offer 80–192 GB of HBM while the host server provides 512 GB to 2 TB of DDR: the *fast* memory holds the active computation (weights, activations, gradients that are accessed every cycle), and the *cheap* memory holds everything else (optimizer states, checkpoint buffers, data loading queues).

The boundary between what resides in HBM and what resides in DDR is a critical design parameter for training frameworks, and managing this boundary efficiently is one of the key challenges addressed by ZeRO (Zero Redundancy Optimizer) sharding and offloading strategies described in section 5.3.3. Sharding divides optimizer, gradient, and parameter state across workers; offloading places colder state in host DRAM or NVMe when HBM capacity is the binding limit. Getting this boundary wrong in either direction is costly: placing too much data in HBM wastes expensive capacity, while placing too much in DDR creates bandwidth stalls that idle the arithmetic units.

2.3.1.1 HBM generations and the scaling boundary

The evolution of HBM tracks the growth of model sizes with close correspondence. Each generation increases the number of stacked dies, the signaling rate per pin, and the total capacity per stack, driven by the relentless growth in model parameters. Table 2.5 shows the scaling boundary: HBM4 must widen the interface because pin-rate increases alone are no longer enough.

Table 2.5: Evolution of High Bandwidth Memory: Each generation increases aggregate accelerator memory bandwidth, tracking the growth of large model sizes. The jump to HBM4 doubles the interface width for the first time since HBM's introduction, signaling that pin-rate increases alone can no longer sustain the required bandwidth growth.

Metric	HBM2e (A100)	HBM3 (H100)	HBM3e (B200)	HBM4 (Future)
Peak Bandwidth	~2.0 TB/s	~3.3 TB/s	~8.0 TB/s	12–16 TB/s (projected)
Typical Capacity	40–80 GB	80–96 GB	192 GB	288 GB+
Interface Width	1024-bit	1024-bit	1024-bit	2048-bit
Stack Height	8 dies	8–12 dies	12 dies	16 dies

As table 2.5 shows, the transition from HBM3 to HBM3e is particularly significant for our running example. An A100 with 80 GB of HBM2e can hold only 23 percent of our 175B model's weights (at FP16). An H100 with 80 GB of HBM3 can hold the same fraction but deliver the data 65 percent faster. A B200 with 192 GB of HBM3e can hold 55 percent of the weights and deliver them at about 8 TB/s. Neither can hold the full model, which is precisely *why* we need multiple accelerators in a node, a topic we address in section 2.4.

However, the capacity story changes significantly when quantization is applied. The same 175B model quantized to INT8 requires only 175 GB, fitting in 3 H100 GPUs or a single B200. Quantized to INT4, it requires only 87.5 GB, fitting in two H100 GPUs. The capacity constraints that drive the need for multi-accelerator nodes for training (where FP16 or BF16 precision is typically required) are substantially relaxed for inference (where INT8 or INT4 quantization is often acceptable). This is another reason *why* training and inference infrastructure have different optimal configurations.

The bandwidth improvement matters independently of capacity. Each generation of HBM produces a nearly proportional reduction in per-token latency during autoregressive inference once the capacity problem has been handled by sharding or quantization.

A 70.6B FP16 model has 141 GB of weights, so a bandwidth-floor calculation gives roughly 141 GB/2.04 TB/s = 69.2 ms per token at A100-class bandwidth, 141 GB/3.35 TB/s = 42.1 ms at H100-class bandwidth, and 141 GB/8 TB/s = 17.7 ms at B200-class bandwidth. The A100 and H100 cases do not

imply that the full FP16 model fits on one device; they isolate the latency floor imposed by a single accelerator-equivalent HBM path. For interactive applications (chatbots, code assistants, real-time translation), where users perceive delays above 50 ms as "slow," these bandwidth improvements translate directly into better user experience and into the ability to serve larger models within latency budgets.

The projected jump to HBM4 doubles the interface width from 1024 bits to 2048 bits for the first time since HBM's introduction. This signals that per-pin signaling rate increases alone cannot sustain bandwidth growth indefinitely, and the bus must widen. The doubling of interface width requires a correspondingly larger interposer area for routing, which is one reason accelerator packages have tended to grow as bandwidth targets rise.

HBM4-class packages projected in the 12–16 TB/s aggregate range illustrate the planning pressure created by trillion-parameter-scale models. A 1T-parameter model requires roughly 2 TB of weight storage in FP16. At B200-class HBM3e bandwidths (about 8 TB/s), serving such a model at batch size 1 would produce tokens at 2000 GB/8 TB/s = 250 ms per token, far too slow for interactive applications. Even at 16 TB/s, the bandwidth floor remains roughly 125 ms per token before batching, quantization, or tensor-parallel sharding. This calculation suggests that trillion-parameter models require aggressive quantization and multi-accelerator tensor parallelism even for single-request inference. The co-evolution of model scale and memory technology continues to drive infrastructure requirements.

An important architectural consideration is the number of HBM stacks per accelerator and how they connect to the processor. The H100 has 5 HBM3 stacks, each providing approximately 670 GB/s, for a total of 3.35 TB/s aggregate bandwidth. A B200-class Blackwell GPU package uses 8 HBM3e stacks for an aggregate of about 8 TB/s. The number of stacks is constrained by the interposer area available for HBM placement: each HBM stack occupies approximately 100 mm^2 of interposer area, and the total interposer must accommodate both the processor die(s) and all HBM stacks. Larger interposers allow more HBM stacks (and therefore higher bandwidth and capacity) but are more expensive and harder to manufacture.

The interposer area constraint creates a concrete design tension. Making the processor die larger (more Tensor Cores, more SMs) leaves less interposer area for HBM stacks, potentially reducing bandwidth. Making the processor die smaller (fewer Tensor Cores) frees interposer area for more HBM but reduces peak compute throughput. The optimal balance depends on the target workload's position on the Roofline plot: compute-bound workloads benefit from a larger processor die (more Tensor Cores), while bandwidth-bound workloads benefit from more HBM stacks. Section C.3.1 derives the ridge point that separates these two regimes, expressing it through equation C.2 and equation C.4 so that a workload's arithmetic intensity predicts which side of the balance it falls on before any silicon is committed.

2.3.1.2 Memory bandwidth and token latency

The distinction between memory *capacity* (how many gigabytes the HBM can store) and memory *bandwidth* (how many terabytes per second it can deliver) is one of the most practically important concepts in ML infrastructure. Capacity determines whether a model's weights *fit*. Bandwidth determines how fast the model can *run*. For autoregressive text generation, where each token requires a full pass through the model's weights, bandwidth is almost always the binding constraint. Section D.1 formalizes the same latency-versus-bandwidth decomposition for data movement across the network, giving the reader a single algebraic model that applies equally to the memory system here and to inter-node transfers later.

> Napkin Math 2.1: The physics of token latency
>
> **Problem**: Generating one token from Llama-3 70B (141.2 GB of FP16 weights) requires both arithmetic and a full weight load from HBM. Assuming capacity has been solved (by sharding, a higher-capacity device, or quantization), which dominates the per-token time on an H100: compute or memory transfer?

1. **Compute Time**: Each token requires $2 \times 70 \times 10^9 = 1.4 \times 10^{11}$ floating-point operations. At 989 TFLOP/s of FP16/BF16 Tensor Core peak throughput, the arithmetic takes $T_{\text{compute}} \approx \mathbf{0.14}$ ms.
2. **Memory Time**:

Loading 141.2 GB of weights from HBM at 3.35 TB/s takes $T_{\text{mem}} \approx \mathbf{42.1}$ ms.

Systems insight: The processor spends 99.7 percent of its time waiting for data from memory. The arithmetic units are idle for almost the entire token generation. Even a hypothetical processor with infinite compute throughput would generate tokens only negligibly faster, because the memory transfer time dominates completely. This is *why* HBM bandwidth improvements deliver nearly linear speedups for inference workloads.

That bandwidth floor also gives us a clean way to compare adjacent hardware generations: if compute stays fixed while HBM capacity and bandwidth improve, decode latency should move with the memory system rather than the arithmetic units.

Napkin Math 2.2: The memory wall: H100 vs. H200

To understand *why* the "memory wall" is the primary constraint for modern large language models (LLMs), we compare the NVIDIA H100 against its successor, the H200. While both chips share the same compute cores (same peak TFLOP/s), the H200 provides 1.4× higher memory bandwidth and 1.6× more HBM capacity.

This LEGO comparison uses a Llama 3 model with 70.6B parameters, sequence length 2,048, and batch size 1; the reported speedup is the serving solver's inter-token latency estimate, not a raw bandwidth ratio alone. A napkin estimate brackets that solver number: decode spends 99.7 percent of its per-token time on the weight load, so the expected gain is the HBM bandwidth ratio, 4.8 TB/s ÷ 3.35 TB/s ≈ 1.4×.

Systems insight: For LLM decoding, the H200 is 1.4× faster than the H100 despite having identical compute power; the solver's estimate lands almost exactly on the bandwidth ratio because decode is bandwidth-bound. This proves that for large-scale autoregressive models, the "Wall" is the memory interface, not the arithmetic logic.

The napkin math reveals a profound asymmetry at the heart of modern ML infrastructure. The accelerator vendors invest billions of dollars in designing faster arithmetic units (more Tensor Cores, higher clock speeds, wider datapaths), yet for single-request inference, the arithmetic completes in a fraction of a millisecond while the memory transfer takes tens of milliseconds. The arithmetic units are about 300× faster than the memory system for this workload, meaning that over 99 percent of the silicon dedicated to computation is idle during inference. That compute-memory asymmetry is the single most important physical fact about ML inference, and it shapes every architectural and economic decision about serving infrastructure.

The fraction of token time spent waiting on memory, 99.7 percent in this example, is called the **memory-boundedness** of the workload. A workload that is 99 percent memory-bound will see almost no benefit from a faster processor (more TFLOP/s) but will see nearly linear speedup from faster memory (more TB/s bandwidth).

Memory-boundedness is a quantitative expression of the memory wall: the gap between processing speed and memory speed has widened for decades and is particularly acute for ML inference workloads. The memory wall was first identified by Wulf and McKee in 1995, who observed that processor speed was improving at 60 percent per year while DRAM speed improved at only 7 percent per year. The growing disparity meant that processors would increasingly spend their time waiting for data rather than computing on it. Thirty years later, their prediction has proven accurate, and the ML inference workload is the most extreme manifestation of the memory wall in modern computing.

The practical implication is that accelerator selection for inference workloads should prioritize bandwidth-per-dollar over FLOP/s-per-dollar. An older-generation GPU with high memory band-

width but moderate compute throughput may deliver better inference performance per dollar than a latest-generation GPU with extreme compute but insufficient bandwidth improvement.

Figure 2.8 makes this divergence visible across four GPU generations. While peak Tensor throughput has grown 36× from V100 to B200 when following each generation's advertised low-precision mode, memory bandwidth has grown only 9× over the same period. *The widening gap between compute growth and bandwidth growth is the memory wall*: each generation of accelerator becomes more powerful in arithmetic but proportionally more starved for data.

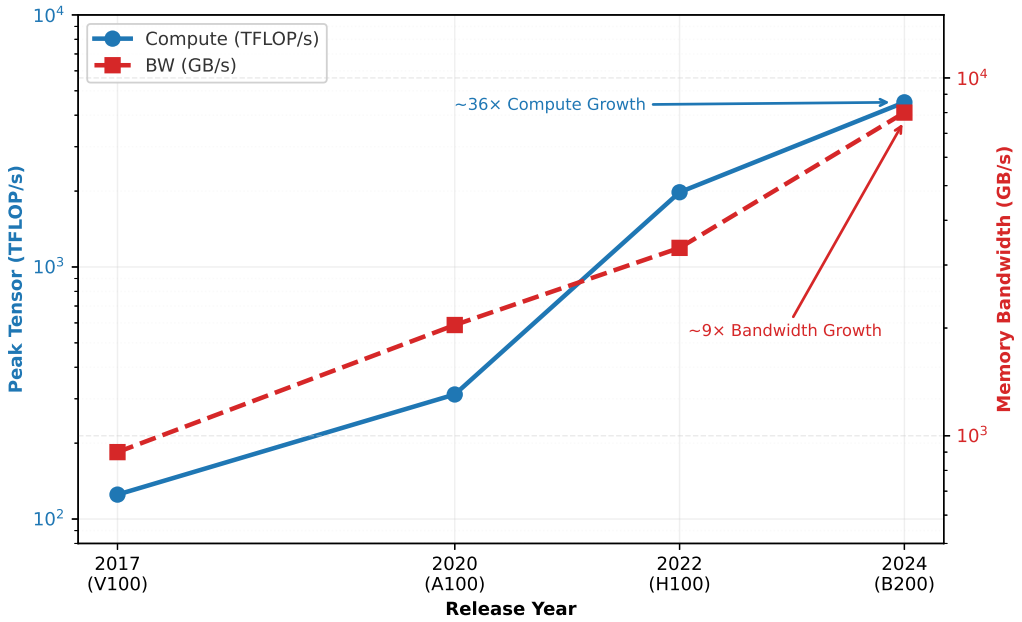


Figure 2.8: The Compute-Memory Divergence: GPU peak Tensor throughput has grown approximately 36× from V100 to B200 when following each generation's advertised low-precision Tensor Core mode, while HBM bandwidth has grown only 9× over the same period. The widening gap between these curves is the memory wall: each generation of accelerator becomes proportionally more starved for data.

For training workloads, where large batch sizes increase arithmetic intensity, the calculus shifts toward compute: peak TFLOP/s per dollar becomes the relevant metric because the weight data loaded from HBM is amortized across many tokens in the batch. The Roofline Model, which we examine next, provides the formal framework for making this trade-off precise and for determining which metric matters for any given workload.

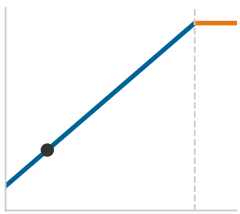
2.3.2 Roofline model

The token latency calculation above demonstrated that a 70B model on an H100 is overwhelmingly memory-bound. The question is how to determine this systematically for any workload on any hardware. The **Roofline Model**[7], introduced by Williams, Waterman, and Patterson (Williams et al. 2009), provides a visual and analytical framework that answers this question with a single number: the arithmetic intensity of the workload.

For an H100 (989 TFLOP/s FP16, 3.35 TB/s HBM), the ridge point is ~295 FLOP/byte; most LLM inference operators fall well below this threshold, which is *why* the Roofline remains the first diagnostic tool for identifying whether more compute or more bandwidth will improve performance. The quantity that places a workload on this plot is its **arithmetic intensity** (I), defined as $I =$ FLOP/byte: the ratio of computation to memory traffic that determines whether a workload is bandwidth-bound or compute-bound. The same quantity carries forward to fleet-scale performance analysis at section 9.1.5.

The Roofline Model expresses the maximum achievable performance of a workload as the lesser of two ceilings:

7 **Roofline Model**: Introduced by Williams, Waterman, and Patterson at UC Berkeley in 2009, the model plots attainable FLOP/s against arithmetic intensity on a log-log scale, producing two intersecting lines whose crossover point – the *ridge point* – separates memory-bound from compute-bound regimes.



Decode sits deep in the memory-bound regime, far below the ridge.

$$\text{Achievable FLOP/s} = \min\left(R_{\text{peak}},\ \text{BW} \times I\right) \tag{2.1}$$

Equation 2.1 has a direct physical interpretation. If the workload's arithmetic intensity is low (it needs many bytes per operation), then performance is limited by how fast memory can deliver those bytes. The achievable FLOP/s grows linearly with I, tracing a sloped line on a log-log plot. If the arithmetic intensity is high (each byte fuels many operations), then performance plateaus at the hardware's peak compute rate, regardless of further increases in I. This plateau is the flat "roof" of the model.

Definition 2.2: Ridge point

Ridge Point is the ML accelerator arithmetic intensity where the memory bandwidth ceiling meets the compute ceiling ($I_{\text{ridge}} = \frac{R_{\text{peak}}}{\text{BW}}$).

1. **Significance**: It defines the Hardware Efficiency Threshold. Workloads with an intensity below the ridge point are bandwidth-bound (BW), while those above are compute-bound (R_{peak}).
2. **Distinction**: Unlike Peak FLOP/s (which only describes the horizontal ceiling), the ridge point describes the Balance of the architecture. A rising ridge point over hardware generations indicates that compute is growing faster than bandwidth, making utilization harder.
3. **Common pitfall**: A frequent misconception is that all GPUs have the same ridge point. In reality, it varies by Precision: because R_{peak} is higher for INT8 than FP32 while BW is constant, the ridge point for INT8 is much higher, requiring more data reuse to saturate the hardware.

Specific ML workloads occupy different regions of this plot depending on the operation and the batch size:

- **LLM decode (batch size 1)**: Each token requires loading the full weight tensor (~2 bytes per parameter) for just 2 FLOPs per parameter, yielding $I \approx 1$ FLOP/byte. This is deep in the memory-bound region, well below the ridge point of 295.2 FLOP/byte. Our token latency calculation confirmed this: the arithmetic finished in microseconds while the memory transfer took milliseconds.
- **LLM prefill (large context)**: Processing a long input sequence in parallel increases the FLOPs (matrix-matrix multiply instead of matrix-vector) without proportionally increasing memory traffic, pushing I to 100–500 FLOP/byte. This range straddles the H100 ridge point: the lower end remains memory-bound or near the ridge, while the upper end crosses into the compute-bound region.
- **CNN training (large batch)**: Convolution with large spatial dimensions and batch sizes achieves I of 50–200 FLOP/byte, placing it near or above the ridge point for most accelerators.
- **Attention (long sequences)**: The self-attention mechanism scales quadratically with sequence length in FLOPs but linearly in memory traffic for the KV cache, making its arithmetic intensity sequence-length-dependent. Short sequences are memory-bound; long sequences are compute-bound.

The point is to classify each phase and batch shape on the roofline, not to assign one permanent region to an entire model family.

Reading a Roofline plot requires understanding what each axis represents. The horizontal axis is arithmetic intensity (FLOP/byte), plotted on a log scale. The vertical axis is achievable performance (FLOP/s), also on a log scale. Two lines define the "roofline" shape: a diagonal line with slope 1 (on the log-log plot) representing the memory bandwidth limit, and a horizontal line representing the peak compute limit. These two lines meet at the ridge point.

Any workload can be plotted as a single point on this chart by computing its arithmetic intensity and measuring its achieved performance. If the point lies on the diagonal line, the workload is

memory-bound and would benefit from faster memory. If it lies on the horizontal line, the workload is compute bound and would benefit from more arithmetic units.

If the point lies below either line, the workload is not fully using the available resource. This gap indicates an optimization opportunity in the software: kernel inefficiency, poor memory access patterns, or excessive synchronization. Closing this gap is the province of kernel engineering and communication optimization, topics examined in Chapter 9.

The Roofline's diagnostic power extends beyond individual kernels to entire training runs. For our 175B model, the computation graph contains thousands of distinct operations with different arithmetic intensities. The dense Feed-Forward Network (FFN) layers are dominated by large GEMMs with high arithmetic intensity, placing them firmly in the compute-bound regime where Tensor Core utilization is the bottleneck. Conversely, operations like layer normalization and element-wise activations possess low arithmetic intensity, sitting deep in the memory-bound region where the compute units idle while waiting for data. The self-attention mechanism fluctuates between regimes depending on sequence length: while the quadratic complexity of attention scores suggests a compute bound, the loading of Key and Value matrices creates memory pressure at shorter sequences. This diagnostic distinction dictates the optimization strategy: memory-bound layers benefit from kernel fusion (reducing HBM round-trips), while compute-bound layers benefit from precision reduction (moving from FP16 to FP8, which effectively raises the hardware's compute ceiling).

Figure 2.9 makes these relationships visible for the H100. Notice how LLM decode at batch size 1 sits deep in the memory-bound region, achieving less than 1 percent of peak compute, while LLM training at large batch sizes crosses the ridge point into the compute-bound regime. Under the batch-size 2048 simplification used in the running example, the 2,048× gap between these two workloads' arithmetic intensities (~1 FLOP/byte for decode vs. ~2,048 FLOP/byte for training) explains why the same hardware that delivers excellent training throughput can appear woefully underutilized during inference.

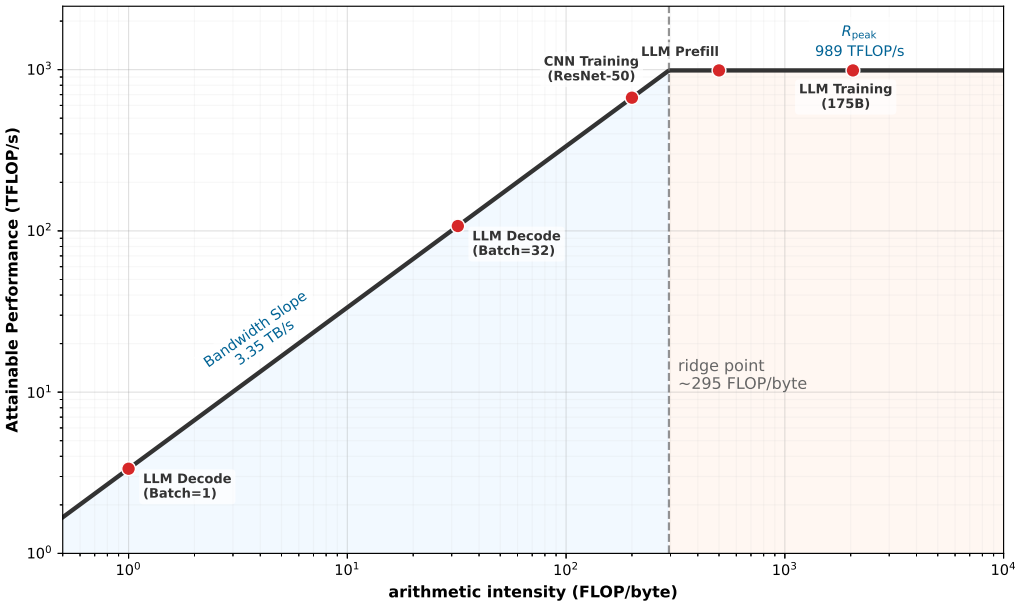


Figure 2.9: The Roofline Landscape: NVIDIA H100 (FP16): H100 roofline showing the memory-bandwidth and peak-compute ceilings, with representative ML workloads spanning memory-bound and compute-bound regions.

The roofline in figure 2.9 is defined by two ceilings: memory bandwidth (diagonal) and peak compute (horizontal), meeting at the FP16 ridge point (~295.2 FLOP/byte for 989 TFLOP/s over 3.35 TB/s). ML workloads span the full range: LLM decode at batch size 1 is deeply memory bound,

while LLM training at large batch sizes is compute bound. The position of each workload determines whether faster memory or faster compute would improve performance.

The Roofline Model also reveals a subtle but important insight about the interaction between batch size and hardware utilization. Increasing the batch size for a given model raises the arithmetic intensity because the weight matrix is loaded once but multiplied against a larger activation matrix (more FLOPs for the same bytes transferred). This shifts the workload point rightward on the plot, potentially crossing the ridge point from memory-bound to compute-bound territory. The arithmetic intensity grows linearly with batch size (doubling the batch size doubles the FLOPs while keeping the weight loading unchanged), creating a simple and predictable relationship between batch size and hardware utilization.

For inference serving, batching multiple requests together dramatically increases hardware utilization and throughput. A model that achieves 1 percent of peak FLOP/s at batch size 1 might achieve 50 percent of peak FLOP/s at batch size 64, simply because the weight data loaded from HBM is reused across 64 independent requests rather than one. This 50× improvement in hardware utilization comes at a cost: the 64 requests must wait until a full batch is assembled before processing begins, introducing queuing latency. Serving systems must carefully balance this trade-off between throughput (larger batches, higher utilization) and latency (smaller batches, faster response per request).

For training, the batch size is a hyperparameter that affects both statistical convergence and hardware efficiency, creating a trade-off that practitioners must navigate carefully. Larger batch sizes improve hardware utilization (pushing the workload into the compute-bound regime) but may require learning rate adjustments and warmup schedules to maintain training quality. The optimal batch size depends on both the model architecture and the hardware's Roofline characteristics, creating a cross-disciplinary optimization problem that spans ML theory and systems engineering.

The practical value of the Roofline Model is that it tells us which resource to optimize. If a workload is memory-bound, buying a faster accelerator (more TFLOP/s) yields no benefit; only higher memory bandwidth will help. Conversely, if a workload is compute bound, upgrading HBM generations is wasted money. This diagnostic power is the reason that experienced infrastructure engineers always begin a hardware selection process by computing the arithmetic intensity of their target workload and plotting it against the candidate hardware's Roofline: the plot immediately reveals which hardware characteristic matters and which is irrelevant.

For our 175B model, training with large batch sizes is compute bound (optimize for TFLOP/s), while serving individual requests is memory bound (optimize for bandwidth). This duality explains why some organizations use different hardware generations for training and inference. An A100, with its lower cost and adequate memory bandwidth, may be more cost-effective for inference than an H100, despite the H100's higher peak FLOP/s.

The Roofline Model also provides a quantitative framework for evaluating the return on investment of different optimizations. If a workload is 10× below the compute ceiling but already touching the bandwidth ceiling, spending engineering effort on kernel optimization (moving toward the compute ceiling) yields no benefit. The effort should instead be directed toward reducing memory traffic (shifting the workload rightward on the plot) through techniques like batching, kernel fusion, or quantization.

The Roofline's diagnostic power makes it one of the most practically useful analytical tools in the infrastructure engineer's toolkit. Before committing to any hardware purchase or optimization effort, plotting the workload on the Roofline reveals immediately whether the investment will produce returns. Teams that skip this analysis risk spending months optimizing the wrong resource, an expensive mistake when GPU-hours cost thousands of dollars per day.

The same arithmetic-intensity calculation separates the running model's inference and training regimes.



Example 2.2: Roofline analysis: Training vs. inference

Consider our 175B model on an H100 with 989 TFLOP/s peak compute and 3.35 TB/s memory bandwidth. The ridge point is 295.2 FLOP/byte.

Inference (decode, batch size 1): Each token loads 350 GB of FP16 weights and performs $2 \times 175 \times 10^9$ FLOPs. Arithmetic intensity: $I = \frac{2 \times 175 \times 10^9}{350 \times 10^9} = 1$ FLOP/byte. Since 1 FLOP/byte $\ll$ 295.2 FLOP/byte (the ridge point), the workload is deeply memory bound. The achievable throughput is $3.35 \times 1 = 3.35$ TFLOP/s, which is about 0.34 percent of the H100's peak 989 TFLOP/s. No amount of additional compute will help; only more memory bandwidth improves throughput.

Training forward pass (batch size 2048): The same weight tensor is multiplied by a batch of 2048 activation vectors simultaneously, turning matrix-vector operations into matrix-matrix operations. The FLOPs increase by 2048$\times$ while the weight loading remains constant: $I = 2048 \times 1 = 2{,}048$ FLOP/byte. Since 2,048 FLOP/byte $\gg$ 295.2 FLOP/byte, the workload is compute bound. The achievable throughput is now limited by the peak compute ceiling of 989 TFLOP/s, and the memory bandwidth is irrelevant. Adding more TFLOP/s (through a newer GPU generation) directly improves performance.

Systems insight: The contrast explains why organizations sometimes use different hardware for training and inference: *training* benefits from peak FLOP/s, while *inference* benefits from memory bandwidth per dollar.

That same diagnostic becomes most valuable when a proposed upgrade changes the wrong ceiling.

Checkpoint 2.2: Roofline diagnosis

A team is serving a 13B-parameter model at batch size 1 on an H100 and observing 25 ms per token. The profile shows low tensor-core utilization during decode.

- ☐ **Arithmetic intensity**: Estimate the arithmetic intensity of the workload.
- ☐ **Peak-throughput upgrade**: Decide whether a higher peak-TFLOP/s accelerator changes the binding term.
- ☐ **Binding constraint**: Identify the measurement or hardware capability that should improve first.

2.3.3 Tensor Cores and matrix units

The Roofline Model tells us whether a workload can reach peak compute. The next question is what determines that peak in the first place. The answer lies in the specialized arithmetic units that occupy the majority of a modern accelerator's die area: **Tensor Cores**[8] on NVIDIA GPUs and **Matrix Multiply Units (MXUs)** on Google TPUs.

A standard CPU floating-point unit performs one multiply-accumulate per cycle per lane. Even with wide SIMD units (AVX-512 provides 16 FP32 lanes), a CPU core performs at most 16 MACs per cycle. With 32 cores, a high-end server CPU achieves approximately 512 MACs per cycle, a respectable number for general-purpose computation but wholly inadequate for the demands of matrix multiplication at neural network scale.

A Tensor Core, by contrast, executes matrix-multiply-accumulate (MMA) instructions of the form $\mathbf{Y} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ on small tile matrices in specialized hardware. On the H100, these tile operations are distributed across the Tensor Core array in each SM, producing hundreds of accumulated results per instruction while keeping operands in registers and shared memory. This concentrates the dominant neural-network operation in hardware that occupies a small fraction of the chip area.

With 528 Tensor Cores across the chip (4 per SM, 132 SMs), the H100 reaches the 989 TFLOP/s FP16/BF16 peak used in the roofline analysis. The important systems point is not the exact per-cycle instruction accounting, but the architectural specialization: the die dedicates a large fraction of its arithmetic area to dense tiled matrix operations rather than general-purpose scalar execution.

Google's MXUs take the same concept further by organizing the multipliers into a systolic array. In a systolic MXU, one matrix is loaded into the array's weight registers while the other matrix streams through the array one row at a time. Each cell multiplies the incoming activation by its stored

[8] **Tensor Core**: Introduced with NVIDIA Volta (2017) as $4 \times 4 \times 4$ FP16 fused matrix-multiply-accumulate units. Each generation widened the tile: Turing (2018) added INT8, Ampere (2020) added TF32/BF16, and Hopper (2022) reached $16 \times 16 \times 16$ with FP8 support. This precision cascade tracks ML's own shift toward lower-precision training and inference—each new format unlocks a 2$\times$ throughput gain on the same silicon, making Tensor Core generations a proxy for how quickly the hardware-software co-design loop can widen the Roofline's compute ceiling.

weight, adds the result to the partial sum flowing from the cell above, and passes both the activation rightward and the partial sum downward. This pipelined flow means the array is performing useful computation on every cycle, with no idle cells once the pipeline is full. The TPU v5p contains two 128×128 MXUs per chip, providing 459 TFLOP/s of BF16 throughput. The systolic dataflow eliminates the register file accesses between operations that a Tensor Core still requires, achieving marginally higher energy efficiency at the cost of the flexibility to run nonmatrix workloads.

For our 175B model, the choice between Tensor Cores and MXUs manifests in the compiler stack. CUDA kernels can intermix Tensor Core operations with arbitrary thread-level code, enabling fused kernels that combine matrix multiplies with activation functions, dropout, and layer normalization in a single launch. This fusion is critical for performance because it eliminates the intermediate memory reads and writes that would otherwise occur between operations, keeping data in registers and shared memory where access is fastest.

XLA compilation for TPUs must decompose the computation into sequences of matrix operations that map onto the systolic dataflow, which can be more efficient for standard Transformer architectures but less flexible for custom operations. The XLA compiler performs whole-program optimization, analyzing the entire computation graph to find the optimal tiling, memory layout, and execution schedule for the systolic array. For standard Transformer layers, this whole-program optimization can achieve higher hardware utilization than hand-tuned CUDA kernels, because the compiler can reason about the entire computation rather than optimizing individual operations in isolation.

The hardware dictates the software abstraction, which in turn shapes what architectures are practical to experiment with. This coupling between hardware and software is a defining characteristic of ML infrastructure and explains why hardware selection has downstream effects on research velocity and model design flexibility.



Definition 2.3: Tensor core

Tensor Core is a specialized ML accelerator hardware unit that performs a fused matrix-multiply-accumulate (MMA) operation $\mathbf{Y} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ on small tiles (for example, $16\times8\times16$ in BF16) as a single hardware instruction, delivering dramatically higher throughput than general-purpose CUDA cores by trading programmability for fixed-function matrix arithmetic.

1. **Significance**: Tensor Cores provide the bulk of the H100's 989 TFLOP/s FP16/BF16 peak (and ~1,979 TFLOP/s at FP8)—roughly 14.8× the ~67 TFLOP/s delivered by CUDA (vector) cores alone. However, this peak is only achievable for operations that decompose into tiled matrix multiplications. Layer normalization and softmax, which involve reductions and element-wise operations rather than matrix multiplications, run at approximately 3–8 percent of peak throughput on the same hardware.
2. **Distinction**: Unlike general-purpose CUDA cores (which execute one floating-point operation per clock per lane in a SIMT pipeline), Tensor Cores execute 512 multiply-accumulate operations per clock per warp on matrix tiles—trading the arbitrary per-element programmability of CUDA cores for the specific throughput of matrix arithmetic.
3. **Common pitfall**: A frequent misconception is that all GPU operations benefit from Tensor Cores. Only operations that map to the $\mathbf{Y} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ tile computation (dense linear layers, attention score computation, convolutions) engage Tensor Cores; element-wise activations, layer norm, and embedding lookups fall back to CUDA cores and can run 10–30× slower per FLOP than Tensor Core operations.

The precision support of Tensor Cores has expanded with each generation, driven by the discovery that neural network training and inference can tolerate lower numerical precision than was previously assumed. The Volta generation supported only FP16 accumulation. Ampere added BF16, TF32, and INT8. Hopper added FP8 (both E4M3 and E5M2 formats) with dynamic per-tensor scaling through the Transformer Engine.

The Transformer Engine automatically monitors the magnitude distribution of each tensor and selects FP8 when precision is adequate or FP16 when higher precision is needed, doubling the

effective throughput for layers where FP8 suffices. This hardware-assisted precision management represents a convergence of arithmetic design and model-level insight: the hardware is no longer a passive executor of the programmer's precision choices but an active participant in the precision decision.

Understanding the distinction between the two FP8 formats clarifies why this matters for practitioners. The E4M3 format (4 exponent bits, 3 mantissa bits) can represent values up to approximately 4.48×10^2, with 3 bits of mantissa providing about 1 part in 8 relative precision. This range and precision are adequate for many neural network weights and activations after per-tensor scaling brings their observed range into the FP8 window.

The E5M2 format (5 exponent bits, 2 mantissa bits) provides a wider dynamic range, with maximum finite values around 5.73×10^4, at the cost of reduced precision (only 2 mantissa bits, or about 1 part in 4 relative precision). This wider range is important for gradients during the backward pass, which can span more orders of magnitude than forward-pass activations, particularly in the early and late layers of deep networks.

The Transformer Engine selects between these formats on a per-layer basis, and the per-tensor scaling factors are maintained in a small metadata buffer that adds negligible memory overhead. The scaling factors are updated every few iterations based on the observed tensor value distributions, ensuring that the limited precision of FP8 is focused on the range of values that actually appear in each tensor.

The practical implication for fleet design is that peak TFLOP/s specifications are precision-dependent. The H100 delivers 989 TFLOP/s in FP16 but twice that (1,979 TFLOP/s) in FP8. A fleet designed for FP8 training effectively has twice the compute density of the same fleet running FP16, with no additional hardware. This makes precision engineering a first-class optimization lever for infrastructure planners, not just a model accuracy concern.

To achieve this peak throughput in practice, the entire accelerator must be viewed as a rigid pipeline where data flows from HBM through a deepening hierarchy of caches before reaching the Tensor Cores. The fundamental constraint is **pipeline balance**: the rate at which data is staged into registers must match or exceed the rate at which the arithmetic units consume it. When the arithmetic intensity of an operation falls below the ridge point, the pipeline stalls, leaving teraflops of compute potential idle while waiting for data. This makes **kernel fusion** the single most critical software optimization for large-scale training. By fusing multiple operations (matrix multiplication, bias addition, activation function) into a single kernel, the system eliminates the round-trips to HBM that would occur if each operation were executed sequentially. Consider the attention mechanism: a naive, unfused implementation writes the $S \times S$ attention matrix to HBM only to read it back for the softmax operation, creating a round trip capped by the 3.35 TB/s memory bandwidth. A fused implementation like FlashAttention keeps these intermediate matrices entirely in on-chip SRAM, bypassing HBM and allowing the Tensor Cores to run near their theoretical peak. For our 175B model, where each training step involves thousands of matrix operations across 96 Transformer layers, the difference between fused and unfused kernels can be a 2–3$\times$ throughput improvement, separating a 2-week training run from a 6-week one.

2.3.3.1 Peak vs. sustained throughput

A critical distinction for infrastructure planning is the difference between *peak* throughput (the maximum the hardware can achieve on a synthetic benchmark) and *sustained* throughput (what the hardware actually delivers during real training runs). Peak TFLOP/s assumes that the Tensor Cores are fed with data on every cycle, which requires perfect scheduling, zero memory stalls, and no communication overhead. In practice, sustained throughput during Transformer training is typically 30–50 percent of peak for compute-bound operations and less than 5 percent of peak for memory-bound operations. The gap between peak and sustained throughput arises from several sources, each of which represents a different physical or software limitation.

Memory stalls occur when the Tensor Cores complete their current tile multiplication before the next tile has been loaded from HBM. Even with the H100's 3.35 TB/s HBM bandwidth, the Tensor Cores can consume data faster than HBM can deliver it for certain matrix dimensions, creating idle cycles while the arithmetic units wait for data.

Pipeline bubbles arise when one operation must complete before the next can begin, leaving some processing elements idle. In a Transformer layer, the attention computation must complete before the

feed-forward network can begin, and the feed-forward computation must complete before the next layer's attention can start. These sequential dependencies create brief periods where some hardware resources are unused. The local mechanism is simple: a bubble is an idle slot between dependent stages, and microbatching reduces the waste by feeding later microbatches into stages that would otherwise wait. At the infrastructure level, the lesson is that peak throughput assumes a perfectly filled pipeline, while real training includes dependency gaps that lower sustained utilization.

Communication overhead from AllReduce operations (for tensor parallelism) or gradient synchronization (for data parallelism) pauses computation entirely while data is exchanged between accelerators. Even when communication is overlapped with computation (using separate communication and compute streams), there are typically operations that cannot be overlapped because they depend on the result of the communication.

Software overhead from kernel launch latency, memory allocation, and Python-level control flow contributes an additional 5–10 percent overhead. Each CUDA kernel launch incurs approximately 5–10 microseconds of latency on the host side, and a single Transformer layer may involve 20–30 kernel launches. For small microbatches where each kernel completes in microseconds, the launch overhead can become a meaningful fraction of total execution time.

Kernel fusion partially addresses this gap by combining multiple operations (matrix multiply, bias add, activation function, dropout) into a single GPU kernel, eliminating the intermediate memory reads and writes that would otherwise stall the Tensor Cores between operations. The art of achieving high sustained utilization is largely the art of minimizing the time Tensor Cores spend idle, which is why specialized kernel libraries like FlashAttention and the Transformer Engine exist.

For capacity planning, the sustained throughput rate, not the peak rate, should be used. A training run estimated at 1,000 GPU-hours using peak FLOP/s will actually require 2,000–3,000 GPU-hours when accounting for real-world utilization. Experienced infrastructure teams track their **Model FLOPs Utilization (MFU)**, defined as the ratio of the model's useful FLOP count to the product of hardware peak FLOP/s and wall-clock time, as the primary metric for infrastructure efficiency. MFU values of 40–50 percent are considered good for large-scale transformer training; values above 50 percent indicate excellent software optimization. Section 9.9.1 formalizes MFU and develops its fleet-scale application.

2.3.4 Power wall and thermal constraints

The memory wall constrains how fast data reaches the compute units; the Roofline Model diagnoses whether compute or memory is the binding constraint; and Tensor Cores maximize the arithmetic value of every byte fetched. A third physical constraint also limits the accelerator's performance: the heat generated by all this computation. Every FLOP dissipates energy, and the faster we compute, the more heat we must remove. This is the power wall. Figure 2.10 traces the journey of energy from grid to transistor.

The cascade shown in figure 2.10 reveals why power delivery is an infrastructure problem, not merely an electrical one: these conversion losses, combined with the overhead of removing heat, determine the facility's total power draw, and the 10–40 kW concentrated at the rack PDU-to-VRM transition dictates the cooling architecture for the entire facility. Equation 2.2 captures that cooling overhead through a single facility ratio, Power Usage Effectiveness (PUE):

$$\text{PUE} = \frac{P_{\text{facility}}}{P_{\text{IT}}} \tag{2.2}$$

Here P_{facility} is total power drawn by the data center and P_{IT} is the power consumed by servers, accelerators, storage, and networking equipment. A PUE of 1.5 means the facility draws 1.5 watts from the grid for every 1 watt delivered to IT equipment; the extra 0.5 watts powers cooling, power conversion, lighting, and other overhead.

Every floating-point operation dissipates energy as heat. Dense Tensor Core activity during large matrix multiplications drives heat proportional to the switching activity of billions of transistors. The Thermal Design Power (TDP)[9] of an accelerator is the maximum sustained heat dissipation that the cooling system must handle, and it has become the single most consequential specification in fleet design. At dense fleet scale, this creates the power wall: the point where facility power delivery

[9] **TDP (Thermal Design Power)**: Often misunderstood as "power consumption," TDP specifies the maximum sustained thermal load (in watts) that the cooling solution must remove. Accelerators can briefly exceed their sustained power target during transient workloads before firmware throttles back toward the configured limit, which forces rack-level VRM and cooling designs to provision for worst-case, not average, power draw.

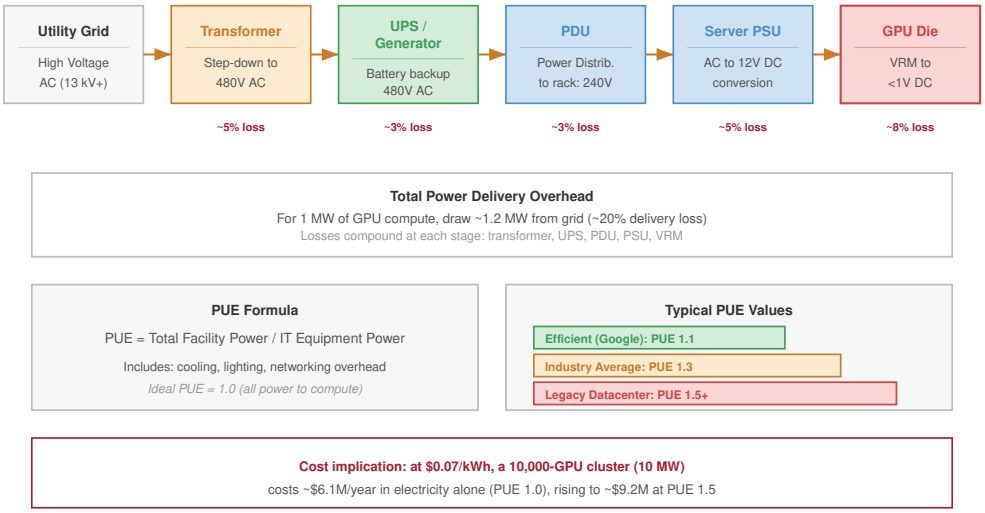


Figure 2.10: The Power Delivery Path: The journey of energy from the high-voltage grid to the low-voltage transistor involves multiple conversion stages, each with its own efficiency loss. The cumulative delivery loss, combined with overhead for cooling (measured by PUE), dictates how much power a facility must draw from the grid. The critical engineering challenge is the rack PDU to VRM transition, where 10–40 kW of power must be delivered within a single cabinet.

and heat removal, not available arithmetic units, determine how much compute can be installed and sustained.

This is the thermodynamic limit (principle 1) in action: the bottleneck of the modern AI data center is not FLOP/s, but Watts per square foot. When a rack generates 100 kW of heat, air cooling fails. The physical design of the fleet is dictated by the ability to move heat away from silicon. The closely related power density wall (principle 2) makes the consequence concrete: liquid cooling becomes a facility requirement, not an option, for large-scale training clusters.

Definition 2.4: Thermal design power (TDP)

Thermal Design Power (TDP) is the maximum sustained thermal load in watts that an ML accelerator's cooling system must continuously remove for the processor to operate at its rated clock frequency—defining both the cooling infrastructure requirement and the performance ceiling for the accelerator.

1. **Significance**: The H100 SXM5 operates at 700 W TDP. When a liquid cooling system can only sustain 500 W of heat removal (inadequate cooling), the GPU firmware reduces clock frequency via thermal throttling, dropping R_{peak} from 989 TFLOP/s to roughly 712 TFLOP/s FP16, a 28 percent reduction in training throughput that propagates directly into longer wall-clock training time and higher cost per model.
2. **Distinction**: Unlike peak power consumption (the instantaneous maximum during a single-instruction burst), TDP specifies the steady-state thermal budget: the long-term average that determines whether training can run continuously at rated performance or must throttle.
3. **Common pitfall**: A frequent misconception is that TDP is a power limit for the software. TDP is a cooling requirement: the processor exceeds its rated speed only if the cooling system removes heat at least as fast as the chip generates it; insufficient cooling causes

hardware throttling silently—no error is raised, training simply slows down and MFU drops without obvious cause.

The trajectory of TDP tells the story of a scaling regime reaching its limits. For three decades, **Dennard scaling**[10] allowed chip designers to shrink transistors while maintaining approximately constant power density: smaller transistors required lower voltage, so packing more transistors into the same area did not increase heat output per unit area. As voltage scaling and leakage limits ended that regime, architects had to treat energy, cooling, and parallelism as first-order constraints rather than free byproducts of process shrinks (Dennard et al. 1974; Hennessy and Patterson 2019). Since then, each generation of accelerator has had to trade performance against power and heat. The era of "free" performance improvements through process shrinks is over; every gain in throughput now has to be paid for in watts, cooling capacity, or specialization.

The numbers are stark: the V100 (2017) operated at 300 W TDP, the A100 (2020) increased to 400 W, the H100 (2022) reached 700 W, and the B200 (2024) draws 1000 W. In seven years, TDP has more than tripled, growing at approximately 18 percent per year.

If this growth rate continued as a planning scenario, a generation after Blackwell would approach 1,200–1,400 W per accelerator, and another step could reach 1,500–2,000 W. At 2,000 W per accelerator, a single 8-GPU node would consume 16 kW of GPU power alone, requiring liquid cooling[11] infrastructure that can remove heat at rates approaching those of industrial process cooling equipment.

10 | **Dennard Scaling**: Named after Dennard et al. (1974) at IBM, who showed how MOSFET dimensions, voltage, and current could scale so that power density stayed roughly constant. Later voltage-scaling limits and leakage effects ended that constant-power-density regime; Hennessy and Patterson describe the resulting post-Dennard power wall as one reason modern architecture turned toward parallelism and specialization (Hennessy and Patterson 2019). For ML infrastructure, the V100-to-B200 TDP trajectory (300 W to 1,000 W) is an illustration of the post-Dennard planning problem: cooling and power delivery are now co-equal design constraints with the silicon itself.

11 | **Liquid Cooling Flow Rate**: A rack of GB200 NVL72 nodes (120 kW) requires approximately 150–200 liters per minute (LPM) of coolant flow. At these rates, the primary risk shift is from thermal throttling to mechanical failure (leaks, pump cavitation), necessitating the industrial-grade manifolds and quick-disconnect fittings common in chemical processing plants.

Systems Perspective 2.2: The end of Dennard scaling

For three decades, Dennard scaling allowed architects to increase transistor count without increasing power density, as smaller transistors required proportionally less voltage. That free lunch ended around 2006 when process nodes dropped below 28nm, where quantum effects like subthreshold leakage and gate oxide tunneling prevent further voltage reduction. The result is that power density rises with transistor density when voltage no longer scales proportionally: accelerators that deliver more TFLOP/s often demand more watts. The transition from the V100 (12nm) to the H100 (4nm) illustrates this physics: while transistor density tripled, the inability to scale voltage meant TDP grew from 300 W to 700 W. The "free" efficiency gains of the silicon era are over; higher performance per watt increasingly comes from architectural specialization – Tensor Cores, systolic arrays, precision engineering – or from accepting higher power and cooling requirements. For ML infrastructure planners, power and cooling capacity must be sized against compute demand, and facility designs cannot assume that later hardware automatically lowers power density.

TDP is not merely a specification on a data sheet; it is a physical constraint that propagates through every level of the infrastructure stack. The power delivery chain, cooling system, rack design, and facility electrical infrastructure must all be designed to accommodate deployed accelerators and the plausible TDP range of future refresh cycles.

Accelerators implement sophisticated power management to operate within their TDP envelope. When the workload does not fully use all SMs (for example, during the communication phase of a training step when the GPUs are waiting for AllReduce to complete), the GPU firmware can disable the clock signal to idle circuitry through **clock gating**, eliminating dynamic switching power and freeing thermal headroom for the active SMs. A separate mechanism, **dynamic voltage and frequency scaling (DVFS)**, adjusts the chip's clock frequency and supply voltage to stay within power and thermal limits; during a large matrix multiplication that activates all SMs simultaneously, DVFS reduces the clock frequency to keep total power within TDP. When some SMs are idle, the active SMs can boost frequency because total chip power is below TDP, so the actual clock, and therefore the actual throughput, varies throughout a training step.

Dynamic power management means that the actual power draw of a GPU varies continuously during training. A typical training step might see power swing from 400 W (during communication phases when most SMs are idle) to 700 W (during the forward and backward pass when all Tensor

Cores are active) and back, with each transition occurring in microseconds. The VRMs on the baseboard must respond to these transitions fast enough to maintain a stable supply voltage despite the rapidly changing current draw.

To appreciate the physical challenge, consider what 700 W means in terms of heat flux. An H100 die measures approximately 814 mm^2 (roughly 29×28 mm). Dissipating 700 W from this area produces a heat flux of approximately 86 W/cm^2. For comparison, the surface of an electric stovetop burner at high heat produces approximately 10 W/cm^2. An H100 die generates 8.6× the heat flux of a stove burner, concentrated on an area the size of a large postage stamp.

The B200 pushes even further: at 1,000 W across a dual-die package, the average heat flux remains comparable, but the total energy that must be removed per package increases by 43 percent. Removing this heat without allowing the junction temperature to exceed 83 degrees Celsius (the typical operating maximum for HBM reliability) requires thermal solutions with extremely low thermal resistance from die to coolant.

The critical component in this thermal path is the **Thermal Interface Material (TIM)** – the microscopic layer of phase-change material or liquid metal between the silicon die and the cold plate (for liquid cooling) or heatsink (for air cooling). Under the extreme thermal cycling of deep learning workloads, where die temperatures spike from 40 degrees C to 80 degrees C in milliseconds during matrix multiplications and drop back during communication phases, inferior TIM can pump out (migrate away from the contact surface) or crack. A mere 5 percent degradation in TIM thermal conductivity forces the GPU to thermally throttle, reducing clock frequency by hundreds of MHz to protect the silicon. This degradation is a *gray failure*, a fault that degrades rather than halts: the GPU continues to function, but at reduced performance that silently degrades the entire cluster's throughput, a failure class we return to in fleet monitoring. Fleet management systems that track per-GPU junction temperatures over time can detect TIM degradation as a gradual upward trend in temperature at constant workload, enabling proactive replacement before the performance impact becomes severe.

At rack scale, four nodes of eight such chips require 33.5 kW of facility-relevant power, enough to heat several homes in winter. To put this in more tangible terms, a single ML rack at full power could run about 33 household space heaters simultaneously, concentrated in a volume roughly the size of a large refrigerator.

At pod scale, the power demand reaches megawatts, requiring dedicated electrical substations and industrial cooling plants. A 10,000-GPU cluster consumes 7–10 MW, comparable to the electrical demand of a large factory or a small town of several thousand residents.

The physical volume of a 10,000-GPU pod operating at this power density is surprisingly compact: the compute hardware (excluding networking and storage) fits in approximately 300 racks, occupying a data center floor area of roughly 1,500 square meters (about one third of an American football field). The density is the point: keeping the accelerators physically close minimizes cable lengths, which reduces signal propagation delay and enables higher interconnect bandwidth.

Physical density, however, makes power delivery and cooling exponentially harder, creating the engineering challenges that define the rack and pod levels of the hierarchy. The paradox of modern ML infrastructure is that computational efficiency demands *physical proximity*, while thermal management demands *physical separation*. Every rack design is a negotiation between these opposing forces.

This frequency variability creates a subtle interaction with software optimization. A kernel that achieves high SM occupancy (many active warps per SM) draws more power, potentially triggering a frequency reduction that partially offsets the occupancy benefit. Conversely, a kernel with moderate occupancy may run at a higher clock frequency, achieving comparable throughput at lower power. Power-normalized throughput (TFLOP/s per watt) is therefore a more robust comparison metric than raw TFLOP/s, and it explains why the same GPU can show different sustained clock frequencies depending on the workload and the cooling solution.

At fleet scale, DVFS also affects power planning. The data center's power delivery must be designed for the *worst-case* power draw (all GPUs at TDP simultaneously), but the *average* power draw is typically 70–85 percent of TDP because DVFS reduces frequency during communication phases and because not all SMs are fully occupied at all times. Some operators deliberately set lower power caps on their GPUs (for example, capping an H100 at 600 W instead of 700 W), accepting a 10–15 percent

reduction in peak performance in exchange for 15 percent lower power consumption and the ability to fit more GPUs within a fixed power budget. Power capping is a form of the compute-efficiency trade-off at the operational level.

The power efficiency trajectory provides a partial counterweight to rising TDP. While absolute power has increased, the advertised low-precision TFLOP/s delivered per watt has improved dramatically: from 0.42 TFLOPs/s/W (V100 FP16) to 2.83 TFLOPs/s/W (H100 FP8) to 4.50 TFLOPs/s/W (B200 FP8). These are not apples-to-apples FP16 numbers; the precision mode is part of the hardware trend. For a fixed compute budget that can use the newer precision modes, each generation requires fewer chips and therefore less total power. Large-model capability targets, however, can grow faster than efficiency improves, so the absolute power demand of training the next larger model may still increase. This is the treadmill of infrastructure: efficiency gains buy time, but scale growth consumes it.

The implications of TDP for fleet design are profound and will recur throughout this chapter. At the node level, TDP determines how many accelerators can share a single chassis (we examine this in section 2.4). At the rack level, TDP dictates the transition from air cooling to liquid cooling (section 2.5). At the pod level, TDP drives the megawatt-scale power delivery systems that connect the fleet to the electrical grid (section 2.6). The accelerator is the engine, but TDP is the exhaust, and every level of infrastructure above the die exists, in part, to manage that exhaust.

2.3.5 Accelerator selection for ML workloads

An accelerator's compute throughput, memory bandwidth, memory capacity, and power consumption are meaningful only in relation to a specific workload. The same H100 that delivers near-peak utilization during large-batch training may achieve less than 5 percent utilization during single-request inference, because the binding constraint shifts from arithmetic to memory bandwidth. Selecting the right accelerator therefore requires mapping each workload archetype to its position on the Roofline plot and identifying which physical resource dominates its performance.

Start with the 175B running example. During training, large batches push the forward and backward passes above the Roofline ridge point, so the immediate constraint is arithmetic throughput. Peak low-precision TFLOP/s matters, but only after memory capacity is sufficient to hold the sharded model state and after the interconnect can keep tensor and data parallel groups synchronized. H100, B200, and TPU v5p pods are all plausible answers to this training problem; the final choice depends on the software stack as much as the silicon. A PyTorch-first research team values the GPU ecosystem, while a JAX/XLA-centered organization may accept tighter compilation constraints for TPU cost efficiency.

Serving the same language model changes the answer. During autoregressive decode, every generated token streams weights through memory, so the key metric becomes memory bandwidth per dollar rather than peak TFLOP/s. At batch size 1, the H100's 3.35 TB/s bandwidth determines throughput while much of the arithmetic silicon waits. Batching raises arithmetic intensity by reusing the same weight movement across more requests, which is why serving systems work so hard to batch without violating tail-latency SLOs. The workload did not change names, but the binding term in the iron law changed, and the accelerator choice changes with it.

Recommendation models split the decision again. Embedding table lookup is dominated by random access and memory capacity, because each request retrieves small vectors from TB-scale tables with poor locality. The dense tower that consumes those embeddings is compute bound. A hybrid CPU-GPU design follows from that split: CPU and DRAM capacity handle the sparse lookup path, while GPUs accelerate the dense neural network layers. A single headline TFLOP/s number cannot describe this workload because it has two different bottlenecks inside one request.

Vision and diffusion workloads move back toward arithmetic throughput because convolutions, attention blocks, and denoising networks reuse data heavily. Diffusion inference makes this especially clear: the Stable Diffusion v1.5 profile in MLSysIM uses 50 denoising steps, so the service pays for a sequence of full model evaluations rather than one decode step per token. For these workloads, TFLOP/s per dollar often dominates bandwidth per dollar once model state fits in memory.

Mixture-of-Experts (MoE) adds a routing constraint: the sparsely gated layer activates only part of a much larger parameter set for each token (Shazeer et al. 2017). Mixtral-8x7B (A. Q. Jiang et al. 2024), for example, has 46.7B total parameters but only 12.9B active parameters per token because

the router chooses 2 experts from 8 experts. That compute saving introduces a many-to-many token exchange pattern, the AllToAll collective examined in Chapter 6, and load-balance risk because tokens must be routed to different expert placements. High-bisection fabrics and load-balancing mechanisms such as auxiliary losses, which penalize overloaded experts, or capacity factors, which cap how many tokens each expert accepts, determine whether the theoretical saving appears in the training trace (Fedus et al. 2022; Lepikhin et al. 2021).

Multi-modal models add heterogeneity rather than only routing. They combine text decode, vision processing, audio features, and sometimes video, each with different arithmetic intensity and batching behavior. These workloads favor platforms with flexible scheduling and strong bisection bandwidth rather than accelerators optimized for one regular dataflow.

> Systems Perspective 2.3: Matching hardware to workload
>
> The fundamental insight from the Roofline Model is that no single accelerator is optimal for all workloads. An H100 that delivers outstanding training throughput for a 175B LLM can sit mostly idle when serving that same model at batch size 1. A TPU pod that provides exceptional cost efficiency for Transformer training may be poorly suited for a recommendation model with irregular memory access patterns.
>
> A mature fleet is therefore heterogeneous by design because each workload makes a different resource decisive. Training clusters spend on arithmetic throughput and fast accelerator-to-accelerator links because synchronization sits in the critical path. Serving clusters spend on memory bandwidth per dollar and latency control because decode streams weights under tail-latency SLOs. Embedding systems spend on DRAM capacity and locality management because sparse lookups dominate the request. Fine-tuning clusters sit between those regimes, balancing memory headroom with enough compute to keep iteration fast. The accelerator spectrum is a design space that must be navigated for each workload independently. Organizations that deploy a single hardware configuration for all workloads inevitably overpay: either the hardware is overprovisioned for simple workloads or underprovisioned for demanding workloads.

2.3.5.1 Precision and throughput trade-offs

The relationship between numerical precision and hardware throughput is one of the most important considerations for infrastructure planning, because it directly affects how much useful work each accelerator can perform per second. Lower precision representations use fewer bits per number, which has two compounding benefits: more numbers fit in the same HBM capacity (reducing memory pressure), and the arithmetic units can process more operations per cycle (increasing throughput).

The precision landscape for ML has evolved rapidly. Table 2.6 summarizes representative precision formats on H100-class accelerators and their common use cases.

Table 2.6: Precision Formats and Hardware Throughput on H100: The table uses dense peak throughput where Tensor Core support exists; structured sparsity can roughly double several of these advertised peaks. The choice of precision affects both the training dynamics (convergence, numerical stability) and the infrastructure requirements (memory capacity, bandwidth utilization). TF32 preserves the FP32 exponent width but reduces mantissa precision so Tensor Cores can accelerate common FP32 training workloads.

Format	Bits	Expo-nent	Mantissa	H100 Dense TFLOP/s	Typical Use
FP32	32	8	23	~67	Master weights, loss computation
TF32	19	8	10	494	Training (NVIDIA default)
BF16	16	8	7	989 TFLOP/s	Training, fine-tuning
FP16	16	5	10	989 TFLOP/s	Training with loss scaling
FP8 (E4M3)	8	4	3	1,979 TFLOP/s	Forward pass, weights

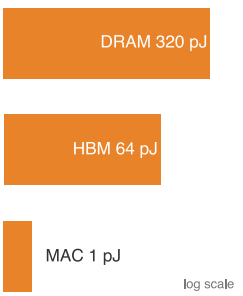
Format	Bits	Expo-nent	Mantissa	H100 Dense TFLOP/s	Typical Use
FP8 (E5M2)	8	5	2	1,979 TFLOP/s	Backward pass, gradients
INT8	8	N/A	N/A	1,979 TFLOP/s	Post-training quantization
INT4	4	N/A	N/A	3,958	Weight-only quantization

The infrastructure implications are substantial. A fleet designed for BF16 training, a common baseline for large models, delivers 989 TFLOP/s per GPU on an H100. The same fleet running FP8 training delivers 1,979 TFLOP/s per GPU, effectively doubling the compute density without additional hardware. For a 10,000-GPU cluster priced at roughly $350,000 per 8-GPU node, the difference between BF16 and FP8 training is equivalent to adding another 10,000 GPUs, or roughly $438 million in node hardware.

However, not all workloads can use FP8 without accuracy degradation. The reduced mantissa precision (3 bits in E4M3) means that values must be scaled carefully to avoid overflow (values exceeding the representable range saturate or become nonfinite, depending on format and implementation) or underflow (small values rounded to zero). The Transformer Engine's dynamic per-tensor scaling addresses this challenge for standard Transformer architectures, but custom model architectures with unusual activation distributions may require manual precision tuning. The infrastructure team must therefore work closely with the model team to determine the lowest precision that maintains acceptable accuracy, as this decision directly affects the effective throughput and therefore the required cluster size.

For inference, **quantization** is one of the most impactful optimizations for serving economics because it directly alters the hardware topology required for large models. Our 175B model at FP16 requires 350 GB, necessitating a minimum of five 80 GB H100 GPUs simply to load the parameters. At INT8, this drops to 175 GB, fitting on three GPUs. At INT4, it shrinks to 87.5 GB, fitting on two GPUs. For memory-bandwidth-bound inference, reading 4-bit weights instead of 16-bit weights effectively quadruples the available bandwidth for weight loading, proportionally reducing per-token latency. A production team deploying at INT4 rather than FP16 can reduce the memory footprint and bandwidth demand of the inference fleet substantially, subject to the quality loss of the chosen quantization method and workload.

For training, many workloads use **mixed-precision training**, a strategy that decouples storage precision from arithmetic precision. A "master copy" of weights remains in FP32 for numerical stability, while the computationally intensive forward and backward passes are cast to BF16 or FP16 to exploit the full throughput of Tensor Cores. Gradients are accumulated in FP32 to prevent underflow before the optimizer updates the master weights. The H100's Transformer Engine extends this paradigm by dynamically selecting between FP8 and FP16 on a per-layer basis, potentially doubling throughput again for layers resilient to reduced precision. The cumulative effect on training velocity can be multiple-fold when the model remains numerically stable at the lower precision.



Data movement dwarfs arithmetic in energy cost.

2.3.5.2 Energy cost of data movement

The Roofline Model and token latency analysis demonstrate that data movement limits performance. Data movement also dominates energy consumption, with direct implications for fleet economics. A useful normalization is an FP16 operand: reading one 16-bit value from HBM at roughly 4 pJ/bit costs about 64 picojoules (pJ), while one FP16 multiply-accumulate (MAC) costs roughly 1 pJ. The raw operand fetch can therefore cost about 64× the arithmetic operation, before tiling and reuse amortize that movement across many MACs.

This operand-level ratio explains why accelerator architects devote so much silicon area to data reuse. A Tensor Core's tile-based execution model loads a small matrix into local registers and reuses each element hundreds of times (once per element in the opposing matrix), amortizing the HBM access cost across hundreds of 1 pJ compute operations. Without this reuse, the energy budget would be dominated by memory access, and most of the chip's power would be spent heating wires rather than switching transistors.

At fleet scale, the energy cost of data movement determines the electricity bill. For a 10,000-GPU cluster running at 700 W per GPU, approximately 400–500 W per GPU is consumed by the memory subsystem (HBM reads and writes, on-chip network transfers, register file accesses), and only 150–250 W is consumed by the Tensor Cores performing useful arithmetic. The rest goes to clock distribution, I/O, and leakage. Improving data reuse, through techniques like kernel fusion, FlashAttention, and activation checkpointing, reduces the energy wasted on data movement and increases the fraction of power that performs useful computation.

The energy perspective provides a physical foundation for understanding why different parallelism strategies have different efficiencies. Tensor parallelism requires data movement over NVLink (approximately 5 pJ per bit at the link level, plus the energy of the NVSwitch chips). Data parallelism requires data movement over InfiniBand (approximately 15–20 pJ per bit, including the energy of the HCA and switch). Pipeline parallelism moves less data but requires more complex scheduling. Each parallelism strategy represents a different point in the trade-off between communication energy and computation efficiency.

Napkin Math 2.3: The energy cost of data movement

The iron law of modern computing is that moving data costs significantly more energy than manipulating it. We can prove this by analyzing the energy hierarchy of a single operation at the 4nm process node.

Performing one FP16 multiply-accumulate (**MAC**) operation – the atomic unit of deep learning – consumes approximately 1 picojoule (pJ). This is the baseline cost of useful work. Reading a single FP16 operand (16 bits) from HBM consumes roughly 4 pJ per bit, totaling 64 pJ. Reading that same operand from off-package DRAM costs approximately 20 pJ per bit, or 320 pJ. The asymmetry is staggering: reading a value from HBM costs 64× more energy than computing on it. Retrieving it from standard DRAM costs 320× more.

The macro impact appears when serving our 175B model. Generating a single token requires loading the entire 350 GB weight tensor from HBM:

$$\text{Data Movement Energy} = 350 \text{ GB} \times 8 \text{ bits/byte} \times 4 \text{ pJ/bit} \approx \mathbf{11.2} \text{ Joules}$$

The computational cost for the associated ≈ 175 billion MACs (about 350 billion FLOPs) is trivial by comparison:

$$\text{Computation Energy} = 175 \times 10^9 \text{ MACs} \times 1 \text{ pJ/MAC} \approx \mathbf{0.175} \text{ Joules}$$

Moving the weights to the compute units consumes 64× more energy than the math itself in this simplified decode model. This energy penalty is why HBM is engineered for physical proximity to the GPU die – shorter traces mean less capacitance and lower energy per bit. It also explains why techniques that reduce data movement, such as quantization (moving fewer bits) and kernel fusion (preventing round-trips to memory), are the primary levers for improving both system performance and power efficiency.

2.3.5.3 Benchmarking accelerator performance

A procurement team comparing accelerators needs a published number that predicts the performance the organization will actually achieve. Peak TFLOP/s, memory bandwidth, and TDP describe *potential*, not *achieved* performance, so the decision turns on whose measured results to trust and under what rules they were produced. The industry-standard reference is MLPerf, maintained by MLCommons: MLPerf Training reports the time to train reference models to a target accuracy, and MLPerf Inference reports serving throughput and latency under realistic batching and streaming conditions. Both capture complete system performance, including software stack efficiency, communication overhead, and memory management, not just the raw silicon.

The number that actually predicts an organization's results is the *closed*-division result, and understanding why is the core of reading MLPerf well. The closed division requires every submission to use the same model architecture, hyperparameters, and training recipe, isolating the hardware

and system software as the only variables and making results directly comparable across vendors. The open division allows arbitrary model modifications, custom kernels, and software optimizations, which demonstrate a platform's ceiling but make cross-vendor comparison unreliable. Infrastructure teams should weight closed-division results more heavily, because they reflect the performance achievable with standard frameworks and configurations, not the performance achievable only by the vendor's own optimization team.

For our 175B model, no single benchmark captures the full picture. The training phase is compute bound at large batch sizes, making peak TFLOP/s and scaling efficiency the dominant metrics. The inference phase at batch size 1 is memory-bandwidth-bound, making GB/s per dollar the relevant metric. The fine-tuning phase falls between these extremes, with moderate batch sizes placing the workload near the Roofline ridge point. A comprehensive evaluation must benchmark all three phases on the candidate hardware, using the organization's actual model and data pipeline rather than relying solely on published MLPerf numbers.

> Systems Perspective 2.4: Beyond peak specifications
>
> When evaluating accelerator options, the following metrics provide a more complete picture than peak TFLOP/s alone:
>
> - **Model FLOPs utilization (MFU)**: The ratio of achieved FLOP/s during real training to peak hardware FLOP/s. An MFU of 50 percent means the hardware spends half its time on useful computation and half waiting for data, synchronizing, or idling.
> - **Time-to-Train (TTT)**: The wall-clock time to train a reference model to a target metric. This captures all system-level effects that MFU alone does not, including I/O stalls, checkpointing overhead, and job restart delays.
> - **Cost per Token**: For language model training, the total cost (hardware amortization plus electricity) divided by the number of tokens processed. This metric normalizes across different hardware generations, cluster sizes, and pricing models.
> - **Tokens per Second per Dollar**: The inverse of cost per token, useful for comparing the economic efficiency of different systems.
>
> No single metric tells the whole story. A system with high MFU but high cost per GPU-hour may be less economical than a system with lower MFU but cheaper hardware. The right metric depends on whether the organization is optimizing for time (training must finish by a deadline), cost (minimize total expenditure), or throughput (maximize tokens processed per unit time).

2.3.5.4 Inference-specific infrastructure considerations

While this chapter focuses primarily on training infrastructure (because training drives the most demanding infrastructure requirements), the infrastructure for serving trained models at scale deserves separate attention because the design constraints differ fundamentally from training. Training infrastructure is optimized for *throughput*: maximizing the total number of tokens processed per unit time across the entire cluster. The workload is a single, long-running job that occupies the full cluster for days or weeks. The communication pattern is predictable and repetitive. The acceptable latency for any individual operation is milliseconds to seconds.

Serving infrastructure is optimized for *latency* and *cost per query*: minimizing the time each user waits for a response while keeping the per-query cost low enough for the business model to be viable. The workload consists of millions of independent, short-lived requests arriving at irregular intervals. The communication pattern is minimal (each request is processed independently on a single node or small group of nodes). The acceptable latency is tens of milliseconds for interactive applications.

The differing requirements drive distinct hardware and architecture choices, summarized in table 2.7.

Table 2.7: Training vs. Serving Infrastructure: Training and serving optimize different hardware behaviors, so capacity planning must compare utilization, model placement, network traffic, and cost in the mode the fleet will actually run.

Dimension	Training infrastructure	Serving infrastructure
GPU utilization	Large batch sizes push workloads into the compute-bound regime, so clusters commonly achieve 30–50% MFU.	Low-batch serving is deeply memory bound and can achieve less than 5% MFU, so batching improves utilization at the cost of latency.
Model placement	Tensor and pipeline parallelism distribute a single model across many GPUs.	Replicas of the model, or large shards of it, handle independent requests, so memory demand scales with concurrent serving capacity.
Network traffic	Gradient synchronization requires high-bandwidth, low-latency inter-GPU communication.	Independent requests need little inter-GPU traffic but substantial client-facing bandwidth for millions of API requests per second.
Cost metric	Training cost is a fixed, one-time expenditure measured in dollars per run.	Serving cost is a variable expense measured in dollars per 1,000 tokens generated, and popular models can exceed their training cost within weeks.

The table shows why serving cannot be planned as training with smaller batches: the optimization target changes from synchronized throughput to latency-bounded, continuously billed requests.

The implication is that effective serving infrastructure often uses different hardware, different software, and different facility designs than the training infrastructure. Some organizations use previous-generation GPUs (A100s) for serving because the memory bandwidth per dollar can be competitive with newer GPUs, and the lower TDP (400 W vs. 700 W) allows higher rack density and lower cooling costs.

The inference workload itself bifurcates into two distinct phases with opposing bottlenecks. The **prefill** phase processes the input prompt, performing a large matrix-matrix multiplication that is compute bound and achieves high Tensor Core utilization. The **decode** phase generates tokens one by one, performing matrix-vector multiplications that are deeply memory-bandwidth-bound. To maximize hardware utilization, high-performance serving systems employ **continuous batching**, which schedules requests at the iteration level rather than the request level. This allows the engine to inject a new prefill computation for a waiting request into the idle compute slots of an ongoing decode batch, dynamically filling the GPU's arithmetic pipelines. For our 175B model, serving 10,000 requests per second with a 50 ms time-to-first-token service level agreement (SLA) requires distributing traffic across hundreds of 8-GPU replicas (each holding the full model via tensor parallelism), with a load balancer routing requests to the replica with the lowest queue depth. Unlike training, where 99.9 percent reliability can be acceptable with checkpointing, inference architectures must account for tail latency, where a single slow replica can violate the SLA for the entire request batch.

2.3.5.5 The accelerator decision matrix

A practical decision framework synthesizes these workload-specific characteristics by mapping each archetype (section 1.6.1) to the hardware resource that dominates its performance and cost. Table 2.8 makes the selection rule explicit: choose for the binding constraint, not the largest advertised peak number.

Table 2.8: The Accelerator Decision Matrix: The optimal accelerator depends not on peak specifications but on which physical resource – compute, bandwidth, or capacity – is the binding constraint for the target workload. The Roofline Model provides the diagnostic: compute the arithmetic intensity, locate the workload relative to the ridge point, and select hardware that maximizes the binding resource per dollar.

Workload	Binding Constraint	Key Metric	Recommended Class
LLM Training (>100B)	Compute (high batch size)	Peak TFLOP/s, NVLink BW	H100/B200, TPU v5p
LLM Inference (batch=1)	Memory bandwidth	GB/s per dollar	A100, H100 (bandwidth/$)
LLM Inference (batched)	Compute + bandwidth	TFLOP/s and GB/s	H100, B200

Workload	Binding Constraint	Key Metric	Recommended Class
Vision Model Training	Compute (large spatial dims)	Peak TFLOP/s	H100/B200, GPU preferred
Recommendation (embeddings)	Memory capacity	GB per dollar	CPU DRAM + GPU hybrid
Fine-tuning (<13B)	Memory capacity	HBM capacity	A100 (cost-effective)
Research/Prototyping	Flexibility	Software ecosystem	GPU (CUDA), avoid ASICs

The binding constraint shifts with the lifecycle phase, as table 2.8 shows for the 175B running example. During training, large batch sizes push the workload into the compute-bound regime, making peak TFLOP/s the dominant metric. The H100 or B200, with their high Tensor Core throughput and fast NVLink for tensor parallelism, are the natural choices. During inference at low batch sizes, the same model becomes deeply memory bound, and the relevant metric shifts to bandwidth per dollar. An A100 with adequate HBM bandwidth at a lower price point may deliver better cost-per-token than an H100 whose additional TFLOP/s go unused.

The organizational dimension adds a further consideration. Research labs that modify model architectures weekly need the flexibility of the GPU's CUDA ecosystem, where custom kernels can be written and tested in hours. Production teams running a fixed Transformer architecture at scale for months may benefit from TPUs, where the XLA compiler's whole-program optimization can achieve higher sustained utilization than hand-tuned CUDA kernels for standard operations. Custom ASICs make economic sense only for organizations with enough scale to amortize the $50–$200 million NRE cost and enough workload stability to justify a 2–3 year design cycle. The accelerator spectrum is ultimately an economic question answered at the intersection of workload physics, organizational scale, and time horizon.

Checkpoint 2.3: Accelerator selection

Your team needs to deploy a 70B-parameter model for both training and inference. Training will use batch size 2048 across 256 GPUs for 3 months. Inference will serve 10,000 requests per second at batch size 1 for 2 years.

- ☐ **Training intensity**: Use the Roofline Model to determine the arithmetic intensity of each workload on an H100.
- ☐ **Hardware split**: Decide whether training and inference should use the same hardware or different hardware.
- ☐ **Two-year cost**: Calculate the 2-year inference cost difference between H100s ($4.00/GPU-hour) and A100s ($2.00/GPU-hour), accounting for throughput at the workload's arithmetic intensity.

With the accelerator's physics established, we face a concrete problem. Our 175B model requires 350 GB of memory for its weights alone in FP16, and Adam-style training state raises the requirement to multiple terabytes once optimizer moments, gradients, and activations are included. A single H100 provides 80 GB of HBM. No single accelerator can hold this model. We must expand to the next physical level: the node, where figure 2.11 contrasts the two dominant approaches to wiring multiple accelerators within a single chassis.

2.4 The Node

The node is the first scale level where accelerator choice becomes an interconnect problem. Multiple GPUs can now share one chassis, but their useful behavior depends on whether the local fabric behaves like a slow ring, a dense mesh, or a switch-backed crossbar.

Aggregating accelerators solves the capacity problem only if the interconnect between them is fast enough that the group behaves as one machine rather than as several machines passing messages. The 175B model's weights span at least five accelerators, and the full training state spans an entire 8-H100 node and more, so the parameters of a single layer now live on different chips and must be reassembled on every forward and backward pass. Whether that reassembly is cheap or

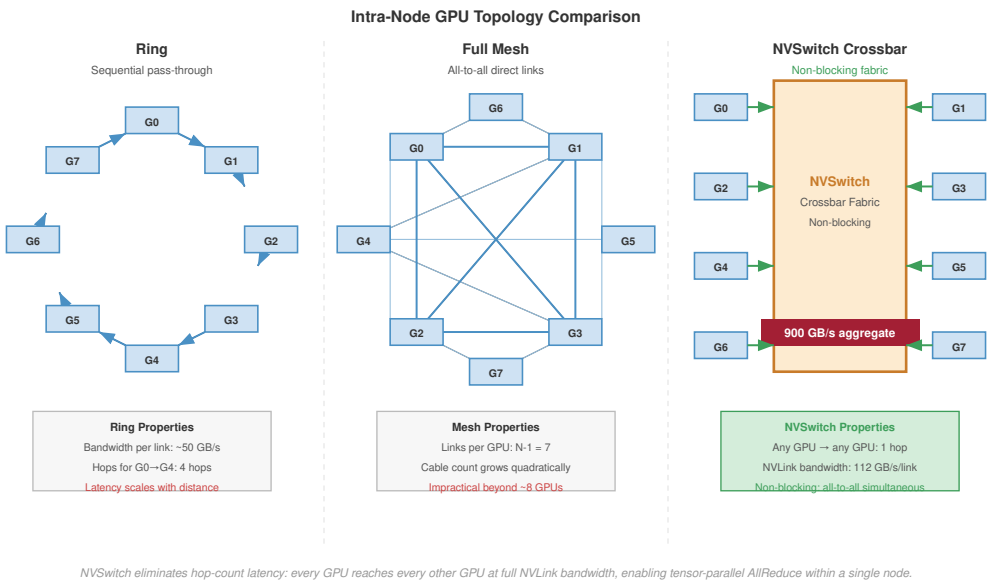


Figure 2.11: Intra-Node GPU Topology Comparison: Three approaches to wiring GPUs inside a node. (Left) Ring: sequential pass-through, ~50 GB/s per link, latency scales with distance. (Center) Full Mesh: all-to-all direct links, $N-1$ links per GPU, impractical beyond ~8 GPUs. (Right) NVSwitch Crossbar: non-blocking fabric with 112 GB/s per NVLink and 900 GB/s aggregate, any-GPU-to-any-GPU in one hop, enabling tensor-parallel AllReduce within a node.

ruinous depends entirely on the local fabric, which is why the node is the first scale level where the accelerator's specifications matter less than the wires between accelerators. The training system must combine tensor parallelism with optimizer sharding and memory offload, and each of those strategies places a different demand on the interconnect.

Definition 2.5: Node

Node is the ML training cluster's physical server chassis that aggregates multiple accelerators (typically 8) through a high-speed intra-node interconnect (NVLink or ICI), creating the fundamental boundary between high-bandwidth local communication and the order-of-magnitude slower inter-node network fabric.

1. **Significance**: The NVLink bandwidth within a DGX H100 node reaches 900 GB/s bidirectional, or 450 GB/s per direction, while inter-node InfiniBand NDR provides 50 GB/s, a 9× per-direction gap that constrains where each parallelism strategy can run. Tensor parallelism, which AllReduces every layer's output, must stay within the node; pipeline parallelism, which transfers only stage-boundary activations, can span nodes over InfiniBand. The node is also the primary failure domain: a single node failure in a bulk synchronous parallel (BSP) training job, where workers advance in lockstep at step barriers, idles the entire cluster until recovery.
2. **Distinction**: Unlike a single accelerator (which provides fast HBM bandwidth but limited capacity), a node aggregates 8× the HBM capacity and 8× the compute of a single chip – enough to place large FP16 weight shards (8×80 GB = 640 GB on DGX H100), but not enough to keep the full Adam training state in HBM.
3. **Common pitfall**: A frequent misconception is that nodes in a distributed training job operate independently. In BSP training, all nodes are synchronously coupled at every step barrier: a performance degradation on one node (from thermal throttling, a faulty NIC, or a slow storage read) stalls the entire cluster and drops overall MFU for the duration of that step.

The node exists because of a gap in the memory hierarchy. HBM provides enough *bandwidth* to feed the accelerator's arithmetic units, but not enough *capacity* to hold the full 175B-parameter running example. Host DRAM provides capacity (512 GB to 2 TB per server) but not enough bandwidth.

The node bridges this gap by aggregating the HBM of multiple accelerators into a shared pool, connected by an interconnect fast enough to allow cooperative computation across all of them. The key engineering insight is that by placing 8 accelerators in a single chassis and connecting them with a high-speed fabric, the node creates a virtual accelerator with $8\times$ the memory capacity and $8\times$ the compute throughput of any individual chip, while the fast interconnect ensures that this virtual accelerator can operate nearly as efficiently as a single monolithic device.

The economic argument for multi-accelerator nodes is equally compelling. Consider the alternative: building a single accelerator with enough HBM to hold a 175B model (350 GB in FP16). Current packages provide roughly 16–24 GB per HBM stack, so a 350 GB single-accelerator design would require on the order of 15–22 HBM stacks – far beyond the 5-stack H100 or 8-stack B200-class package. The manufacturing cost of such a package would be prohibitive (yield decreases with package and interposer area), and the power delivery to a single chip with enough Tensor Cores to use all that bandwidth would exceed any practical cooling solution. By distributing the computation across 8 accelerators connected by NVLink, the node achieves the aggregate memory capacity and compute throughput of this hypothetical super-chip while remaining within manufacturable and coolable boundaries.

Understanding how the node partitions memory across its components is essential for selecting parallelism strategies. Consider the memory budget for training our 175B model. The model weights in FP16 require 350 GB. The Adam optimizer maintains two additional copies of every parameter (first and second moments) in FP32 precision, adding $175 \times 10^9 \times 2 \times 4$ bytes = 1,400 GB. Gradients require another 350 GB. The total is approximately 2.1 TB before activations, far exceeding both a single accelerator's 80 GB HBM and an 8-H100 node's 640 GB aggregate HBM.

When using ZeRO or FSDP-style optimization (which shards optimizer states, gradients, and optionally parameters across data-parallel workers), the training system avoids replicating this full 2.1 TB state on every GPU. Tensor parallelism first reduces each GPU's active weight shard; data-parallel sharding then spreads optimizer state and gradients across many workers; activation checkpointing reduces the activation peak; and offload moves cold optimizer state to host DRAM over PCIe. The node is therefore a high-bandwidth compute and communication unit, not a standalone container for the entire training state.

The node's internal memory hierarchy, consisting of HBM (fast, small), host DRAM (medium speed, larger), and Non-Volatile Memory Express (NVMe) storage (slow, very large), forms a tiered system that distributed training frameworks exploit aggressively. Chapter 5 examines these memory optimization strategies in detail.

2.4.1 Bandwidth hierarchy

The defining characteristic of multi-accelerator systems is not the *number of chips* but the *speed at which they can exchange data*. Data movement speed drops by orders of magnitude as it crosses physical boundaries. Each boundary represents a different physical medium, a different connector technology, and a different set of engineering constraints. This physical ordering of data transfer rates, from on-chip SRAM at the fast end to the wide-area network at the slow end, is the **bandwidth hierarchy**, and understanding it is essential because it dictates where in the hierarchy each type of parallelism can operate efficiently.

The hierarchy reflects the physical cost of distance: shorter signal paths over denser wiring achieve higher throughput at lower energy per bit, so at each boundary the effective bandwidth (BW) drops by roughly an order of magnitude while latency (L_{lat}) rises. That gradient sets a scaling ceiling for distributed training. The common mistake is to treat all cluster communication as equal, but parallelism placement must be hierarchy-aware: putting high-frequency synchronization such as tensor parallelism on a slow tier idles the compute units (R_{peak}) and collapses scaling efficiency (η_{scaling}).

The bandwidth hierarchy is best understood as a series of concentric zones around each accelerator. The innermost zone is the HBM on the same package, with terabytes per second of bandwidth and sub-microsecond latency. The next zone encompasses the other accelerators within the same node,

reachable over NVLink at hundreds of gigabytes per second with microsecond-scale latency. The outermost zone spans all other nodes in the cluster, connected by InfiniBand at tens of gigabytes per second with latency measured in single-digit microseconds. At each zone boundary, the bandwidth drops by roughly an order of magnitude while the latency increases by roughly an order of magnitude. Table 2.9 maps those zones to the parallelism strategies they can support.

Table 2.9: The Bandwidth Hierarchy: Each physical boundary introduces an order-of-magnitude bandwidth cliff. These cliffs are not engineering failures to be optimized away; they reflect fundamental differences in the physics of each interconnect medium. The cliffs dictate model partitioning: Tensor Parallelism, which requires AllReduce after every layer, is strictly confined to the intra-node domain.

Domain	Interconnect	Bandwidth	Latency	Scaling Limit
Intra-Package	Silicon Interposer	~3.3 TB/s	<100 ns	Single Chip
Intra-Node	NVLink/ICI	~900 GB/s	~1 µs	Node (8–16 Chips)
Intra-Node (IO)	PCIe Gen5 x16	~64 GB/s	~2 µs	CPU-GPU, NIC-GPU
Inter-Node	InfiniBand NDR	~50 GB/s	~5–10 µs	Pod (Thousands)

The concentric view in figure 2.12 presents the same hierarchy spatially, so each outward boundary crossing reads as both a bandwidth drop and a latency increase.

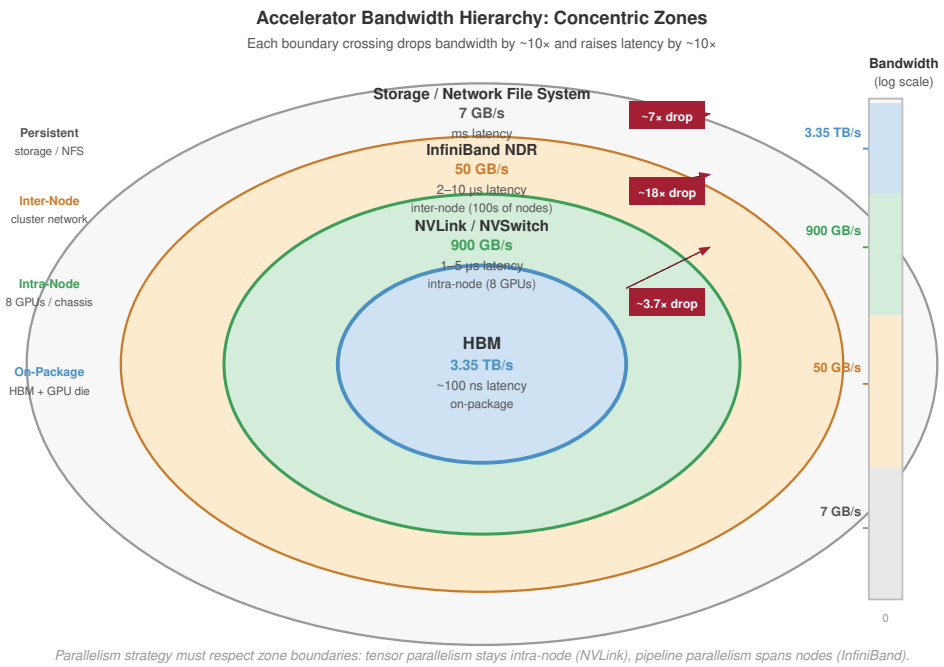


Figure 2.12: The Infrastructure Bandwidth Hierarchy: Concentric zones visualize the bandwidth tiers an ML workload crosses. HBM (3.35 TB/s) sits at the innermost zone; each outward boundary crossing drops bandwidth ~10× and raises latency ~10×: NVLink/NVSwitch (900 GB/s), InfiniBand NDR (50 GB/s), and Storage/Network File System (7 GB/s) at the outermost zone. Exploiting this hierarchy through tiered algorithms is the primary task of distributed systems engineering.

As table 2.9 shows, parallelism strategies must respect these boundaries. A simple synchronization calculation makes the bandwidth gaps concrete.

Napkin Math 2.4: The physics of the staircase

Problem: A 10 GB buffer is synchronized across the fleet. How much does the physical "Layer" of the fleet affect transfer time?

Math: Transfer time is $T_{\text{transfer}} = D_{\text{vol}}/\text{BW}$.

1. **HBM (intra-chip)**: 10 GB / 3,350 GB/s $\approx$ 3.0 ms.
2. **NVLink (intra-node)**: 10 GB / 450 GB/s $\approx$ 22.2 ms.
3. **InfiniBand (inter-node)**: 10 GB / 50 GB/s $\approx$ 200 ms.

Systems insight: Moving data across the cluster is 9× slower than moving it within a node. This "Bandwidth Staircase" is the primary driver of all parallelization strategies: we use tensor parallelism where bandwidth is abundant (NVLink) and data parallelism where it is scarce (InfiniBand). A model that "fits" in memory but ignores the staircase will spend 90 percent of its time waiting for the network.

Each cliff in the bandwidth staircase exists for a different physical reason, rooted in the signal-propagation regime at its distance scale. The **intra-package** interconnect achieves terabytes per second because signals travel through lithographically patterned copper traces on a silicon interposer, with path lengths measured in millimeters. At these distances, signal attenuation is negligible, no amplification is needed, and the signaling rate is limited only by the trace geometry and the transceiver design.

NVLink achieves hundreds of gigabytes per second using high-speed SerDes[12] transceivers over short copper cables or traces within a chassis, with path lengths of tens of centimeters. At these distances, signal attenuation is measurable but manageable with simple equalization circuits. The SerDes transceivers use PAM-4 (4-level pulse amplitude modulation) signaling at 112 Gb/s per lane, packing 2 bits per symbol to double the data rate relative to NRZ (nonreturn-to-zero) signaling.

PCIe uses a standardized protocol with flow control overhead, reducing effective bandwidth. While PCIe Gen5 uses the same 32 GT/s signaling rate as NVLink's individual lanes, the protocol overhead (packet headers, flow control credits, error checking) consumes approximately 20 percent of the raw bandwidth. The standardization that makes PCIe universally compatible also makes it less efficient than proprietary interconnects.

InfiniBand crosses meters of cable between racks, requiring signal amplification, error correction, and switch hops that add both latency and protocol overhead. Active optical cables (AOCs) convert electrical signals to light at the transmitter, propagate through optical fiber, and convert back to electrical signals at the receiver. Each electro-optical conversion adds approximately 2–5 nanoseconds of latency and consumes power for the laser driver and photodetector. Switch hops add another 100–300 nanoseconds each for packet routing and buffering.

As table 2.9 illustrates, the practical consequence is that parallelism strategies must respect these boundaries. Tensor parallelism (TP), which splits individual matrix multiplications across accelerators and requires an AllReduce operation after every layer, generates communication volume proportional to the model's hidden dimension hundreds of times per second. At NVLink bandwidth, this synchronization takes roughly 1 ms per layer. At InfiniBand bandwidth, the same synchronization takes 10–20 ms, which would leave the accelerators idle for the majority of each training step.

TP is therefore confined to within a single node, while data parallelism (which synchronizes gradients only once per training step) spans the inter-node network. The bandwidth hierarchy is the physical law that determines the topology of distributed training.

The hierarchy also explains why pipeline parallelism occupies an intermediate position in the bandwidth requirements. In pipeline parallelism, the model is divided into sequential stages, with each stage assigned to a different group of accelerators. The communication between stages consists of activations flowing forward during the forward pass and gradients flowing backward during the backward pass, with a volume proportional to the batch size times the hidden dimension.

For our 175B model with hidden dimension 12,288 and a microbatch of 4 sequences at 2,048, the activation tensor at each stage boundary is approximately $4 \times 2,048 \times 12,288 \times 2B$ bytes (FP16) $\approx$

[12] **SerDes (Serializer/Deserializer)**: From Latin *serialis* (in a row) and *de-* (reverse). A circuit pair that converts parallel data to a serial bit stream for transmission and back again. Modern high-speed links use PAM-4 signaling (4-level pulse amplitude modulation, encoding 2 bits per symbol) at very high lane rates, and the transceiver energy becomes a material part of node power. This is why dense interconnects such as NVLink, InfiniBand, and PCIe must be treated as power and cooling loads, not just as bandwidth specifications.

201 MB. This is far smaller than the 350 GB gradient AllReduce required by data parallelism, which is why pipeline parallelism places less demanding requirements on the inter-node network.

Activation transfers occur once per microbatch per stage boundary, far less frequently than tensor parallelism's per-layer AllReduce. Pipeline parallelism can therefore tolerate the lower bandwidth of inter-node links, making it the preferred strategy for spanning multiple nodes when the model's depth exceeds a single node's capacity.

The result is a natural mapping between parallelism types and interconnect domains: tensor parallelism within the node (NVLink), pipeline parallelism across nearby nodes (InfiniBand), and data parallelism across the full cluster (InfiniBand with gradient compression). This mapping is so fundamental that it has become a de facto standard in production-scale Transformer training systems, with the details differing primarily in the number of pipeline stages, the degree of tensor parallelism, and the degree of data parallelism.

The mapping also determines how the job scheduler assigns nodes to training jobs. Nodes within the same tensor-parallel group should be physically adjacent (ideally in the same chassis, connected via NVLink). Nodes within the same pipeline-parallel group should be in the same rack or adjacent racks (minimizing InfiniBand switch hops). Data-parallel groups can span the entire cluster because their communication (gradient AllReduce) is the least bandwidth-intensive and most latency-tolerant of the three parallelism types. One intra-node boundary remains before that mapping reaches the scheduler: the PCIe hierarchy that connects accelerators to host CPUs, NICs, storage, and DRAM.

2.4.1.1 The PCIe hierarchy within a node

While NVLink provides a high-bandwidth freeway for GPU-to-GPU communication, **PCIe Gen5** serves as the universal glue connecting the heterogeneous components of the node. It links GPUs to the host CPU, GPUs to InfiniBand Host Channel Adapters (HCAs), the host CPU to NVMe storage, and the host CPU to system DRAM. Understanding this topology is critical because PCIe bandwidth – though substantial at ~64 GB/s per direction (128 GB/s bidirectional) per x16 link – is often the bottleneck for operations that cross the accelerator boundary, such as data loading, checkpointing, and inter-node gradient synchronization.

A standard DGX H100 node features two host CPUs, each managing a **PCIe root complex** with 128 lanes. These lanes are distributed to maximize simultaneous throughput: 8 GPUs each receive a dedicated x16 link, and 8 InfiniBand HCAs each receive their own x16 link. This configuration provides an aggregate theoretical bandwidth of nearly 2 TB/s within the chassis. However, this bandwidth is shared among competing traffic streams. During a single training step, the PCIe bus simultaneously carries host-to-GPU data batches, GPU-to-HCA gradient shards for inter-node AllReduce (via GPUDirect RDMA), and periodic CPU-to-NVMe checkpoint writes. Contention between these streams can cause unexpected pipeline stalls, particularly when gradient synchronization saturates the links typically used for data loading.

The dual-CPU architecture introduces significant **Non-Uniform Memory Access (NUMA)** effects. The 8 GPUs are physically partitioned, with 4 connected to CPU 0's root complex and 4 to CPU 1's. A GPU connected to CPU 0 can access CPU 0's DRAM at full PCIe bandwidth but must traverse the inter-processor interconnect (UPI or Infinity Fabric) to access CPU 1's DRAM. This traversal incurs additional latency and reduces effective bandwidth by up to 50 percent. Robust data loading pipelines must employ *NUMA-aware scheduling*, pinning data loader worker processes to the CPU cores physically closest to the target GPUs. Empirical benchmarks show that aligning data loaders with the correct NUMA domain can improve data ingestion throughput by 20–30 percent, preventing the CPU from becoming the bottleneck in high-throughput training runs.

The practical consequence is that placement is part of the parallelism strategy: a scheduler that ignores PCIe roots, NIC locality, and NUMA domains can turn an otherwise valid tensor- or pipeline-parallel plan into a host-bound bottleneck. Chapter 5 formalizes how these intra-node and inter-node boundaries interact with data, tensor, and pipeline parallelism.

2.4.2 Dense node designs

Given the bandwidth hierarchy, the engineering challenge within a node is clear: connect 8 accelerators with enough bandwidth that they can function as a single logical device for tensor-parallel operations. The solution is a specialized switching fabric that provides full bisection bandwidth

between all accelerator pairs. Understanding the design of this fabric requires appreciating why simpler alternatives fail.

The simplest approach would be to connect the 8 GPUs in a ring, where each GPU has links to its two neighbors. A ring is inexpensive (requiring only N links for N GPUs) and works well for ring-AllReduce, where data flows sequentially around the ring in a circular pattern. The ring-AllReduce algorithm achieves optimal bandwidth utilization for gradient synchronization because each GPU simultaneously sends data to its right neighbor and receives data from its left neighbor, keeping all links busy throughout the operation.

However, tensor parallelism requires *all-to-all* communication: every GPU must exchange partial results with every other GPU after each layer. This is a fundamentally different communication pattern from the sequential flow of ring-AllReduce, and the ring topology handles it poorly. In a ring of 8 GPUs, communicating between opposite sides requires 4 hops, reducing effective bandwidth by $4\times$ compared to a direct link. A fully connected mesh, where every GPU has a direct link to every other GPU, eliminates this problem but requires $N(N-1)/2 = 28$ links for 8 GPUs, a wiring challenge that becomes impractical as the link count grows quadratically. The NVSwitch crossbar provides the best of both worlds: full bisection bandwidth with a manageable number of switch chips.

Consider the NVIDIA DGX H100 architecture. Eight H100 GPUs sit on a single baseboard, each connected to four NVSwitch chips via 18 NVLink 4.0 lanes. The NVSwitch chips form a non-blocking crossbar[13]: any GPU can communicate with any other GPU at the full 900 GB/s bidirectional bandwidth simultaneously, without contention. This is the equivalent of giving every pair of GPUs their own private highway, rather than forcing them to share a single road.

13 **Crossbar Switch**: A topology providing a dedicated path between every input-output pair simultaneously. An $N \times N$ crossbar has N^2 crosspoints, achieving full bisection bandwidth. NVSwitch implements a 3-stage crossbar using four switch chips (each with 64 NVLink ports), consuming ~800 W total – more than an entire V100 GPU – but this power cost is justified because it enables all-to-all tensor parallelism within a node, the communication pattern that would otherwise dominate training time.

The result is that the 8 GPUs within a DGX H100 can be treated as a single logical device with 640 GB of aggregate HBM and a combined compute throughput of 8×1979 TFLOP/s. This abstraction is critical for software: the training framework can partition a model across the 8 GPUs using tensor parallelism as if they were a single larger processor, with the NVSwitch fabric making the partitioning nearly transparent from a performance perspective.

The physical layout of the DGX H100 baseboard reflects this design goal. The 8 GPUs are arranged on the baseboard with the 4 NVSwitch chips positioned centrally. Each GPU connects to all 4 NVSwitch chips via NVLink lanes, and each NVSwitch chip has 64 NVLink ports that cross-connect the GPUs.

The NVSwitch chips themselves consume approximately 200 W each (800 W total for the NVSwitch fabric), a nontrivial power overhead that is justified by the bisection bandwidth it provides. To put this in perspective, the NVSwitch fabric alone consumes more power than an entire previous-generation GPU (the V100 drew 300 W). Without NVSwitch, achieving full all-to-all connectivity would require 28 direct GPU-to-GPU links, a wiring nightmare that would also consume more total NVLink lane capacity.

During a tensor-parallel forward pass, each GPU holds one shard of each weight matrix and computes its portion of the matrix multiplication. For a Transformer layer with hidden dimension 12,288 (typical for a 175B model), each of the 8 GPUs holds a $12{,}288 \times 1{,}536$ slice of the weight matrix. After each GPU computes its partial result, an AllReduce operation sums the partial results across all 8 GPUs over NVLink.

The AllReduce for a single layer's output involves exchanging approximately $4 \times 2{,}048 \times 12{,}288 \times 2$ bytes (FP16) across the NVLink fabric. For a typical microbatch size of 4 sequences at 2,048 tokens, this is roughly 201 MB per AllReduce, which completes in about 0.2 ms at 900 GB/s.

Even with 96 layers (forward pass) and 96 layers (backward pass), the cumulative AllReduce time is about 42.9 ms per training step. This is possible because the NVSwitch crossbar allows all 8 GPUs to communicate simultaneously without contention, unlike a ring topology where data must traverse multiple hops sequentially.

The communication overhead is small relative to the compute time per layer, which is approximately 2–5 ms depending on the sequence length and batch size. The resulting tensor-parallel scaling efficiency within the node is typically 85–95 percent, meaning that using 8 GPUs achieves 6.8–7.6$\times$ the throughput of a single GPU.

The remaining 5–15 percent efficiency loss comes from two sources. First, the AllReduce communication itself, even at NVLink bandwidth, consumes a fraction of each layer's execution time.

Second, some operations in the Transformer (layer normalization, dropout, activation functions) are not tensor-parallelized because their computation is small relative to the matrix multiplications. These sequential operations must complete on every GPU before the next layer's tensor-parallel computation can begin, introducing small but cumulative idle time.

Example 2.3: The NVLink bandwidth surprise

Scenario: Early users of multi-GPU nodes often attempted tensor parallelism over PCIe, expecting the 64 GB/s bandwidth to suffice.

Diagnosis: Tensor parallelism requires AllReduce after *every* Transformer layer, not just *once* per training step. A model with 96 layers generates 96 AllReduce operations per forward pass and another 96 during the backward pass, totaling 192 synchronization events per step. At PCIe bandwidth, each AllReduce for a 12,288-dimensional hidden state takes approximately 3.1 ms, accumulating to 604 ms of pure communication overhead per step. With a 200 ms compute-step anchor, the accelerators spend roughly 75.1 percent of their time waiting for PCIe transfers. Switching to NVLink reduces each AllReduce to 0.2 ms, bringing total communication overhead to 42.9 ms and restoring accelerator utilization to over 82.3 percent.

Systems lesson: For tensor parallelism, the interconnect is the bottleneck, and the difference between PCIe and NVLink is not incremental; it is the difference between a functional and a nonfunctional system.

Beyond NVLink, the node also connects to the external network via InfiniBand Host Channel Adapters (HCAs) and to host storage via PCIe. The DGX H100 includes eight InfiniBand ConnectX-7 HCAs, one per GPU, enabling GPUDirect RDMA: the network adapter can read from and write to GPU HBM directly, without staging data through host DRAM or involving the host CPU in the data path.

GPUDirect RDMA is critical for inter-node gradient synchronization, where each GPU's gradient shard must be sent to its peers on other nodes. Without GPUDirect, each gradient transfer would require two extra copies: first from GPU HBM to host DRAM (over PCIe), and then from host DRAM to the network adapter (over another PCIe path). The double-copy adds latency (two PCIe traversals instead of one direct DMA operation) and halves the effective bandwidth (each byte traverses the PCIe bus twice).

GPUDirect RDMA eliminates both copies by allowing the InfiniBand HCA to read directly from GPU HBM, bypassing the host CPU and host DRAM entirely. The data path goes from GPU HBM through the NVLink-to-PCIe bridge to the InfiniBand HCA in a single transfer. The single-hop data path is one of the reasons that modern training clusters achieve inter-node communication bandwidth close to the raw InfiniBand line rate.

The host CPU manages job scheduling, data preprocessing, data loading from storage, and network protocol processing, but it sits outside the critical path of the GPU-to-GPU computation. The separation is deliberate: the host handles the "slow" operations (disk I/O, network management, job coordination) while the GPU-NVSwitch fabric handles the "fast" operations (matrix math, gradient synchronization).

The host CPU's role is analogous to an operating system kernel in a traditional computer: it manages resources, handles exceptions, and coordinates I/O, but it does not perform the application's core computation. In a well-optimized training loop, the host CPU is almost never on the critical path, and its performance specifications (core count, clock speed) are less important than its I/O capabilities (PCIe lane count, memory channels, NVMe controller bandwidth).

The host CPU also runs the training framework's Python runtime, which orchestrates kernel launches, manages the computation graph, and coordinates collective communication operations. In well-optimized training loops, the host CPU is pipelining the next batch's data loading and preprocessing while the GPUs are executing the current batch's forward and backward passes, keeping all components busy simultaneously.

As the "data center tax" on host CPUs has grown – with up to 30 percent of CPU cycles consumed by network protocol processing, storage virtualization, and security functions – ML nodes can incorporate **Data Processing Units (DPUs)** or SmartNICs. Devices like the NVIDIA BlueField DPU

offload these infrastructure tasks to dedicated ARM cores and hardware accelerators integrated into the network adapter itself. By moving the control plane for RDMA, firewall rules, and storage protocols to the DPU, the host CPU recovers cycles for the ML pipeline: data loading, tokenization, and kernel orchestration. The DPU also acts as an isolated security domain, allowing cloud providers to maintain control over the network and storage layer via the DPU while granting customers full access to the host CPU and GPUs. For our 175B model training, DPU offload can keep RDMA protocol processing outside the host CPU critical path for gradient synchronization, which flows from GPU HBM toward the network fabric without host CPU involvement.

2.4.2.1 Alternative node architectures

These alternatives are accelerator-system designs, not interchangeable product names. An AMD MI300X is a GPU-class accelerator whose package emphasizes very large HBM capacity, while Intel/Habana Gaudi designs emphasize Ethernet/RDMA attachment as a first-class communication path. Comparing them against a DGX H100 node reveals where each architecture spends its scarce budget.

The DGX H100 answers a general node-design problem by making local accelerator exchange cheap. A switch-rich design spends power, board area, and cost on universal any-to-any exchange. That choice is easy to justify when tensor parallelism is the critical path, because each Transformer layer turns local accelerator communication into part of the model's inner loop. The 900 GB/s H100 NVLink fabric therefore buys a software abstraction: the training framework can treat eight accelerators as one tightly coupled device for the layers that must be split across chips.

Other systems move that boundary. TPU pods use the Inter-Chip Interconnect (ICI) and a mesh-oriented topology instead of making a central switch the defining component of the node; the TPU v5p profile in MLSysIM models ICI at 1,200 GB/s. The mesh can reduce switch cost and expose a regular structure to the compiler, but it also makes placement more visible: the runtime must respect neighbor relationships rather than assuming that every chip pair has the same communication path. That is a reasonable trade when the workload is regular enough for the compiler and collective libraries to map communication onto the topology.

The MI300X shifts the pressure point from local switching toward memory capacity. Its package exposes 192 GB of HBM at 5.3 TB/s, compared with 80 GB on an H100-class accelerator. An eight-accelerator MI300X node would therefore provide 1,536 GB of aggregate HBM, enough to hold the 350 GB FP16 weight tensor for our 175B running model. That does not eliminate the optimizer, gradient, activation, and fragmentation problem introduced in section 2.4, but it changes how often the system must spill state into slower memory tiers and how much microbatch headroom remains before offload becomes mandatory.

Gaudi takes a different route by integrating RDMA-capable Ethernet into the accelerator and using a more network-centric design for communication. The Gaudi 2 profile exposes 300 GB/s of aggregate RoCE bandwidth through 24 attached 100 GbE ports, while an H100-class NVLink node offers 900 GB/s of local GPU exchange and the newer Gaudi 3-class accelerator still exposes 128 GB of HBM. The question is whether simplifying the fabric and using Ethernet consistently across node and pod is worth giving up the very high local exchange bandwidth of a switch-rich NVLink design. That trade can be attractive when data parallelism, inference serving, or cluster cost dominates, and less attractive when tensor-parallel collectives sit inside every layer.

These alternatives are not a ranking of vendors. They are different placements of scarce resources: switching bandwidth in the node, regular topology in the pod, HBM capacity in the package, or network attachment on the accelerator. For the 175B model used throughout this chapter, the tensor-parallel portion of training makes local accelerator communication unusually important, so the DGX-style answer is natural. In other regimes, especially memory-heavy recommenders, topology-regular training, or cost-sensitive serving fleets, a different boundary can become the one that matters most. Software ecosystem maturity, supply availability, and total cost of ownership then decide whether the theoretical architectural fit becomes a deployable system.

2.4.2.2 Node health and reliability

The node is the fundamental failure domain in the fleet hierarchy. When a single GPU fails within a node, the entire node typically becomes unusable for the training job, because tensor parallelism

requires all 8 GPUs to participate in every AllReduce operation. A single missing GPU breaks the collective communication, stalling all other GPUs in the node. The *least reliable* component, not the *average*, therefore determines the node's effective MTBF.

The components most prone to failure within a node form a diagnostic list, roughly in order of frequency:

- **GPU memory**: HBM can encounter ECC-uncorrectable errors that remove the GPU from service.
- **NVLink connections**: Signal integrity can degrade over time, causing link retraining or communication errors.
- **Power supplies**: Capacitors age under sustained high-load operation.
- **Cooling components**: Pumps in liquid-cooled systems and fan bearings in air-cooled systems introduce mechanical failure modes.

A well-managed fleet tracks the error rates of each component and preemptively migrates workloads away from nodes showing early warning signs. The most important early warning indicators are increasing corrected ECC error rates (which often precede uncorrectable errors by hours to days), rising junction temperatures at constant workload (indicating cooling degradation), and intermittent NVLink retraining events (indicating cable or connector degradation). Proactive replacement of components showing these warning signs can prevent the much more costly outcome of an unplanned job failure during a multi-week training run.

Node-level health monitoring is therefore a critical operational practice. Modern fleet management systems continuously collect telemetry from each GPU (temperature, power draw, ECC error counts, NVLink error rates) and from the host BMC (baseboard management controller). Automated health checkers run short diagnostic workloads on idle nodes to verify that all GPUs, NVLinks, and InfiniBand connections are functioning correctly before the job scheduler assigns training work to that node. Without this proactive monitoring, a silently degraded node can corrupt training gradients (if the error is in the arithmetic path) or slow the entire job (if the error causes NVLink retraining, which temporarily reduces bandwidth). Chapter 7 discusses fleet-level fault tolerance strategies in detail.

For our 175B model training across 128 nodes, the probability of at least one node experiencing a hardware issue during a two-week training run is substantial. Under a composite node-level MTBF assumption of 1,000 hours, which includes GPUs, host components, power, cooling, and network links, the expected number of node failures during a run of 336 hours is approximately 43. This means the training run will experience, on average, roughly 3.1 failures/day. A GPU-only lower bound from the canonical per-GPU MTBF of 50,000 hours would predict only 6.9 GPU-driven node outages over the same run, before adding host, power, cooling, and network failure terms. Each failure requires detecting the degraded node, draining its workload, substituting a spare node, and resuming from the last checkpoint – a process that takes 10 minutes–30 minutes with automated tooling. With automated recovery, the direct restart overhead is roughly 2.1 percent–6.4 percent of the run; without spare nodes and automated recovery, multi-hour interventions can push overhead into the 20–30 percent range, consuming millions of dollars in wasted GPU-hours.

2.4.3 Node memory partitioning

A common misconception is that a model's memory footprint equals its weight storage. In practice, training state dwarfs the weights: Adam optimizer moments, gradients, and activations collectively consume 5–7$\times$ more memory than the parameters alone, and these components reside in different tiers of the node's memory hierarchy.

As table 2.10 shows, for our 175B model, the training memory breaks down as follows:

Table 2.10: Training Memory Breakdown for a 175B-Parameter Frontier Model: The optimizer state alone (first and second moments in FP32) consumes 4× the memory of the model weights, making optimizer memory the dominant component of training memory. This is why memory optimization techniques focus heavily on optimizer state sharding and offloading.

Component	Precision	Memory per Parameter	Total for 175B Model
Model Weights	FP16	2 bytes	350 GB
Gradients	FP16	2 bytes	350 GB
Optimizer (Adam m)	FP32	4 bytes	700 GB
Optimizer (Adam v)	FP32	4 bytes	700 GB
Activations	Mixed	Variable	100–400 GB
Total			**2.2–2.5 TB**

Table 2.10 makes the partitioning concrete. The same 2.2–2.5 TB budget established earlier reappears here, but now decomposed by component and tier: the Adam optimizer state in FP32, not the FP16 weights, is the dominant term, and the activation peak (which scales with batch size and sequence length) is the most variable. The lesson for parallelism selection is that the weights are the smallest piece of what the node must hold, so a strategy that only shards weights leaves the largest consumers, optimizer state and activations, untouched.

A DGX H100 node provides 640 GB of aggregate HBM across its 8 GPUs, plus 2 TB of host DDR5 DRAM. The HBM capacity alone is insufficient for the full training state. To appreciate why, consider the memory arithmetic for different parallelism strategies. With **pure data parallelism**, each GPU must hold the complete model: 350 GB weights + 350 GB gradients + 1,400 GB optimizer states = 2,100 GB – physically impossible on an 80 GB device. Even **ZeRO Stage 3**, which shards all three components across only 8 GPUs, yields 2,100 GB / 8 ≈ 262.5 GB per GPU, still more than triple the available HBM. The solution is to combine tensor parallelism (which splits the model's layers across GPUs, reducing per-GPU weights to 350 GB / 8 = 43.75 GB) with data-parallel sharding across many replicas. In a configuration with TP-8 within each node and 128-way data parallelism across the cluster, each GPU holds 43.75 GB of weight shards. The optimizer state is likewise partitioned, first by the 8-way tensor split and then across the 128 data-parallel replicas, so each GPU holds approximately 1,400 GB / (1024) ≈ 1.4 GB of optimizer state, or 10.9 GB per 8-GPU tensor-parallel group, before gradients, activations, buffers, and fragmentation. The weight-plus-optimizer subtotal of roughly 45.1 GB is only the starting point; the memory-budget exercise below shows why gradients and activations still require checkpointing and offload. This arithmetic is why tensor parallelism, optimizer sharding, activation checkpointing, and memory tiering are used together for large-model training.

ZeRO Stage 3 fully shards optimizer states, gradients, and parameters across all GPUs participating in data parallelism. With sharding across the 8 GPUs in a node, each GPU holds only 1/8 of each component, reducing per-GPU memory to roughly 262.5 GB of equivalent storage (2,100 GB total state ÷ 8). This still exceeds 80 GB per GPU, so the sharding must be combined with other techniques. Activation checkpointing stores only a subset of intermediate activations for the backward pass and recomputes the remaining activations from the nearest checkpoint. It trades roughly 33 percent more FLOPs for up to 10× lower activation memory.

CPU offloading stores optimizer states in host DDR5 DRAM and transfers them to GPU HBM only when the parameter update needs them. The PCIe Gen5 link at 64 GB/s introduces a transfer delay, but the overhead is manageable because the parameter update is a small fraction of total step time. NVMe offloading extends the same idea to SSDs for even larger models, with sequential read bandwidth of 5–7 GB/s and a larger latency penalty that requires careful pipelining to overlap with computation.

The node's memory hierarchy thus operates as a tiered system. HBM holds the active computation: weight shards that are needed for the current layer's matrix multiplication, the current microbatch's activations, and the gradients being accumulated during the backward pass. DDR5 holds the bulk optimizer state, which is accessed only during the parameter update step at the end of each training iteration. NVMe provides overflow capacity for the largest models, storing seldom-accessed shards that can be prefetched during computation.

The training framework's memory manager orchestrates the flow of data between these tiers, operating much like a hardware cache controller but at a coarser granularity. The analogy to hardware caching is instructive: just as a CPU's cache controller uses prefetching to hide memory latency by loading data before the processor needs it, the training framework's memory manager uses software-level prefetching to hide the PCIe transfer latency by loading weights before the GPU needs them. During the forward pass of layer ℓ, the manager simultaneously prefetches the weights for layer $\ell + 1$ from DDR5 to HBM (if they have been offloaded) and evicts the weights for layer $\ell - 2$ from HBM back to DDR5 (if memory is tight).

The pipelining ensures that each layer's computation can proceed without waiting for data transfers, at the cost of increased software complexity and careful tuning of the prefetch depth. If the prefetch is too shallow, the computation stalls waiting for data. If the prefetch is too aggressive, HBM fills up with data that will not be used for several layers, crowding out the activations and gradients needed for the current computation. The capacity-bandwidth trade-off across these tiers determines which data a training framework can keep "hot" and which it must prefetch; table 2.11 quantifies the three-order-of-magnitude bandwidth gap that the memory manager must bridge.

Table 2.11: Node Memory Hierarchy: Each tier trades capacity for bandwidth. The training framework's memory manager must orchestrate data flow across these tiers to fit models whose total state exceeds the aggregate HBM capacity.

Memory Tier	Capacity	Bandwidth	Latency	Role
GPU HBM3	80 GB × 8 = 640 GB	3.35 TB/s per GPU	<1 µs	Active computation
Host DDR5	2 TB	hundreds of GB/s aggregate	~100 ns	Optimizer state
NVMe SSD	8–30 TB	up to 7 GB/s per drive	~100 µs	Overflow, checkpoints

The orchestration is complex but essential: without it, training our 175B model on H100-class hardware would be impossible. Chapter 5 examines these memory optimization strategies and their interaction with parallelism in full detail.

2.4.3.1 Putting it together: Memory budget exercise

To solidify the memory planning concepts, consider a concrete sizing exercise for our 175B model on a DGX H100 node.

Example 2.4: Memory budget for 175B training on DGX H100

Given:

- Model: 175B parameters
- Node: 8× H100 GPUs, 80 GB HBM each (640 GB total HBM)
- Host: 2 TB DDR5
- Parallelism: 8-way tensor parallelism within the node

Step 1: Weight Distribution With 8-way TP, each GPU holds 1/8 of every weight tensor. Per-GPU weight memory: 43.75 GB (FP16).

Step 2: Optimizer State with ZeRO Stage 1 ZeRO Stage 1 shards the optimizer states across data-parallel workers. Within a single node using only TP (no data parallelism), the optimizer is not sharded. Each GPU stores the full optimizer state for its TP shard: 43.75 GB × 4 (FP32 m and v) = 175 GB.

The total exceeds the 80 GB HBM capacity. The fix is to offload optimizer states to host DDR5. The 2 TB of DDR5 can hold the full 1,400 GB of optimizer state with room to spare.

Step 3: Activations With activation checkpointing (recomputing every other layer), the activation memory for a microbatch of 4 sequences at 2,048 token length is approximately 15–25 GB per GPU. Without checkpointing, it would be 100–200 GB per GPU, which is impossible.

Step 4: Gradient Accumulation Gradients match the weight size: 43.75 GB per GPU in FP16.

Total per-GPU HBM usage:

- Weights: 43.75 GB
- Activations: ~20 GB (with checkpointing)
- Gradients: 43.75 GB
- Buffers and fragmentation: ~5 GB
- **Total**: ~113 GB (exceeds 80 GB HBM)

Resolution: Use gradient accumulation (accumulate over 2–4 microbatches before AllReduce, reducing peak activation memory) and gradient offloading (store inactive gradient shards in DDR5). With these techniques, the per-GPU HBM usage drops to approximately 70–75 GB, fitting within the 80 GB envelope with a small margin for NCCL communication buffers.

Systems insight: Model capacity is not the only memory constraint. Training fits only when weights, gradients, optimizer state, activations, communication buffers, and fragmentation are managed together.

The example illustrates why memory planning for large models is a careful engineering exercise rather than a simple capacity calculation. Beyond memory-tier placement, the bandwidth hierarchy imposes a sharp penalty on data that must cross the node boundary, a cost the next calculation makes explicit.

Napkin Math 2.5: The cost of crossing the cliff

Problem: A training step must synchronize a 1 GB gradient buffer. How much longer does the synchronization take when it crosses the node boundary (InfiniBand) compared to staying within the node (NVLink)?

1. **Intra-Node (NVLink)**: Bandwidth = 450 GB/s one-way effective bandwidth; the bidirectional aggregate is 900 GB/s, but this transfer-time calculation moves one buffer in one direction. $T_{\text{intra}} = 1 \text{ GB} / 450 \text{ GB/s} \approx \mathbf{2.2}$ ms
2. **Inter-Node (InfiniBand NDR)**: Bandwidth = 50 GB/s. $T_{\text{inter}} = 1 \text{ GB} / 50 \text{ GB/s} \approx \mathbf{20}$ ms

Systems insight: Crossing the node boundary increases communication time by approximately 9×. In a training loop where gradient synchronization happens at every step, this penalty would stall the accelerators for the majority of each iteration, reducing utilization to well below 50 percent. This is why training frameworks use tensor parallelism within the node (fast NVLink) and data parallelism across nodes (tolerant of slower InfiniBand).

2.4.4 Data loading and I/O pipeline

The discussion so far has focused on how data moves within the node (HBM to Tensor Core, GPU to GPU via NVLink, GPU to host via PCIe). Training also requires feeding the node with a continuous stream of training data from external storage. If the data loading pipeline cannot keep up with the GPU's consumption rate, the most expensive component in the system (the GPU) sits idle waiting for data. This is the **I/O bottleneck**, and avoiding it requires careful engineering of the data loading pipeline.

For language model training, the training throughput determines the data consumption rate. If the cluster processes 500,000 tokens per second and each token requires 2 bytes of input data (token ID), the raw data ingestion rate is only 1 MB/s, which is trivially handled by any storage system.

However, the actual I/O demand is much larger because training data must be tokenized, shuffled, batched, and transferred to GPU memory. A realistic data pipeline turns raw bytes into GPU-ready batches through five ordered stages:

1. **Reading**: Raw text is read from a distributed file system (NFS, Lustre, or cloud object storage like S3) into host memory. For large datasets (terabytes of text), the data is typically stored in a binary format (TFRecord, WebDataset, or memory-mapped files) to minimize parsing overhead.
2. **Preprocessing**: Tokenization, sequence packing, and random cropping are performed on the host CPU. These operations are parallelized across multiple CPU cores using data loader workers (typically 4–8 workers per GPU).
3. **Batching**: Individual sequences are assembled into microbatches, padded to uniform length, and organized into tensors.
4. **Transfer**: The assembled batch is transferred from host DRAM to GPU HBM over PCIe. For tokenized language modeling, this transfer is usually small because token IDs and masks are compact; multimodal batches or pipelines that materialize FP16 embeddings on the host can reach 100–200 MB and take milliseconds at PCIe Gen5 bandwidth.
5. **Prefetching**: While the GPU processes the current batch, the data loader prefetches and preprocesses the next 2–4 batches in parallel, ensuring that the next batch is already in GPU HBM when the current computation completes.

The critical design goal is to make the data pipeline invisible to the GPU: the prefetching must be deep enough that a batch is always ready when the GPU needs it, regardless of transient variations in storage bandwidth or preprocessing time. When this pipeline is well-tuned, the GPU utilization is limited only by compute and communication, not by data loading. When it is poorly tuned, the GPU can spend 10–30 percent of its time waiting for data, which directly reduces training throughput.

The traditional data path routes every byte through the host CPU: storage controller to host DRAM (one PCIe traversal), then host DRAM to GPU HBM (a second PCIe traversal). This "bounce buffer" architecture doubles the traffic on the PCIe root complex and makes the CPU a bottleneck for I/O-intensive workloads, capping effective bandwidth at 3–4 GB/s per NVMe drive due to kernel context switching and interrupt handling overhead. **GPUDirect Storage (GDS)** eliminates this inefficiency by establishing a direct DMA path between the NVMe controller and GPU memory, bypassing the host CPU entirely. A single NVMe drive can deliver 5–7 GB/s directly to GPU HBM through GDS, saturating the drive's internal bandwidth rather than the host's I/O subsystem. For a standard node equipped with four NVMe drives, GDS unlocks an aggregate storage-to-GPU bandwidth of 20–28 GB/s, freeing the CPU to focus exclusively on preprocessing tasks like tokenization and sequence packing. The storage chapter's GPUDirect Storage discussion in section 4.5 examines this direct path in detail.

The required prefetch depth is not arbitrary but follows from the statistics of I/O latency variance. If the GPU processes one batch every T_{compute} seconds and the storage system delivers a batch every $T_{\text{I/O}}$ seconds with standard deviation $\sigma_{\text{I/O}}$, the minimum prefetch depth k required to maintain a 99.7 percent probability of zero stalls is $k \geq \lceil T_{\text{I/O}}/T_{\text{compute}} + 3\sigma_{\text{I/O}}/T_{\text{compute}} \rceil$. For our 175B model training run where $T_{\text{compute}} = 2.0$ seconds and the storage layer delivers batches at $T_{\text{I/O}} = 1.5 \pm 0.5$ seconds, the required depth is $\lceil 0.75 + 0.75 \rceil = 2$ batches. In production environments where "noisy neighbors" on shared file systems induce heavy tail latencies, engineers typically over-provision this buffer to 4–8 batches to insulate the GPU from the erratic physics of distributed storage. The memory cost of this prefetch buffer is negligible compared to the cost of a GPU stall for tokenized language workloads, though multimodal pipelines must budget larger host-side batch buffers.

As cluster sizes expand, a new storage bottleneck emerges. When training our 175B model across 128 nodes, the system acts as a synchronized "thundering herd" – all 128 nodes simultaneously demand the next microbatch of tokens at the start of every training step. A shared parallel filesystem like Lustre, even with 500 GB/s of aggregate throughput, will buckle when 128 clients simultaneously pull data, causing read latencies to spike from milliseconds to seconds. The architectural solution is **tiered storage** with aggressive local caching. By provisioning each training node with local NVMe SSDs capable of delivering 25 GB/s per node, the cluster decouples its immediate data dependency from the shared filesystem. A background process prefetches data from the central store to the

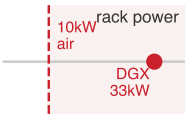
local NVMe cache asynchronously, smoothing out I/O spikes. For the 128-node cluster, local NVMe creates an aggregate read bandwidth of 3.2 TB/s (128×25 GB/s), eclipsing the capability of even the most expensive centralized storage arrays and ensuring the GPUs are never starved.

> **Systems Perspective 2.5: The data loading trap**
>
> A subtle failure mode occurs when data loading appears fast in benchmarks but degrades under production conditions. Single-node benchmarks often read data from local NVMe SSDs, achieving 5–7 GB/s read bandwidth. In production, where data resides on a shared storage system accessed by hundreds of nodes simultaneously, the effective per-node bandwidth may drop to 100–500 MB/s due to network congestion and storage controller contention. Teams that design their data pipelines based on single-node benchmarks may discover that their GPUs are I/O-starved in production. Chapter 4 examines the storage architecture strategies that address this bottleneck, including the tiered storage hierarchies that ensure training data reaches GPUs without stalling.

For our 175B model, a single DGX H100 node provides 640 GB of aggregate HBM, enough to hold the FP16 model weights but not the full Adam training state without sharding and offload. Training also requires processing trillions of tokens, and a single node's compute throughput limits training time to months. To complete training in a reasonable timeframe (weeks rather than months), we need tens or hundreds of nodes. Stacking those nodes into a physical enclosure brings us to the next level of infrastructure, where the constraints shift from bandwidth and capacity to raw power and heat.

2.5 The Rack

A standard 42U server rack in a traditional data center draws 5–10 kW and can be cooled by room-temperature air pushed through perforated floor tiles. Now place four DGX H100 nodes in that same rack: 32 GPUs, each drawing 700 W, plus host CPUs, memory, networking, power conversion losses, and cooling overhead. The rack power reaches 33.5 kW, an order of magnitude beyond what traditional data center infrastructure was designed to deliver or cool. At this density, the engineering constraints shift from silicon and signal integrity to power delivery and thermodynamics. The rack is where the power wall and the laws of heat transfer become the dominant design forces.



Modern GPU racks cross the air-cooling envelope.

For our 175B model, training across 1,024 GPUs requires 32 racks (4 nodes of 8 GPUs each per rack). Each rack dissipates 33.5 kW as heat, the thermal output of a small industrial furnace. The aggregate facility-relevant power draw of the training cluster is approximately 1.1 MW, enough to power several hundred homes. Delivering this power reliably, converting it efficiently, and removing the resulting heat without allowing any component to exceed its thermal limit is a multi-disciplinary engineering challenge that spans electrical, mechanical, and civil engineering. A failure at any point in the power delivery chain, from the utility substation to the individual GPU voltage regulator, can halt the entire training run, wasting hours of computation and potentially corrupting the training state.

> **Definition 2.6: Rack**
>
> ***Rack*** is the physical infrastructure unit, a standardized 42U enclosure, that houses multiple compute nodes, a Top-of-Rack (ToR) switch (the first network aggregation point connecting all nodes in the rack to the broader cluster fabric), power distribution units, and cooling distribution manifolds, defining the granularity at which power and cooling capacity must be provisioned.
>
> 1. **Significance**: AI rack power density has grown dramatically: a rack of 4 DGX H100 nodes contains 32 GPUs at 700 W each, totaling 22.4 kW of GPU power alone, plus 11+ kW for CPUs, NVSwitch, NICs, power conversion losses, and cooling overhead, reaching 33–40 kW per rack. This far exceeds the 5–10 kW typical of general-purpose data center

racks, requiring dedicated liquid cooling infrastructure and electrical circuits that must be co-designed with the building.

2. **Distinction**: Unlike a node (the compute unit), a rack is the infrastructure unit: it is the physical level at which cooling manifolds, power distribution, and the ToR switch are integrated—making the rack the minimum replaceable and serviceable unit for facility operations, not the node.
3. **Common pitfall**: A frequent misconception is that rack placement is a logistical afterthought. Nodes within the same rack share a ToR switch and have single-hop connectivity; nodes in different racks cross at least two switch hops. Placing all pipeline-parallel stages of a training job within the same rack minimizes inter-stage latency and reduces inter-rack fabric load.

The rack opener promised that the laws of heat transfer become the dominant design force at this level, and the reason is a hard physical limit on how fast a moving fluid can carry heat away from a surface. Air is a poor coolant: it has low density and low heat capacity, so a given volume of air absorbs little energy before its temperature rises to the point where it can no longer cool the silicon. Forced-air cooling, even with hot-aisle/cold-aisle containment and high-static-pressure fans, removes heat at a rate set by how much air the facility can physically push through the rack. That rate tops out around 30 kW per rack. Below roughly 10 kW, room-temperature air through perforated floor tiles suffices, which is why traditional 42U racks were never a thermal problem. Engineered air cooling stretches the envelope toward 30 kW, but no amount of additional fan power moves past it: beyond that density the air leaving the rack is already too hot to absorb more heat, and adding fans consumes power and produces noise without removing additional watts. A rack of four DGX H100 nodes sits at 33–40 kW, already past the air ceiling, and Blackwell-class racks push toward 100 kW and beyond, the point at which air cooling fails outright. Air is not throttled at these densities; it is disqualified.

This thermal ceiling is what forces the rack from air to liquid: above the air-cooling threshold the only way to remove the heat is to put a far denser coolant in direct contact with the silicon, which is why high-density AI racks are liquid-cooled by physical necessity rather than by choice. Section 15.4.6 takes up the cooling architectures that meet this requirement, the efficiency comparison between air and liquid cooling, and the facility-power overhead the choice carries every hour the cluster runs.

2.5.1 Rack design considerations

The physical design of an ML rack differs substantially from a traditional server rack in ways that go beyond power and cooling. In a conventional 42U rack, servers are installed as independent 1U or 2U pizza-box units, each with its own fans, power supplies, and cable connections. ML racks use a fundamentally different form factor. A DGX H100 occupies 8U and contains 8 GPUs, the NVSwitch fabric, multiple power supplies, and (in liquid-cooled configurations) coolant manifolds with quick-connect fittings. Four DGX units in a 42U rack leave only 10U for networking switches, cable management, and PDUs.

The transition from traditional server form factors to ML-optimized designs is driven by the same power density challenge that forced the transition from air cooling to liquid cooling. A traditional 1U server dissipates 300–500 W and requires only modest airflow for cooling. A DGX H100 node dissipates approximately 8.4 kW across its 8 GPUs, NVSwitches, and supporting components, requiring either massive airflow (in air-cooled configurations) or liquid cooling plumbing (in liquid-cooled configurations). The form factor must accommodate both the compute hardware and the thermal management infrastructure, which is why ML nodes are substantially taller (8U vs. 1U) than traditional servers.

Cable management is a nontrivial engineering challenge. Each DGX H100 node has 8 InfiniBand cables (one per GPU), 2 Ethernet management cables, power cables, and (for liquid-cooled units) coolant hoses. A fully populated rack has over 40 high-speed cables, each of which must be routed carefully.

Three physical constraints govern cable routing:

- **Airflow clearance**: Cables must not obstruct airflow in air-cooled designs, because even a partial blockage can create hot spots that throttle nearby components.
- **Bend radius**: High-speed cables have a minimum bend radius, typically 10–15$\times$ the cable diameter for active optical cables, below which the signal path is damaged and can produce bit errors or complete link failure.
- **Electromagnetic interference**: Bundling too many high-speed copper cables together can degrade signal quality on neighboring links.

In large installations, the cable plant alone can take weeks to install and test, and cable routing errors are one of the most common causes of postinstallation debugging delays.

As rack power densities climb toward 100 kW for liquid-cooled clusters, the inefficiency of per-server AC conversion becomes untenable. Traditional designs dedicate volume in every chassis to redundant AC power supply units (PSUs), while many dense-rack designs use **rack-level DC power distribution**. In this architecture, a centralized power shelf converts mains AC to a **48V DC busbar** that runs the height of the rack, replacing individual server PSUs. This consolidation eliminates one conversion stage, yielding a system-wide efficiency gain of 2–3 percent – a meaningful reduction in thermal load when training a 175B-parameter model across thousands of GPUs. Power is delivered via the 48V bus directly to the server backplane, where compact DC-to-DC converters step it down to 12V and finally to the GPU's sub-1V operating voltage. Pioneered by Google and Meta through the Open Compute Project (OCP), this topology reduces failure points and recovers valuable chassis volume for hydraulic cooling loops and high-bandwidth interconnects.

The **Top-of-Rack (ToR) switch** deserves special attention. In traditional data centers, a ToR switch provides 1–10 Gb/s Ethernet connectivity to each server, aggregated into uplinks to higher-level switches. In an ML rack, the ToR switch is an InfiniBand or high-speed Ethernet switch providing 400 Gb/s per port, and it must handle the bursty, synchronized traffic patterns of distributed training.

The placement and configuration of ToR switches directly affects the network topology. Rail-optimized designs (used by Meta and others) assign each GPU within a node to a specific "rail" that connects to a dedicated ToR switch. In a conventional design, all 8 GPUs in a node connect to the same ToR switch, creating a potential bottleneck when multiple GPUs simultaneously send AllReduce traffic. In a rail-optimized design, GPU 0 in every node connects to ToR switch 0, GPU 1 to ToR switch 1, and so on. This ensures that AllReduce traffic within a parallelism group (where all participants are the same GPU index across different nodes) is confined to a single switch rather than traversing the full fabric.

The rail-optimized design requires 8 ToR switches per rack group instead of 1, which increases switch cost and cabling complexity. However, it eliminates cross-switch congestion for the most common communication pattern (data-parallel AllReduce across nodes), improving scaling efficiency by 5–15 percent compared to a conventional single-ToR design. Chapter 3 examines rail-optimized topologies in detail, including the formal analysis of their bandwidth properties and the conditions under which they outperform fat-tree alternatives.

2.5.1.1 Facility reliability and environmental controls

Rack design also defines physical failure domains. Data centers housing ML infrastructure implement multiple layers of physical security and environmental monitoring, not only for regulatory compliance but also because a single rack of DGX H100 nodes ($1.4 million in hardware alone) concentrates enough value, heat, and liquid-cooling risk to justify controls that would be excessive for traditional server equipment.

Environmental monitoring is therefore part of the compute design rather than a facilities afterthought. Water detection sensors beneath raised floor tiles and around liquid cooling piping can identify leaks within seconds and trigger automatic coolant isolation valves. Aspirating smoke detection systems continuously sample air from inside racks, detecting combustion byproducts at concentrations far below what sprinkler-activating detectors can sense. Vibration sensors on the building structure and rack frames catch seismic events or construction-induced vibration that could damage disk drives or loosen cable connections. Humidity control keeps relative humidity between 40–60 percent to prevent both static discharge and condensation.

These environmental controls are not afterthoughts but integral parts of the infrastructure design that affect reliability and uptime. A water leak that reaches a DGX H100 baseboard can cause

millions of dollars in damage and weeks of downtime. A fire in a single rack can shut down an entire data center hall for days while the fire suppression system is recharged and the affected area is decontaminated. The 2021 OVHcloud fire in Strasbourg shows how a single-building incident, absent the right compartmentalization, escalates into a site-wide failure domain.

War Story 2.1: The data center that became the failure domain

Context: OVHcloud operated four data centers at its Strasbourg site, serving customers whose systems depended on the site's physical power, fire, and recovery design (OVHcloud 2021).

Failure mode: On March 10, 2021, a fire broke out in SBG2. OVHcloud reported that SBG2 was destroyed, SBG1 was severely damaged, and other Strasbourg data centers were powered down even when not directly burned.

Consequence: The incident forced service interruptions and data-recovery work across a site that many customers had treated as stable infrastructure rather than as a shared physical failure domain.

Systems lesson: Compute infrastructure is not only GPUs and network topology. Fire compartments, power isolation, backup geography, and disaster-recovery assumptions determine whether a rack-scale incident becomes a site-scale outage.

Once rack-level controls are in place, the reliability question moves up to facility redundancy. The same failure-domain logic that isolates a leaking rack also decides how much duplicate power, cooling, and recovery capacity the site should buy.

Data center reliability is commonly classified using the Uptime Institute Tier system, with **Tier III** and **Tier IV** representing common options for mission-critical compute (Uptime Institute 2026). A Tier III facility is concurrently maintainable: redundant capacity components and distribution paths let operators maintain or replace equipment without shutting down IT operations. Tier IV adds fault tolerance through independent, physically isolated systems, so a single equipment failure or distribution-path interruption should not affect IT operations. The practical difference is not a magic availability number; it is how much failure prevention the building buys in electrical and mechanical infrastructure before software-level resilience takes over.

For ML training infrastructure, the calculus differs from traditional enterprise computing. A training run is a long-running batch process, not a real-time transaction system. If a facility-level outage occurs during a two-week training run, the cost is usually a restart from the last checkpoint rather than permanent data loss. Many ML training deployments therefore pair concurrently maintainable facility designs with aggressive application-level fault tolerance: instead of buying every possible layer of building-level redundancy, engineers invest in checkpointing, job restart, and storage systems that can absorb rare facility interruptions. This software resilience commoditizes part of the facility risk, treating the data center as a fallible utility rather than a fortress. Avoided facility-resiliency spend can then be redirected toward accelerator capacity, networking, or storage, which may produce more training progress for the same budget.

Checkpoint 2.4: Infrastructure physics

A team is planning to deploy 256 H100 GPUs (32 nodes) in an existing air-cooled data center with 250 kW of available power capacity and cooling rated for 200 kW at PUE 1.5.

- ☐ **Facility power**: Calculate the total power draw, including cooling overhead.
- ☐ **Binding capacity**: Identify whether power capacity or cooling capacity is the binding constraint.
- ☐ **Required modifications**: Specify what modifications would be needed to support the full deployment.

The rack concentrates power and heat into a physical volume where thermodynamics, not software, sets the limits. A single rack of 32 GPUs, however, is far from sufficient for our 175B model, which may

require thousands of accelerators to train in a reasonable timeframe. The next level of infrastructure aggregates hundreds of racks into a unified computing system: the pod, whose network topology (figure 2.13) is designed around the communication pattern the workload must sustain. Full any-to-any bisection is one design point; torus and rail-optimized designs trade that universality for lower cost or better locality.

2.6 The Pod

Training our 175B model on a single DGX H100 node (8 GPUs, roughly 16,000 TFLOP/s aggregate) would take several months, assuming we can fit the model at all. Reducing training time to weeks requires 100–1,000 nodes operating in concert. A pod is the cluster-scale unit that makes this possible: many racks under one high-speed fabric, power envelope, and cooling design. Pod design is the first point where topology, power, and failure domains become shared infrastructure rather than per-rack concerns.

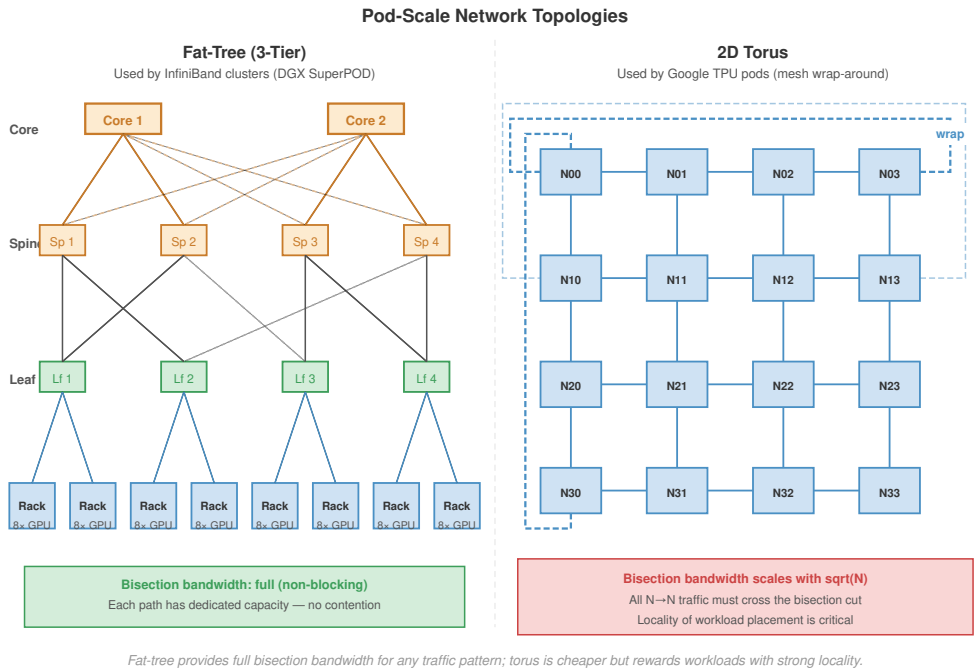


Figure 2.13: Pod-Scale Network Topologies: Two dominant pod topologies compared. (Left) Fat-Tree (3-Tier) with Leaf, Spine, and Core tiers: full bisection bandwidth, used by InfiniBand clusters such as DGX SuperPOD. (Right) 2D Torus with wrap-around links: bisection bandwidth scales as $\mathcal{O}(\sqrt{N})$, cheaper at scale but rewards workloads with strong locality; used by Google TPU pods. The choice shapes which parallelism strategies run efficiently.

The engineering challenge of the pod is wiring those nodes together into a network fast enough to keep gradient synchronization from becoming the bottleneck. The pod aggregates hundreds of racks into a single, coordinated computing system where the network fabric serves the same role that the system bus serves within a single machine.

The scale of this challenge is worth appreciating concretely. A 1,024-node DGX H100 cluster contains 8,192 GPUs, 4,096 NVSwitch chips, 8,192 InfiniBand HCAs, several hundred InfiniBand switches, tens of thousands of cables, and consumes approximately 7–10 MW of power. It occupies roughly 250 racks across one or more data center halls.

The physical weight of such a cluster is also substantial. Each DGX H100 node weighs approximately 130 kg (287 pounds), so 1,024 nodes weigh over 133 metric tons. Combined with racks, switches, cables, and cooling infrastructure, the total weight can exceed 300 metric tons. The data center floor must be structurally engineered to support this concentrated load, with typical floor

loading requirements of 1,500–2,500 kg per square meter for ML clusters, compared to 500–1,000 kg per square meter for traditional server installations. This structural requirement is often overlooked during facility planning and can be a blocking constraint for retrofitting existing buildings.

The physical layout of the data center hall reflects these density constraints. The extreme power density of ML training racks necessitates rigid hot aisle/cold aisle containment in air-cooled sections, where cold air is forced into the enclosed front of the rack and waste heat is captured immediately at the rear exhaust. The "spine" of the hall – the central cable corridor connecting all racks to the aggregation switches – must accommodate thousands of fiber optic cables and power feeds. Overhead cable trays are preferred over under-floor routing to improve airflow and accessibility, carrying the heavy copper power feeds and fragile fiber interconnects that form the nervous system of the cluster. The layout must also accommodate the liquid cooling infrastructure: CDU placement, coolant piping runs, and isolation valves that allow individual racks to be serviced without draining the entire cooling loop. These physical layout decisions, made during facility design, constrain the network topology options available to the training team years later.

Training a 175B model on this cluster requires $6 \times 175 \times 10^9 \times 300 \times 10^9$ total floating-point operations (assuming 300 billion training tokens). At 45 percent MFU – a strong but achievable figure for well-tuned large-model training – the cluster sustains $8,192 \times 1979 \times 10^{12} \times 0.45$ FLOP/s, or about 7.30 EFLOP/s, yielding a physics-limit training time of approximately 12 hours.

In practice, communication overhead, pipeline bubbles, checkpoint I/O, hardware failures, and maintenance windows compound to extend the actual wall-clock time beyond the physics limit. A simplified multiplier chain reaches an approximately 18-hour system minimum; production schedules can stretch to days or weeks once queueing, data readiness, debugging, reruns, and site-specific operational margins enter the plan. The central theme is unchanged: at pod scale, infrastructure imperfections dominate over the raw capability of the silicon. Recovering this lost time is the central challenge of distributed systems engineering, and the solutions span hardware (better networks), software (overlapped communication), and operations (proactive maintenance to minimize downtime).

Napkin Math 2.6: Training time for 175B

We can derive the training time for our 175B model from first principles.

1. **Total FLOPs**: Using the approximation $6 \times P \times D$, where $P = 175 \times 10^9$ and $D = 300 \times 10^9$ tokens: 3.15×10^{23} FLOPs
2. **Cluster throughput**: 8,192 H100 GPUs at 1979 TFLOP/s peak, operating at 45 percent MFU: $8,192 \times 1979 \times 10^{12} \times 0.45$ FLOP/s sustained $\approx$ 7.30 EFLOP/s
3. **Idealized Training Time** (the "physics limit"): $(3.15 \times 10^{23}$ FLOPs$)/(8,192 \times 1979 \times 10^{12} \times 0.45$ FLOP/s$) \div 3600 \approx$ 12.0 hours
4. **Real-world multipliers**:
   - Communication overhead (scaling efficiency ≈ 0.85): $\div 0.85 \rightarrow$ 14.1 hours
   - Pipeline bubbles ($\approx 5\%$): $\times 1.05 \rightarrow$ 14.8 hours
   - Checkpoint overhead ($\approx 3\%$): $\times 1.03 \rightarrow$ 15.3 hours
   - Hardware failures and restarts ($\approx 10\%$ of wall time): $\times 1.10 \rightarrow$ 16.8 hours
   - Maintenance windows ($\approx 5\%$): $\times 1.05 \rightarrow$ 17.6 hours

Total: 17.6 hours in this simplified system-minimum chain, or 0.1 weeks when expressed in weeks. Real production calendars may still stretch to days or weeks after queueing, debugging, reruns, and site-specific operational margins are included. Recovering this lost time is the domain of distributed systems engineering – the subject of Chapter 5.

At this scale, the unit of analysis is no longer a rack or pod of independent servers; the facility has to behave as one coherent machine.

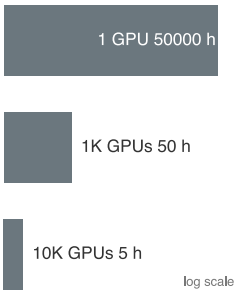
Definition 2.7: Warehouse-scale computer (WSC)

Warehouse-Scale Computer (WSC) is a building-scale computing system in which thousands of servers are operated as a single coherent machine (with the network fabric serving as the system bus, distributed storage as the disk subsystem, and a cluster orchestrator as the operating system), enabling training workloads that would be physically impossible on any single machine.

1. **Significance**: A WSC of 10000 H100 GPUs delivers approximately 9.89 EFLOP/s FP16 peak (and ~19.8 EFLOP/s at FP8), enabling 175B-class training workloads at scales impossible on a single node. However, the system is only as fast as its bisection bandwidth: non-blocking InfiniBand at 50 GB/s per GPU provides 500 TB/s aggregate injection bandwidth; if the fabric is 4:1 oversubscribed, effective bisection bandwidth drops to 125 TB/s, potentially bottlenecking AllReduce and dropping training throughput by 30–60 percent.
2. **Distinction**: Unlike a general-purpose compute cluster (which runs many independent jobs on separate nodes with no requirement for tight coupling), a WSC training cluster operates as a single synchronous instrument in which every node must participate in every AllReduce, making the failure or slowdown of a single node a cluster-wide performance event.
3. **Common pitfall**: A frequent misconception is that WSC design is primarily software engineering or IT operations. WSC design is building-level computer architecture: the physical layout (rack placement to minimize inter-rack hops), power delivery chain (PUE targets, liquid cooling manifolds), and network topology must be co-designed as a unified system: a mistake at any layer propagates through all layers.

The concept was formalized by Luiz Andre Barroso, Urs Holzle, and Parthasarathy Ranganathan at Google in *The Datacenter as a Computer* (Barroso et al. 2019). The key insight is that at Google's scale, *the data center is the computer*, and traditional computer architecture principles must be applied at the building level rather than the chip level.

2.6.1 WSC architecture principles



Fleet-level MTBF falls inversely with GPU count.

The concept of the Warehouse-Scale Computer, articulated by Barroso, Holzle, and Ranganathan at Google, reframes the data center from a *room full of computers* to a *single computer that happens to fill a room* (Barroso et al. 2019). This reframing has profound implications for physical design.

In a traditional data center, each server is an independent unit with its own operating system, storage, and network identity. The facility simply provides power, cooling, and physical security. Servers can be added, removed, or replaced independently, and the workloads running on different servers are unrelated to each other.

In a WSC, the individual server is analogous to a core in a multicore processor: it has no independent utility and exists only as part of the larger system. A DGX H100 node running in isolation cannot train a 175B model; it becomes useful only when connected to hundreds of other nodes through a carefully designed network fabric, with a distributed storage system providing training data and checkpoint storage, and a fleet orchestrator coordinating the work across all nodes.

Two physical consequences follow directly from this reframing. The first is that the network, not the compute, sets the ceiling: within a single node NVLink supplies enough bandwidth for tensor parallelism, but across nodes the fabric must carry gradient tensors, activation checkpoints, and pipeline stage outputs, so a building-scale machine is only as fast as the wires that bind its cores. Chapter 3 develops that fabric in full; here it is enough to note that the WSC reframing makes the interconnect a first-class architectural component rather than a peripheral.

The second consequence is that *failure is a physical certainty, not an exceptional event*. A single GPU has a mean time between failures (MTBF) of approximately 50,000 hours (the canonical GPU MTTF anchor `Systems.Reliability.Gpu.mttf_hours` used in Chapter 7), but reliability degrades linearly with component count: a cluster of 10,000 such GPUs experiences a failure roughly once every five hours. Section E.1.2 derives why the fleet-level rate is simply the per-component lifetime divided

by the component count. This is a building-level property of the physical machine, and it is why a WSC must be designed from the start so that individual component failures degrade rather than halt the running workload, with dual-path power feeds, redundant switch fabrics, and persistent checkpoint storage at every level.

The third consequence is that the building blocks combine differently depending on the workload they must carry. A pod is not assembled from a single canonical part list; the interconnect, the switching, and the memory placement are each chosen against the communication and capacity pattern the dominant workload imposes. Table 2.12 contrasts three production pods to make the point: an NVIDIA SuperPOD wires GPUs through an InfiniBand fat-tree for any-to-any flexibility, a Google TPU pod hard-wires chips into a torus for dataflow throughput, and Meta's Grand Teton arranges the same class of accelerators around embedding memory rather than raw compute.

Table 2.12: Pod Architecture by Workload: Each pod's interconnect, switching, and memory emphasis reflects the communication and capacity pattern of its dominant workload, not a single best design. The economics that decide which workload an organization optimizes for are taken up in section 12.3.

System	NVIDIA SuperPOD	Google TPU Pod	Meta Grand Teton
Interconnect	InfiniBand Fat-Tree	3D Torus ICI	RoCE (RDMA over Ethernet)
Switching	External Switches	Direct Chip-to-Chip	Leaf/Spine Ethernet
Topology	Any-to-any	Nearest-Neighbor Mesh	Hierarchical Rail
Optimization	Flexibility	Dataflow Throughput	Embedding Memory
Dominant Workload	Diverse Research	LLM Training	Recommendation

Grand Teton is the clearest case of memory placement, not compute, driving the layout. Meta's recommendation and ranking models use embedding tables that can reach terabytes, far exceeding the HBM capacity of any single accelerator, while the dense neural-network layers that operate on those embeddings are comparatively small. The design splits the two across the memory hierarchy: the terabyte-scale embedding tables reside in host CPU DRAM (roughly $3–5/GB), and the dense layers execute on GPUs, where the compute density justifies the $10–15/GB cost of HBM. The embedding lookups are memory-capacity-bound but not bandwidth-bound: each request touches only a small slice of a vast table, so cheap, high-capacity DRAM is the right home for them, and the expensive accelerator memory is reserved for the work that saturates it. The systems lesson is that optimal infrastructure is workload-specific, not specification-maximizing: deploying compute-optimized nodes for a memory-capacity-bound workload would strand most of the accelerator's throughput on a problem its silicon was never the bottleneck for.

How the fleet then operates within those physical limits, how it detects degradation, sizes its checkpoint cadence against the failure rate, schedules jobs across the fabric, and how it is sited, priced, and procured, is taken up where the fleet is operated and paid for: section 12.3 for siting, build-vs-buy, and capacity planning, section 12.6 for fleet telemetry and gray-failure detection, Chapter 8 for gang scheduling and topology-aware placement, and section 7.1.2 for the checkpoint-interval trade-off. The physical building blocks are now in place; the operating discipline builds on top of them.

These physical constraints define warehouse-scale computer design. New technologies do not remove the walls; they move them. The technologies below matter because they change where the memory, communication, or power constraint binds.

2.7 Infrastructure Technologies That Move the Walls

A new data center will host three or four generations of accelerators over its 15-year structural lifetime, so future-facing technologies matter only when they change a durable planning constraint. The useful test is not whether a technology is novel, but which wall it moves, what new wall appears after the move, and which fleet-design decision changes as a result. Compute Express Link (CXL) moves the memory-capacity boundary. Wafer-scale integration moves the chip-to-chip communication boundary. Advanced packaging moves the die boundary inside the accelerator package. None of them repeals the physics of data movement; each relocates the place where infrastructure teams must spend power, cooling, cost, and software complexity.

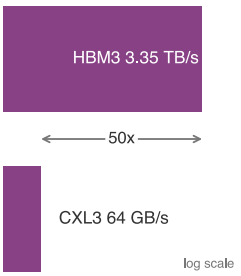
2.7.1 Compute express link (CXL)

An important node-architecture technology is **Compute Express Link (CXL)**[14], a cache-coherent interconnect built on the PCIe physical layer that enables CPUs, accelerators, and memory expansion devices to share a coherent address space. Its value is not that it makes remote memory fast. Its value is that it makes capacity less rigid. Instead of treating each node's host DRAM as a fixed local pool and each accelerator's HBM as an isolated high-speed island, a CXL fabric can expose memory expanders or pooled memory to the devices that need capacity at a particular phase of the training step.

For ML training, the natural target is cold or phase-local state rather than the active matrix-multiply working set. The Adam optimizer state for our 175B model occupies 1,400 GB, and it is touched during parameter updates rather than streamed through every Tensor Core operation. A shared CXL pool could therefore reduce per-node memory pressure by holding optimizer shards, embedding-table spillover, or local data-cache buffers outside the accelerator HBM tier. The training framework still has to schedule this state carefully, but the placement problem becomes less binary than "fits in this node" or "spills to storage."

The new wall is bandwidth. A representative CXL memory path over PCIe Gen5 x16 provides 64 GB/s, while H100 HBM provides 3.35 TB/s. That 52× gap means weights and activations for the forward and backward passes still belong in HBM. CXL is a capacity tier, not an HBM replacement. The fleet-design implication is concrete: planners should ask whether server platforms, rack layouts, and orchestration software can absorb CXL memory pooling, not whether CXL removes the need for high-bandwidth accelerator memory.

14 **CXL (Compute Express Link)**: An open standard (CXL Consortium, founded by Intel in 2019) that adds cache-coherent memory semantics atop the PCIe physical layer through CXL.io, CXL.cache, and CXL.mem. CXL 3.0 extends the model toward memory pooling, allowing capacity to be shared through a fabric rather than fixed permanently inside one server chassis.



HBM remains 52× faster, so CXL is capacity only.

2.7.2 Wafer-scale integration

At the opposite extreme from CXL sits the wafer-scale engine introduced in section 2.2.1, which collapses the cluster onto a single die and makes on-chip bandwidth nearly free. The architecture and yield mechanics were covered there; what matters for infrastructure planning is the wall it moves. By putting the working set on the wafer, the chip-to-chip communication wall largely disappears, but a new wall takes its place: capacity and injection bandwidth.

Our 175B running model needs 350 GB for FP16 weights alone, far beyond the 44 GB of on-wafer SRAM. Cerebras addresses this through weight streaming: external MemoryX units feed weights to the wafer over a 1.2 TB/s path while the wafer executes each layer, decoupling model capacity from the on-wafer memory. That can be a good match for inference and for workloads whose computation stages cleanly, but training reintroduces harder scheduling questions because activations, gradients, and optimizer updates must be coordinated across forward and backward passes. The wafer trades the communication wall for a streaming-bandwidth wall.

The lesson for infrastructure planners is not that wafer-scale systems replace GPU fleets. It is that memory proximity and memory capacity trade against each other. WSE-style designs buy extraordinary local bandwidth by putting the working set near the arithmetic, while CXL buys flexible capacity by moving memory farther away. HBM sits between those extremes: off-chip but on-package, smaller than host memory but fast enough for the active computation. The right architecture is the one whose memory tier matches the workload's reuse pattern.

2.7.3 Advanced packaging

Underpinning both directions is the evolution of **advanced packaging**. Packaging moves the boundary inside the accelerator package rather than all the way out to the rack or all the way in to a single wafer, and it matters because the reticle limit is only about 858 mm² for a single exposure. **2.5D interposers**, such as TSMC's CoWoS, mount compute dies and HBM stacks on a shared silicon substrate, allowing chiplet-based accelerators to behave like a single device for most software. **3D stacking**, with approaches such as SoIC, places SRAM caches or logic directly above one another and shortens data paths further. The H100-to-B200 transition shows the fleet consequence: the package can deliver far more local arithmetic and die-to-die bandwidth, but the rack must absorb higher power density, tighter cooling requirements, and less tolerance for power transients before the silicon advantage becomes useful infrastructure.

The common theme is relocation rather than elimination. CXL moves capacity outward into a fabric and leaves bandwidth as the limiting discipline. Wafer-scale integration moves communication

inward and leaves capacity and yield as the limiting disciplines. Advanced packaging compresses die-to-die distance and leaves power density as the limiting discipline. Infrastructure planning therefore still begins by identifying where the next byte moves, how quickly it must arrive, and what physical system must be built so that movement does not dominate the workload.

2.8 Fallacies and Pitfalls

Because the bottlenecks move rather than disappear, the same planning mistakes recur: teams optimize the visible specification and miss the binding constraint. Compute infrastructure mistakes usually come from optimizing the visible specification while ignoring the binding physical constraint. The fallacies below translate the chapter's node, rack, pod, and facility lessons into operational warnings.

Fallacy: *More GPUs always means faster training.*

Engineers frequently assume that training time scales linearly with GPU count: doubling the GPUs should halve the training time. In practice, communication overhead grows with cluster size and eventually dominates.

Amdahl's Law establishes the theoretical limit: the sequential fraction of the computation (gradient synchronization, pipeline bubble time, data loading stalls) bounds the maximum speedup regardless of parallelism. For a 175B-parameter model with 350 GB of gradients, the AllReduce at each training step requires every GPU to exchange data with every other GPU.

On a cluster of 1,024 GPUs, this synchronization can consume 30–50 percent of the total step time if the network is not carefully engineered. Scaling from 1,024 to 2,048 GPUs doubles the hardware cost but may reduce training time by only 30–40 percent, yielding rapidly diminishing returns. The scaling efficiency curve is concave: the first 100 GPUs provide nearly linear speedup, the next 900 provide diminishing returns, and beyond 2,000–4,000 GPUs the marginal benefit per GPU approaches zero for most model sizes.

The correct approach is to profile useful compute time, exposed communication time, and idle time at the current scale before buying the next pod. If the added GPUs mostly increase exposed communication, procurement is buying idle time rather than training progress. Section 5.4.1 later turns this local ratio into a scaling-efficiency estimate before committing to procurement.

Pitfall: *Planning training throughput from peak FLOP/s alone.*

Procurement teams often select accelerators by comparing peak FLOP/s specifications. This reasoning ignores the Roofline Model's central insight: if the workload's arithmetic intensity falls below the ridge point, memory bandwidth, not compute, limits performance. The H100's ridge point is approximately 295.2 FLOP/byte. LLM inference operates at 1–2 FLOP/byte, achieving less than 1 percent of peak FLOP/s. Even LLM training, which benefits from batching, typically achieves 30–50 percent of peak FLOP/s due to memory-bound attention kernels, activation recomputation, and communication stalls. Selecting hardware by peak FLOP/s alone is analogous to selecting a car by top speed while ignoring fuel efficiency for a daily commute.

Fallacy: *Power and cooling constraints can be handled after accelerator purchase.*

Teams plan GPU purchases based on compute requirements without verifying that the target facility can deliver the required power and cooling. These are hard physical limits with long lead times.

A single rack of four DGX H100 systems requires 33.5 kW of power. Upgrading a data center's electrical infrastructure (new transformers, switchgear, and UPS capacity) requires 12–18 months of lead time. Cooling plant upgrades (chillers, piping, pumps) require 6–12 months. Even seemingly simple changes, such as upgrading the PDU in a single rack from 20 kW to 60 kW, can require new cabling and circuit breaker panels.

Organizations that order hardware without confirming facility readiness risk having millions of dollars of GPUs sitting in shipping containers in the parking lot while the building is retrofitted. This scenario is not hypothetical: several organizations experienced exactly this situation during the 2023–2024 GPU rush, when GPU delivery times shortened faster than data center construction timelines.

Pitfall: *Buying a newer accelerator without checking the workload's arithmetic intensity.*

A team deploys B200 GPUs (4,500 TFLOP/s FP8 peak) for an inference workload serving a 7B-parameter model at batch size 1. The arithmetic intensity is approximately 1 FLOP/byte, placing the workload deep in the memory-bound region.

The B200's memory bandwidth (8 TB/s) is only 2.4× higher than the H100's (3.35 TB/s), so this batch-1 workload tracks bandwidth rather than headline compute. If the headline comparison is B200 FP8 against H100 FP16/BF16-class throughput, the compute peak is about 4.6× higher but decode still improves by only about 2.4×. The team paid for compute they cannot use. An H100 with its lower cost per unit of memory bandwidth would have been a more cost-effective choice for this specific workload.

Fallacy: *Air cooling is sufficient for AI workloads.*

Operators of legacy facilities assume air cooling can scale to dense AI racks with enough additional fans and chiller capacity. It cannot. Air physically cannot remove heat fast enough above roughly 30 kW per rack; dense AI racks can consume 60 to 120 kW, and Blackwell-class rack designs push past 130 kW. Attempting to air-cool such a cluster leads to thermal throttling, sustained clock reductions, and ultimately component failure—not a performance penalty but a hardware liability. Liquid cooling is not an optimization for these clusters; it is a thermodynamic requirement that the rack-density choice forces independently of any TCO analysis.

Fallacy: *All GPU-hours are equivalent in TCO analysis.*

A common error in cost comparisons is to treat one GPU-hour on an H100 as interchangeable with one GPU-hour on an A100 or V100. In reality, the effective cost per unit of useful computation varies enormously across GPU generations: newer accelerators deliver far more useful throughput per dollar. An H100 GPU-hour delivers 6.3× the TFLOP/s of an A100 GPU-hour and 15.8× the TFLOP/s of a V100 GPU-hour. A training run that takes 10,000 V100 GPU-hours would require only approximately 632 H100 GPU-hours to complete the same amount of computation. The relevant metric is not cost per GPU-hour but cost per useful TFLOP-hour, which accounts for both the price per hour and the effective throughput delivered. Organizations that benchmark cloud options by GPU-hour cost alone systematically overestimate the expense of newer, more efficient hardware.

Pitfall: *Upgrading the network before identifying the training bottleneck.*

Teams sometimes invest heavily in upgrading their network fabric (from 200 Gb/s to 400 Gb/s InfiniBand, for example) expecting a proportional improvement in training speed. The improvement depends entirely on whether the training loop is communication-bound at the current network bandwidth. If the compute time per training step is 20 seconds and the communication time is 2 seconds, the workload is compute bound: doubling the network speed reduces total step time from 22 to 21 seconds, a mere 4.5 percent improvement. The same investment in additional GPUs (reducing compute time) or higher-MFU software optimization (reducing wasted compute) would yield far greater returns. The Roofline Model applies to the cluster as a whole, not just to individual chips: diagnose whether the bottleneck is compute or communication before investing in either.

Fallacy: *Average power draw is sufficient for cooling design.*

Accelerator power draw varies dynamically between communication phases (lower power) and compute phases (higher power, at or near TDP). The average power draw across a training step is typically 70–85 percent of TDP. Teams that design their cooling infrastructure for this average, rather than for the TDP-level peak, discover that chip temperatures spike during compute phases, triggering thermal throttling that reduces sustained throughput. The cooling system must be sized for the worst-case thermal load (all GPUs at TDP simultaneously), even though this peak is sustained for only a fraction of each training step. The penalty for undersized cooling is insidious: the system appears to function normally (no crashes, no errors), but MFU quietly degrades by 10–20 percent because the GPUs are throttling their clock frequency to stay within thermal limits.

Pitfall: *Selecting models by HBM capacity without checking bandwidth.*

Procurement teams sometimes focus exclusively on whether a model's weights "fit" in the available HBM, neglecting the bandwidth dimension. A model that fits in HBM but cannot be served at acceptable latency due to insufficient bandwidth is just as unusable as a model that does not fit. Consider two scenarios for serving a 70-billion-parameter model: (a) an accelerator with 192 GB of HBM2e at 2.04 TB/s bandwidth, and (b) an accelerator with 80 GB of HBM3 at 3.35 TB/s bandwidth. Option (a) has more than enough capacity (the model requires only 141 GB in FP16) but delivers tokens at 141 GB/2.04 TB/s = 69.2 ms per token. Option (b) requires careful quantization to fit (141

GB in FP16 would not fit in 80 GB, but 71 GB in INT8 does) but delivers tokens at 71 GB/3.35 TB/s = 21 ms per token, which is 3.3× faster. For latency-sensitive serving, option (b) is the better choice despite its smaller capacity. The right accelerator is the one whose bandwidth and capacity jointly satisfy the workload's requirements.

Fallacy: *Data loading can be fixed after deployment.*

Teams that focus all their predeployment optimization on GPU kernel performance and communication efficiency sometimes discover, postdeployment, that their GPUs are starved for data. The data loading pipeline (storage I/O, preprocessing, host-to-GPU transfer) must sustain a throughput equal to or greater than the GPU cluster's consumption rate. For language model training, the raw data ingestion rate is modest (megabytes per second), but the preprocessing pipeline (tokenization, sequence packing, shuffling) can become a CPU bottleneck when each GPU consumes data faster than a single CPU core can prepare it. The fix is straightforward but must be planned in advance: allocate sufficient CPU cores for data loading workers (typically 4-8 per GPU), stage training data on fast local NVMe storage rather than relying on shared network filesystems, and pipeline the data loading to overlap with GPU computation.

Pitfall: *Buying homogeneous clusters without modeling workload diversity.*

The intuition that uniform hardware simplifies scheduling and reduces stragglers often leads organizations to retire capable older generations prematurely. For our 175B model, the cost-optimal strategy frequently involves a mixed-generation fleet. While training requires the raw FLOP/s and interconnect bandwidth of H100s to minimize synchronization overhead, inference serving is memory-bandwidth-bound rather than compute-bound.

An A100 offers 2.04 TB/s of bandwidth at a significantly lower capital expenditure than the H100's 3.35 TB/s. By dedicating H100 nodes to training and A100 nodes to inference, an organization can reduce TCO by 15–25 percent compared to an all-H100 fleet. The fallacy lies in conflating *job-level* homogeneity – which is critical to prevent stragglers within a single distributed training run – with *cluster-level* homogeneity. A sophisticated scheduler can effectively manage a heterogeneous fleet, routing bandwidth-intensive inference jobs to older hardware where the bandwidth-per-dollar ratio is competitive, while reserving peak compute nodes for training throughput.

Fallacy: *The last mile is a minor installation detail.*

Engineering teams often allocate 90 percent of their planning effort to hardware selection and facility design, assuming that racking and stacking is a deterministic commodity task. In reality, the last mile – physically installing servers, routing cables, filling coolant loops, and running burn-in tests – frequently delays production availability by 2–4 months. A cluster for our 175B model involves thousands of cables; a single loose InfiniBand connection or a pinched fiber optic cable can degrade effective bisection bandwidth by 50 percent, stalling distributed training. Insidious issues like firmware incompatibilities between GPU driver versions and InfiniBand switch firmware, or NUMA misconfigurations in the BIOS, often manifest only under sustained load. Experienced infrastructure teams allocate 15–20 percent of the total project timeline specifically for commissioning and burn-in, running synthetic stress tests (NCCL-tests, HPL benchmarks) for weeks to weed out "infant mortality" failures before a single production job is scheduled.

Pitfall: *Treating commissioning and burn-in as schedule slack.*

Commissioning is part of the system build, not a buffer that can be consumed when hardware delivery slips. Burn-in tests, cable validation, firmware qualification, thermal soak, and NCCL stress runs are the first time the node, rack, network, and facility layers operate as one machine. Compressing that phase turns installation defects into production failures, where each debugging cycle idles the full fleet rather than a small acceptance-test window.

2.9 Summary

The physical infrastructure of machine learning spans from the transistor to the data center, with the running example of training a 175B-parameter model grounding each concept in quantitative reality. The chapter's central lesson is a constraint cascade: accelerator power determines rack cooling, rack density determines pod layout, pod topology determines scaling efficiency, and scaling efficiency determines whether the economics make sense.

That cascade begins at the accelerator. Matrix multiplication workloads demanded Tensor Cores and systolic arrays; weight streaming exposed the memory wall and drove HBM; model states that

exceeded single-chip capacity required multi-accelerator nodes and NVLink. It continues beyond the server, where synchronous training creates power transients, rack densities above 100 kW force liquid cooling, and pod-scale communication determines whether more GPUs accelerate training or simply wait on the network.

Two analytical themes organize those details. The generality tax recurs at every level: accelerators trade control flexibility for arithmetic throughput, networks trade any-to-any flexibility for workload-specific efficiency, and cooling systems trade general-purpose air handling for liquid loops that match dense racks. The Roofline Model introduced in section 2.3.2 supplies the common diagnostic task: identifying whether useful work at each boundary is limited by compute or by data movement.

The resulting hierarchy mirrors the classical memory hierarchy at warehouse scale. HBM is fast and small, NVLink is fast within a node, InfiniBand is slower across nodes, and distributed storage is slowest but largest. The bandwidth hierarchy introduced in section 2.4.1 therefore dictates the parallelism hierarchy: tensor parallelism maps to NVLink, pipeline parallelism maps to rack-local fabrics, and data parallelism spans the full pod. Software parameters such as precision, checkpoint interval, batch size, and gradient accumulation are not independent knobs; they are responses to those physical limits.

Our 175B-parameter model has served as the persistent architectural forcing function connecting every layer of this hierarchy. At the accelerator level, processing a single token requires about 350 billion floating-point operations, yet the arithmetic is not the limiting resource on Tensor Cores. At the memory level, the 350 GB FP16 weight tensor exceeds a single accelerator's HBM capacity and saturates HBM bandwidth once it is sharded or compressed for serving; streaming these weights at 3.35 TB/s creates a bandwidth-floor latency of over 100 ms per token for one accelerator-equivalent HBM path. Moving to the node level, the full training state – comprising weights, gradients, optimizer states, and activations – expands beyond 2 TB, shattering the 80 GB limit of individual chips and requiring tensor parallelism, data-parallel sharding, and memory offload. At the rack level, the thermal density of 32 such accelerators drawing 700 W each, plus host CPUs, memory, and networking overhead, necessitates 33.5 kW of power delivery and liquid cooling infrastructure. At the pod level, training within a viable two-week window requires synchronizing over 1,000 GPUs across a non-blocking InfiniBand fabric, where the statistical certainty of hardware failure forces a checkpointing strategy that trades compute cycles for reliability. At the economics level, bursty cloud rental can beat ownership, while owned clusters become compelling only when sustained utilization is high enough to amortize hardware, power, cooling, and facility risk. Every constraint in this chapter – from the width of the HBM bus to the topology of the data center network – traces back to the single arithmetic fact that this model does not fit on a single chip.

The practical method transfers beyond this particular model. Identify the binding constraint at each level, quantify its effect, and propagate the consequence upward through the stack. Selecting the fastest accelerator is counterproductive if the cooling infrastructure cannot remove its heat. Building the densest rack is futile if the power grid cannot supply its demand. Deploying the widest network is wasteful if the workload's communication pattern does not use the bandwidth. Infrastructure engineering is therefore systems engineering in its purest form: a quantitative discipline for reasoning about how chip-level choices alter rack power, facility cooling, network topology, reliability strategy, and total cost of ownership.

Key Takeaways: The data center is the computer

- **The generality tax**: Accelerators achieve 10–100× higher throughput than CPUs by dedicating silicon area to matrix arithmetic rather than branch prediction and out-of-order execution. The trade-off is reduced flexibility. Each step along the spectrum from GPU to TPU to custom ASIC reclaims more die area for arithmetic.
- **Memory wall diagnosis starts with roofline**: For single-request LLM inference, memory bandwidth determines latency because arithmetic completes in microseconds while data transfer takes milliseconds. The ridge point (I_{ridge} = peak FLOP/s divided by memory BW) separates memory-bound LLM decode ($I \approx 1$) from compute-bound training ($I > 2{,}000$).

- **Bandwidth hierarchy dictates parallelism**: The 9× per-direction bandwidth cliff between NVLink and InfiniBand confines tensor parallelism to within a node and data parallelism to across nodes. Pipeline parallelism spans nodes when model depth exceeds a single node's capacity.
- **Peak vs. sustained**: Model FLOPs Utilization (MFU) of 40–50 percent is considered good for large-scale training. Capacity planning must use sustained throughput, not peak specifications.
- **Facility efficiency is part of the computer**: Liquid cooling can deliver 90–95 percent of facility power to computation, compared to 50–65 percent for air-cooled facilities. At pod scale, network topology, cooling architecture, and power delivery must be co-designed for the dominant workload.
- **Failure at scale**: A 10,000-GPU cluster experiences roughly one GPU failure every five hours. Checkpointing strategy is a first-order performance concern that must be designed into the system from the start.
- **Lifetime cost beats purchase price**: Ownership cost includes electricity, facility construction, networking, and staffing, not just the purchase price, so owning infrastructure pays off only above a utilization threshold that hardware price alone never reveals. Section 12.3 works the build-vs-buy breakeven and the full fleet cost where the fleet is operated and paid for.

For most of computing history, the computer was the chip. This chapter marks where that stops being true. Once a model no longer fits on a single accelerator, a single fact propagates upward through every layer: the chip sets the node's power, the node sets the rack's heat, the rack sets the pod's topology, and the topology sets whether the economics close. No layer can be tuned in isolation, because each inherits its limits from the one beneath it. At this scale the unit of computation is no longer the *processor* but the *building*, which is why the rest of this volume treats the data center, not the GPU, as the machine being engineered.

What's Next: From silicon to wires

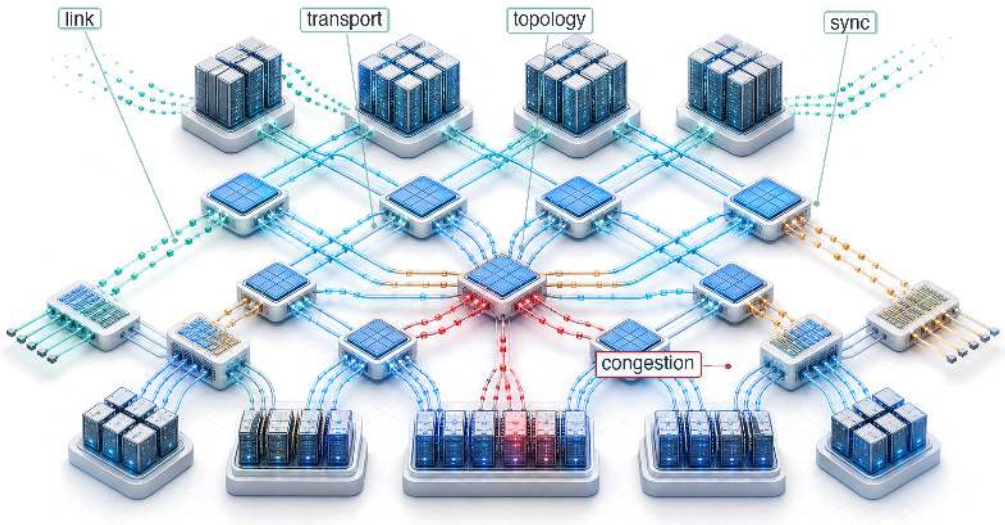
We have mapped physical infrastructure from Tensor Core transistors to megawatt data-center substations. A fleet of disconnected nodes, however, is not yet a computer; it is a warehouse of expensive accelerators. The network fabric turns those nodes into a unified training system by carrying gradients, activations, and checkpoints between them. The chapter repeatedly exposed the same constraint: communication bandwidth limits scaling efficiency. The bandwidth hierarchy showed a 9× cliff between NVLink and InfiniBand, and the topology examples showed that fabric design can change AllReduce time by 2–3×. Chapter 3 examines the physics, protocols, collectives, and topology decisions that determine whether a 10,000-GPU cluster behaves like one coherent machine or thousands of isolated devices.

Research Questions: For further inquiry

- Given a target workload, how should a team decide whether the next dollar should buy FLOP/s, HBM bandwidth, network capacity, or facility power?
- How can the same model require different accelerator choices for training and inference?
- What makes the data center, rather than the accelerator, the relevant unit of computation?
- When is specialization worth the loss of flexibility across accelerators, ASICs, wafer-scale systems, CXL, and advanced packaging?

3

Network Fabrics

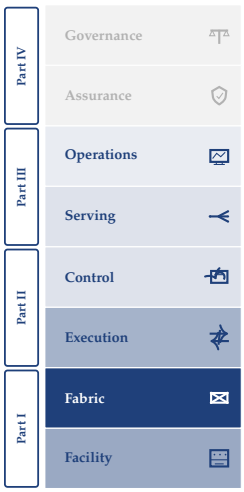


3.1 The Synchronization Backbone
3.2 How ML Networking Inverts Data-Center Assumptions
3.3 Level 1: Wire and Link
3.4 Level 2: Transport and the Performance Model
3.5 Level 3: Switch and Topology
3.6 Level 4: Fabric Behavior (Congestion, Routing)
3.7 Level 5: Cluster Design and Case Studies
3.8 Network Virtualization
3.9 Monitoring and Debugging
3.10 Network Technologies That Move the Ceiling
3.11 Fallacies and Pitfalls
3.12 Summary

Purpose

Why does the network connecting accelerators matter more than the accelerators themselves at scale?

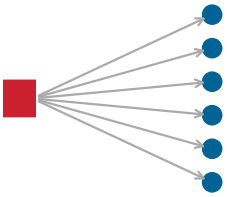
A single accelerator can perform trillions of operations per second, but distributed training requires those operations to coordinate across thousands of devices. Every synchronization point, whether gradient averaging, activation exchange, or parameter update, depends on network bandwidth and latency. When network capacity cannot keep pace with accelerator throughput, accelerators sit idle waiting for data to arrive, and adding more accelerators makes the problem worse rather than better. At sufficient scale, network design dominates system performance: the topology determines which communication patterns are efficient, the bandwidth determines how large models can be partitioned, and the latency determines how tightly coupled training can be. Organizations that treat networking as an afterthought discover that their expensive accelerators deliver a fraction of theoretical performance because the network became the bottleneck nobody planned for. In C^3 terms, the fabric is where the communication tax is levied: its topology and bandwidth set how tightly communication bounds compute across the fleet.



Learning Objectives

- Model network communication cost using the α-β framework and identify bandwidth-dominated vs. latency-dominated regimes
- Compare high-performance transport protocols by latency, lossless guarantees, and operational complexity
- Analyze network topologies (fat-tree, rail-optimized, dragonfly) by computing $\text{BW}_{\text{bisect}}$ and hop count for ML collective patterns
- Evaluate congestion-control mechanisms and their impact on tail latency during distributed training
- Design network virtualization strategies for multi-tenant GPU clusters using device sharing and traffic isolation
- Diagnose network performance bottlenecks using RDMA counters, link-level telemetry, and bandwidth testing tools

3.1 The Synchronization Backbone



One slow link stalls every peer that waits on it.

Consider a 175-billion-parameter language model partitioned across 1,024 GPUs, where dense accelerator nodes are no longer useful unless they synchronize as one machine. Each training step requires an AllReduce of 350 GB of gradient data, meaning every GPU must send and receive its share before the next step can begin. If even one link in the fabric is slow, all 1023 other GPUs wait. The network fabric is not auxiliary infrastructure; it is the synchronization backbone that determines whether this cluster trains efficiently or wastes millions of dollars in idle compute. The 350 GB figure follows from the model-size and gradient-volume assumptions used throughout the fleet examples.

In the fleet stack shown in figure 1.13, the fabric turns isolated accelerator nodes into a coherent training system. Accelerators, power delivery, cooling, racks, and pods define what each node can compute in isolation; the fabric determines whether the nodes can act together before communication cost overwhelms computation. Figure 3.1 organizes that design space into five co-dependent levels, from physical signaling to cluster-scale orchestration. The acronyms in the figure are a roadmap: each mechanism is defined when it becomes the binding constraint.

At scale, communication cost can dominate computation cost. The fleet law in equation 1.3 makes this explicit: the nonoverlapped communication term $T_{\text{comm}}(N) - T_{\text{overlap}}$ determines how much of each training step is exposed to the network fabric. The fabric constrains every layer above it in the stack by determining whether communication can be overlapped, which collective algorithms are viable, and how network partitions interact with node failures.

The physical network fabric exists to carry three fundamental collective communication patterns:

- **AllReduce**: An AllReduce[1] sums gradients across thousands of GPUs so that every device holds the identical average, forming the heartbeat of synchronous training.
- **AllGather**: An AllGather[2] collects different model portions so that every GPU can reconstruct the full model state.
- **AllToAll**: An AllToAll, the most demanding pattern, requires every GPU to send unique data to every other GPU, a requirement critical to expert parallelism[3].

Chapter 6 covers the *algorithms* that orchestrate these patterns; the fabric layer supplies the *physics* of the wires and switches that carry them. The distinction matters because the fabric's physical properties (bandwidth, latency, and topology) determine which patterns are efficient and which become bottlenecks.

Systems Perspective 3.1: The network as a gradient bus

In a single machine, the memory bus moves data between the processor and memory. In a distributed training cluster, the network fabric serves the analogous role: it is the **Gradient Bus**

1 **AllReduce**: A collective operation that sums data from all nodes and distributes the result back to all nodes. Chapter 6 develops the mathematical cost models for ring and tree-based AllReduce implementations.

2 **Collective Primitives**: Higher-level communication patterns involving groups of nodes. While Chapter 6 derives the algorithms for AllGather, ReduceScatter, and AllToAll, this chapter addresses the physical fabric requirements ($\text{BW}_{\text{bisect}}$, switch radix) that enable them at scale.

3 **Mixture-of-Experts (MoE)**: An architecture that activates only a subset of model "experts" per input, necessitating AllToAll communication. Section 10.5.4 examines how MoE decouples model capacity from per-token compute cost (O).

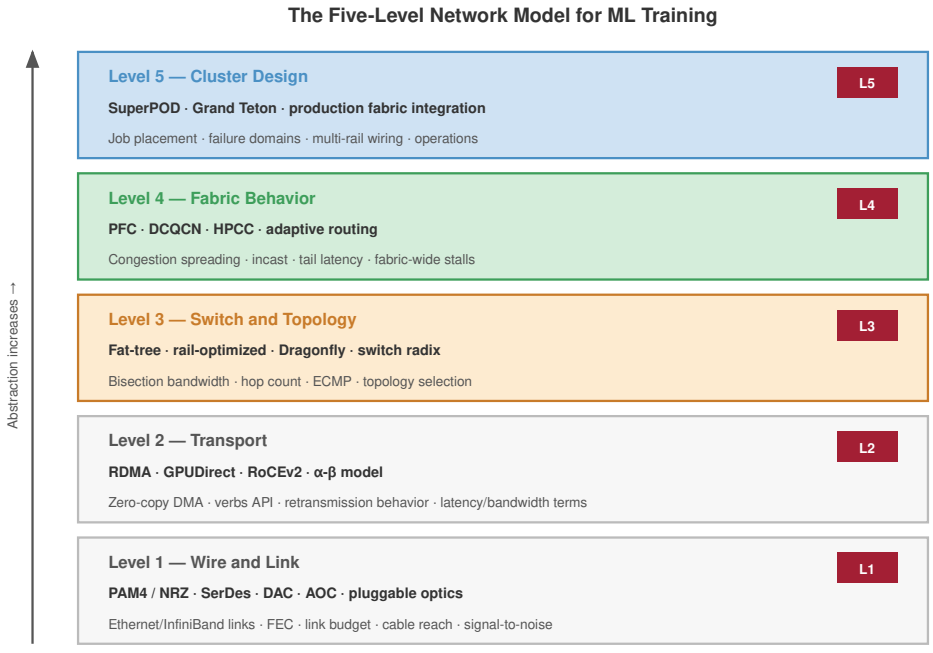


Figure 3.1: The Five-Level Network Model for ML Training: Five layered levels span from the physical wire to cluster-scale design. Level 1, Wire and Link: PAM4/NRZ signaling, SerDes, DAC, AOC, pluggable optics, Ethernet and InfiniBand link layers. Level 2, Transport: RDMA, GPUDirect, RoCEv2, zero-copy DMA, and the α-β communication model. Level 3, Switch and Topology: fat-tree, rail-optimized, Dragonfly, adaptive routing. Level 4, Fabric Behavior: PFC, DCQCN, HPCC congestion control, adaptive routing, incast handling. Level 5, Cluster Design: NVIDIA SuperPOD, Meta Grand Teton, and other production fabric integrations.

> that moves parameter updates between workers. Just as the memory wall in section 2.3 limits single-device throughput, network fabric bandwidth determines multi-device throughput (the communication wall). Protocols, topologies, congestion control, and collective-routing choices all determine how close that bus comes to its physical limits.

The concrete bandwidth cliff that separates intra-node from inter-node communication makes this gradient bus analogy precise. Chapter 2 established that a single H100 delivers 989 TFLOP/s of FP16 throughput with 3.35 TB/s of memory bandwidth. Within a node, eight such accelerators communicate through NVLink at 900 GB/s of aggregate bidirectional bandwidth. The moment computation crosses a node boundary, however, the available per-direction bandwidth drops by a factor of 9×, from 450 GB/s of NVLink (one direction of that bidirectional link) to 50 GB/s (NDR InfiniBand per port). This cliff, the transition from intra-node to inter-node communication, is the central challenge of network fabric design. Section C.2 collects the canonical NVLink and InfiniBand bandwidth specifications and the 1K-GPU cluster reference configuration that anchor these figures. For our 175B model, moving 350 GB of gradients through 50 GB/s links means that the AllReduce alone can take seconds, during which every GPU in the cluster sits idle unless the fabric and collective algorithms can overlap that transfer with computation. Figure 3.2 makes this hierarchy concrete by plotting the bandwidth at each level across four GPU generations.

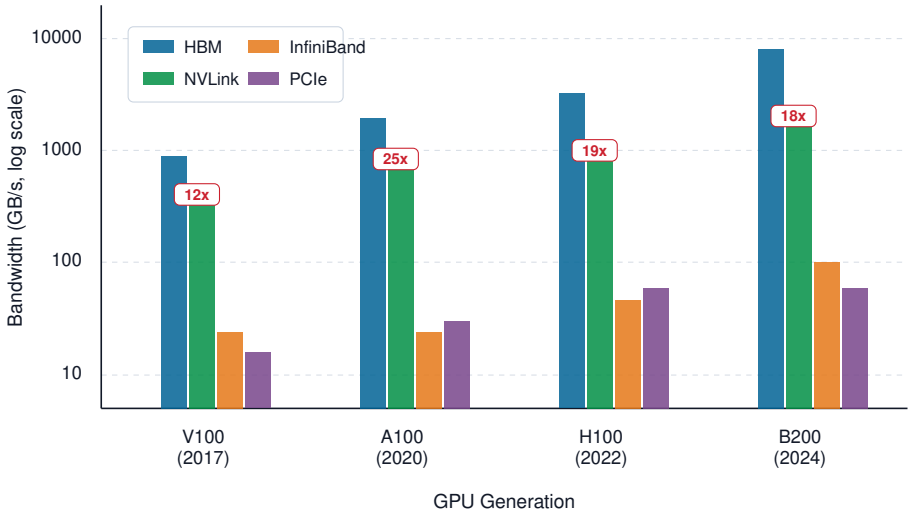


Figure 3.2: The Bandwidth Hierarchy: Bandwidth at four levels of the communication hierarchy across four GPU generations, on a logarithmic scale. While HBM and NVLink bandwidth have grown roughly 9× and 6× respectively from V100 to B200, InfiniBand has grown only 4×. The annotations show the ratio of NVLink bidirectional-total bandwidth to one InfiniBand port, quantifying the cliff that distributed training must cross at every synchronization point; on a like-with-like per-direction basis the H100-to-NDR cliff is about half that, roughly 9× (see text).

Figure 3.2 reveals that this cliff is not an artifact of one accelerator generation. Its annotations compare NVLink's bidirectional-total bandwidth against one InfiniBand port, so they read larger (about 19× for the H100 generation) than the like-with-like cliff. On a per-direction basis, crossing from the local NVLink domain to one InfiniBand port reduces bandwidth by roughly 9×, despite absolute bandwidth improvements at every tier. The persistent cliff reflects fundamental physics: on-package interconnects (NVLink) operate over millimeters of copper, while inter-node links (InfiniBand) span meters of cable and must traverse switches. The ratio determines which parallelism strategies are efficient. Tensor parallelism, which requires continuous high-bandwidth exchange of activations, is viable within a node but impractical across nodes. Pipeline and data parallelism, which tolerate lower inter-node bandwidth, must carry the burden of cross-node communication. Every topology and protocol decision in this chapter attempts to minimize the impact of this hierarchy on collective communication performance.

The ratio becomes visible during a single training step. Figure 3.3 uses a normalized timeline: the compute phases are held fixed at 100 ms, and the AllReduce block represents a small gradient shard chosen to make the intra-node and inter-node contrast visible. It is not the full 175B-parameter gradient exchange; it isolates how crossing the node boundary changes utilization.

The 12.5 percentage-point utilization gap represents millions of dollars in wasted compute over a months-long training run. Network fabric design is therefore the central engineering challenge of distributed training, not an afterthought.

3.2 How ML Networking Inverts Data-Center Assumptions

A network architect from the world of large-scale web services would find the traffic patterns of a distributed training cluster counter-intuitive. Traditional data-center traffic is characterized by a vast number of small, independent, and asynchronous flows. Millions of users accessing a web service generate a stochastic traffic pattern that is well-served by standard Transmission Control Protocol/Internet Protocol (TCP/IP) and statistically multiplexed, oversubscribed networks.

ML training workloads are the complete opposite: synchronous, periodic, and dominated by a small number of massive, collective communication operations. This ML networking inversion re-

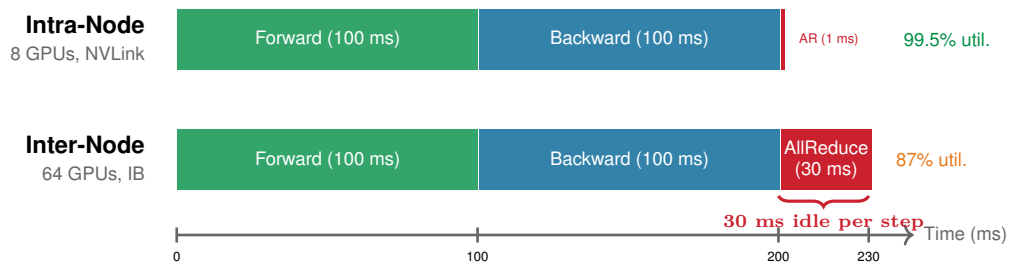


Figure 3.3: The Bandwidth Cliff in a Training Step: Illustrative normalized timelines for the same compute phases: an 8-GPU intra-node job (top) using NVLink at 900 GB/s, and a 64-GPU inter-node job (bottom) using InfiniBand at 50 GB/s. The AllReduce phase (red) grows from a thin sliver to a dominant fraction of the step. Without communication–computation overlap, GPUs sit idle during the entire AllReduce, and the utilization gap between the two scenarios is 12.5 percentage points (99.5 percent versus 87 percent).

verses the core assumptions of traditional network design. Table 3.1 shows the practical consequence: the fabric is optimized for global synchronization time, not average per-flow throughput.

Table 3.1: ML Networking Inverts Traditional Data-Center Assumptions: Where web services require fairness for millions of independent flows, ML training requires a globally synchronized, lossless fabric optimized for a handful of massive collective operations.

Workload Pattern	Traditional Data-Center Assumption	ML Reality
Traffic Pattern	Asynchronous, stochastic, many-to-many	Synchronous, periodic collectives
Flow Type	Millions of small, short-lived flows	A few massive, long-lived "elephant" flows
Performance Metric	Average throughput, per-flow fairness	Tail latency, global synchronization time
Loss Tolerance	Tolerant (TCP retransmits)	Intolerant (one dropped packet stalls all)
Congestion	Localized, independent events	Global, correlated (incast)

As table 3.1 summarizes, the synchronicity inversion is the most fundamental. Web traffic is asynchronous; one user's slow connection does not affect another's. The Bulk Synchronous Parallel (BSP)[4] model governs distributed training: all 1,024 GPUs in a training job must complete their gradient exchange before any of them can proceed to the next step. The slowest link therefore dictates the performance of the entire cluster. A single congested switch port that delays one GPU's packets by 100 ms effectively wastes 100 ms of compute time for all 1,024 GPUs. *Tail latency is not an outlier; it is the bottleneck.*

The flow inversion compounds this problem. Traditional networks are designed for fairness among millions of "mice flows"[5], yet ML training is dominated by a few "elephant flows" corresponding to the gradient AllReduce. A single AllReduce on a 175B parameter model can involve exchanging 350 GB of data. Standard flow control and routing mechanisms like Equal-Cost Multi-Path (ECMP), which hash each flow onto one of several equal-cost paths, are poorly suited to this traffic pattern. A static hash cannot subdivide one elephant flow across all available links, and it can inadvertently map multiple elephant flows to the same link, creating massive congestion while other links sit idle.

The loss inversion completes the picture. TCP/IP was designed for unreliable networks and handles packet loss gracefully through retransmission. RDMA-based protocols used in ML clusters (InfiniBand, RoCE) assume a lossless fabric. A single dropped packet can trigger Go-Back-N recovery, which restarts transmission from the missing packet rather than only repairing the one lost frame, and can stall the sender for milliseconds, creating a catastrophic straggler that delays the entire synchronous training step. ML fabrics must therefore be engineered for zero packet loss. InfiniBand uses credit-based flow control; Ethernet-based RoCE deployments use carefully tuned Priority Flow Control (PFC), a link-layer backpressure mechanism that pauses senders before buffers overflow and therefore depends on sufficient switch buffering.

These inversions explain *why* running large-scale distributed training over a standard enterprise Ethernet network is inefficient or impossible. The network fabric for ML is a distributed, high-performance "bus" for collective communication, designed from the ground up for the unique physics of synchronous, large-scale parallelism.

3.2.1 The five-level model

Each of these inversions traces back to a specific physical or protocol constraint. A **Five-Level Model** specific to high-performance interconnects makes these constraints concrete:

4 **BSP (Bulk Synchronous Parallel)**: Proposed by Valiant (1990) as a bridging model between parallel hardware and software. Synchronous data-parallel training has the same barrier shape: workers compute local gradients, exchange them, and update from a shared step boundary, making tail latency a binding th

5 **Elephant and Mice Flows**: In ML clusters, a few massive flows (gradient AllReduce, model parameters) dominate bandwidth, while many small flows (control messages, heartbeats) are latency-sensitive but carry little data. Terminology from Internet traffic engineering (Guo et al. 2016; Benson et al. 2010). In ML fabrics, the "elephants" are synchronized and correlated, creating incast congestion that standard ECMP load balancing cannot resolve.

determine whether that throughput survives real collective traffic (Alizadeh et al. 2010; Zhu et al. 2015; Li et al. 2019; Gangidi et al. 2024).

- **Level 5: Cluster Design** (section 3.7). Production supercomputers like the NVIDIA SuperPOD and Meta Grand Teton integrate these layers into a unified Gradient Bus (NVIDIA 2023; Meta Engineering 2024).

The levels are not an inventory of networking topics; they are the causal chain that turns synchronous ML traffic into infrastructure requirements. Large gradient messages first stress the wire and transport, then force topology choices that preserve bisection bandwidth, then expose congestion behavior that ordinary enterprise networks can hide, and finally determine whether the whole cluster behaves like one training machine.

The remainder of this chapter ascends this stack, starting from the copper and glass at the bottom. Having reached the top rung, the chapter then turns to the operational layers that surround the fabric in production: sharing it across tenants, observing it under load, and evolving it as the underlying technology moves the ceiling.

3.3 Level 1: Wire and Link

Before analyzing protocols and topologies, we must understand the physical medium. Every network design decision is ultimately constrained by *what* the wire can carry. At 400 Gb/s and beyond, the physics of signal transmission imposes hard limits on cable length, power consumption, and error rates. These are not engineering inconveniences; they are fundamental constraints that shape cluster geometry.

3.3.1 Signal integrity and PAM4

Definition 3.1: PAM4 signaling

PAM4 Signaling is an electrical modulation scheme used in high-speed ML cluster fabrics that encodes two bits per symbol period with four distinct voltage levels, doubling the data rate achievable over a given physical medium without requiring a higher symbol rate.

1. **Significance**: PAM4 enables 400 Gb/s and 800 Gb/s link speeds that sustain the BW required for large-scale gradient synchronization. However, the reduced gap between voltage levels increases susceptibility to noise, requiring Forward Error Correction (FEC), typically Reed-Solomon RS(544,514), that adds about 100 ns of host-FEC processing latency per link in common high-speed Ethernet discussions (Anslow 2016; Yin et al. 2022). Across a three-tier fat-tree (three hops each way), FEC alone can contribute hundreds of nanoseconds of fixed latency to every packet, setting a hard floor on the fabric's fixed per-message startup latency, the quantity that section 3.4.4 formalizes as the α term of the communication cost model.
2. **Distinction**: Unlike NRZ (Non-Return-to-Zero) binary signaling, which uses only two voltage levels and offers better noise margin but is limited to half the data rate per lane, PAM4 trades signal-to-noise ratio for bandwidth density—the same copper trace carries twice the data at the cost of requiring FEC at every port.
3. **Common pitfall**: A frequent misconception is that doubling bits per symbol is a free throughput gain. PAM4 links require FEC at both endpoints, making them fundamentally higher-latency than equivalent NRZ links; latency-sensitive collectives like AllReduce pay this FEC tax on every packet regardless of message size.

To achieve 400 Gb/s, we *cannot* simply toggle a voltage on and off faster. Signal attenuation in copper and chromatic dispersion in fiber impose a hard ceiling on the **symbol rate** (the number of signal transitions per second). High-speed links overcome this by using PAM4 to pack more information into each transition.

However, the gap between adjacent voltage levels in PAM4 shrinks by a factor of three compared to NRZ, making the link highly susceptible to noise. Consequently, high-speed links operate near the

physical limits of reliable detection and require **Forward Error Correction (FEC)** at the physical layer, commonly using Reed-Solomon RS(544,514)-family codes in high-speed Ethernet designs (Anslow 2016; Ethernet Alliance 2025). FEC processing is an implementation-dependent latency term; IEEE 802.3df operator material treats roughly 100 ns of host KP4 FEC and hundreds of nanoseconds of end-to-end module latency as meaningful at AI/HPC fabric timescales (Yin et al. 2022). For our 175B model training, which generates 350 GB of gradient traffic per step across 1,024 GPUs, this latency floor is nonnegotiable. A packet crossing three switch hops in a fat-tree incurs a scenario budget of 300 ns–600 ns of one-way FEC decoding and encoding latency, or 600 ns–1200 ns over a three-hop path in each direction. This "physics tax" sets a hard floor on the fabric's fixed startup latency, formalized in section 3.4.4 as the α term, limiting the speed of latency-sensitive collectives like AllReduce regardless of software optimizations. Figure 3.4 shows the signaling trade-off directly: PAM4 doubles the bits per symbol relative to NRZ, but the smaller voltage gaps reduce noise margin and make FEC part of the link's latency floor.

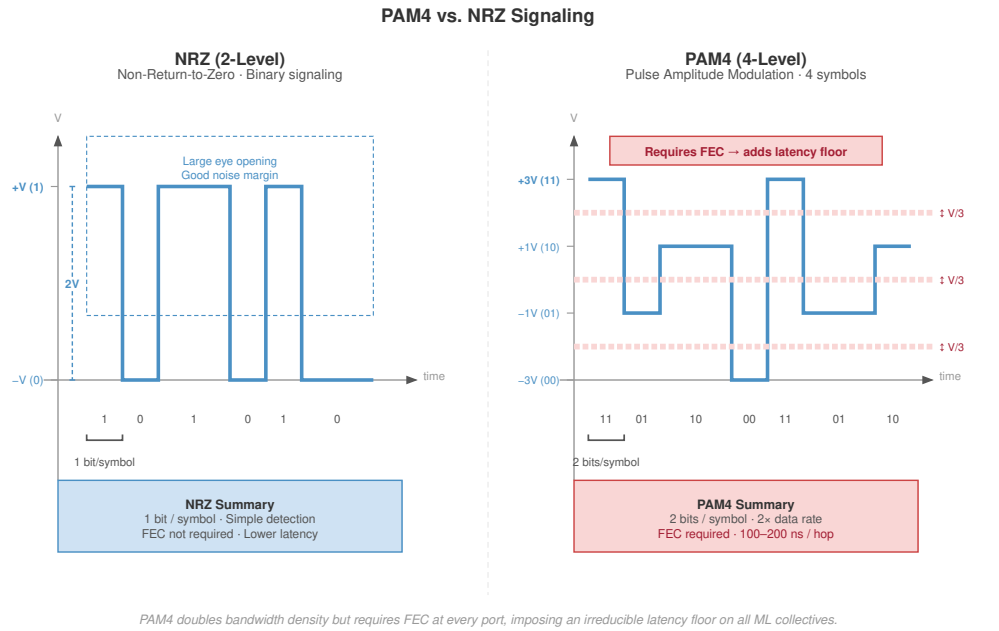


Figure 3.4: Signaling Evolution: NRZ vs. PAM4: To increase bandwidth without doubling clock frequency, high-speed interconnects moved from NRZ (2 voltage levels, 1 bit per symbol) to PAM4 (4 voltage levels, 2 bits per symbol). The trade-off is reduced noise margin: the 'eye' of the signal is smaller, necessitating Forward Error Correction (FEC) and increasing the irreducible α latency of the fabric.

3.3.2 Reach and medium: Copper vs. optics

The choice of physical medium determines how far a link can reach before latency, power, and cost reshape the cluster geometry. Table 3.2 compares the recurring cable and optics options by the attributes that decide where each one belongs.

Table 3.2: Cable and Optics Placement Trade-Offs: Common interconnect media occupy different reach, latency, and cost points, so cluster designers use copper inside racks and optics as distance begins to dominate placement.

Medium	Reach	Cost profile	Latency impact	Typical placement
DAC (Direct Attach Copper)	~3 meters	Lowest cost (~$50) and no optical module power	Lowest media latency; passive copper avoids optical conversion	Within a rack

Medium	Reach	Cost profile	Latency impact	Typical placement
AOC (Active Optical Cable)	3 to 30 meters	Higher cost (~$500) from permanently attached transceivers	Electrical/optical conversion plus Forward Error Correction (FEC) adds hundreds of nanoseconds	Rack-to-rack or row-scale links
Pluggable optics	More than 30 meters	Highest module and power cost, but replaceable optics and fiber cords	Optical conversion and FEC add latency; distance enables topology reach	Network core and pod-scale links

For optical links, Forward Error Correction (FEC)[6] is part of the latency budget that makes reach a topology decision rather than a simple cable choice. The comparison in table 3.2 is the cluster-geometry trade-off in miniature: copper is cheap and low latency but keeps accelerators close together, optical interconnects extend a row or pod at higher power and latency, and pluggable optics reaches the core only when the topology needs distance more than it needs minimum per-hop cost.

[6] **FEC (Forward Error Correction)**: At 400G and 800G speeds, signal integrity is fragile enough that Ethernet PHY designs rely on FEC to reconstruct corrupted bits (Anslow 2016; Ethernet Alliance 2025). The exact latency depends on the PHY and module architecture; for ML fabric modeling, treating FEC as a per-hop latency budget is the conservative point, because a multi-hop path accumulates that budget before software can hide it.

> Systems Perspective 3.2: The cost of distance
>
> In an ML fleet, distance is money. A 10,000 GPUs cluster requires ~20,000 optical links at the spine layer alone. At $500 each with 20 W per pluggable link, that represents $10 million in cabling and 400 kW of continuous power for transceivers alone. The cost dictates **cluster geometry**: architects pack accelerators as densely as possible (70–100 kW per rack) to maximize cheap copper and minimize expensive optics.

3.3.3 SerDes, link budget, and power

Every port depends on a **Serializer/Deserializer (SerDes)** circuit that converts parallel data from the switch ASIC into a serial stream. As bandwidth scales, these circuits have become the dominant power consumer in the network fabric. Consider the energy implications for our cluster of 1,024 GPUs: maintaining full BW_{bisect} for the 350 GB of gradient traffic requires roughly 3,000 high-speed links. If each 400 Gb/s port consumes 25 W (combined SerDes and optical transceiver power), the interconnect alone draws 75 kW continuously—10.5 percent of the cluster's power budget dedicated merely to moving bits, not computing on them.

The **link budget**, a strict decibel (dB) allowance for signal attenuation, governs the reach of these links. As signals traverse copper, they lose energy to skin effect and dielectric absorption. The link budget is the difference between the transmitter's output power and the receiver's sensitivity limit. For a standard 50G PAM4 lane, the budget might be 30 dB. If a 3-meter DAC cable introduces 18 dB of loss and the connectors add another 2 dB, 10 dB of margin remains. However, as frequency doubles to support 100G lanes (for 800 Gb/s ports), the loss per meter increases sharply. The physics forces a brutal trade-off: to maintain signal integrity without exceeding the power budget, cable reach must decrease. Copper links operating at 800 Gb/s are limited to roughly 1–2 meters, physically constraining the diameter of a compute rack and forcing the use of expensive, power-hungry optics for any connection leaving the cabinet. In 1.6 Tb/s-class planning scenarios, SerDes and optical power can account for a large fraction of cluster energy, making power-per-bit a primary scaling constraint. The wire sets hard limits on reach and speed; the transport layer must build a reliable communication primitive on top of these physical links.

3.4 Level 2: Transport and the Performance Model

Large-scale training requires sustained, synchronized bulk transfers. A single AllReduce operation across 1,024 accelerators may move terabytes of gradient data. This pattern demands networks optimized for Remote Direct Memory Access (RDMA) to eliminate CPU overhead and **lossless delivery** to ensure predictable performance.

3.4.1 RDMA and GPUDirect

Definition 3.2: Remote direct memory access (RDMA)

Remote Direct Memory Access (RDMA) is a networking technology used in ML training fabrics that allows one machine to read or write the memory of another machine directly, bypassing the operating system kernel and CPU of both endpoints by offloading transport processing to the network interface card.

1. **Significance**: RDMA reduces end-to-end message latency from the 50–100 µs typical of kernel TCP to approximately 1–2 µs, cutting the L_{lat} term in the iron law by 25–50×. For a 175B-parameter model exchanging 350 GB of gradient data across 1,024 GPUs, RDMA also eliminates 700 GB of redundant memory copies per step by allowing the NIC to read GPU memory directly without staging through host RAM (GPUDirect RDMA).
2. **Distinction**: Unlike traditional TCP/IP, where the CPU processes every packet through the kernel network stack (consuming tens of CPU cores to saturate a 400 Gb/s link), RDMA offloads the entire transport to dedicated NIC hardware, freeing the CPU to orchestrate computation rather than move data.
3. **Common pitfall**: A frequent misconception is that RDMA works reliably on any Ethernet network. RDMA lacks TCP's retransmission logic; a single dropped packet can stall an entire 1,024-GPU AllReduce for 100–500 ms as the Go-Back-N recovery retransmits from the loss point. RDMA requires a lossless fabric—InfiniBand or Ethernet with Priority Flow Control—to operate correctly at scale.

Standard TCP/IP is architecturally unfit for the 400 Gb/s era. The protocol stack was designed when network speeds were orders of magnitude slower than CPU memory bandwidth, but at these line rates that relationship has inverted. Processing a 400 Gb/s stream through the Linux kernel imposes a crushing interrupt tax: copying payload data between user space and kernel buffers can consume the entire memory bandwidth of a dual-socket server, requiring tens of CPU cores merely to keep the pipe full. The result is a CPU wall where the host processor becomes the bottleneck for network traffic, starving the application logic it is meant to serve.

RDMA bypasses this entire layer. By offloading the transport logic to the NIC hardware, it allows applications to read and write directly to remote memory. For ML, GPUDirect RDMA[7] extends this zero-copy principle to the accelerators themselves (NVIDIA 2026a). Without GPUDirect, a gradient update follows a tortuous path: GPU memory → CPU system RAM → kernel buffer → NIC. GPUDirect short-circuits this to a single PCIe transaction: GPU → NIC, as figure 3.5 contrasts. For our 175B model's 350 GB gradient exchange, the optimization eliminates 700 GB of redundant memory copies across the cluster per step, reducing latency and freeing the CPU to orchestrate complex pipelining logic rather than acting as a data mover.

[7] **GPUDirect RDMA**: Introduced by NVIDIA for Kepler-class GPUs and CUDA 5.0, GPUDirect RDMA enables a direct PCIe path between GPU memory and third-party peer devices such as network interfaces (NVIDIA 2026a). Before GPUDirect, every gradient transfer had to stage through host memory, consuming CPU memory bandwidth that competes with data loading. Eliminating this bounce path is one reason overlapping communication with backward-pass computation is feasible at scale.

3.4.2 InfiniBand and RoCE

GPU clusters commonly choose between two RDMA transport stacks, and the decision is a reliability-versus-operations trade-off. Table 3.3 compares where each stack places the burden of losslessness and congestion control.

Table 3.3: RDMA Transport Stack Trade-Offs: InfiniBand and RoCE expose RDMA semantics through different operational contracts, one built around native losslessness and the other around Ethernet compatibility.

Stack	Fabric behavior	Operational burden	Typical fit
InfiniBand (IB)	Purpose-built HPC switched fabric (InfiniBand Trade Association 2000) with credit-based flow control, so losslessness is native at the link layer	Subnet Manager (SM) and Virtual Lanes (VLs) govern routing and traffic isolation	Dedicated training clusters where predictable tail latency outweighs Ethernet ecosystem flexibility

Stack	Fabric behavior	Operational burden	Typical fit
RoCE (RDMA over Converged Ethernet)	RoCEv2 carries RDMA semantics by encapsulating InfiniBand transport headers in UDP/IP packets	Priority Flow Control (PFC), ECN, and workload-aware routing or admission control must approximate InfiniBand's native losslessness (Guo et al. 2016; Gangidi et al. 2024)	Multi-vendor Ethernet fleets that can absorb more congestion-control tuning

The InfiniBand[8] row reflects a protocol heritage that favors hardware-managed losslessness over Ethernet compatibility.

This stack choice is visible at the protocol boundary. Figure 3.6 compares the two stacks, showing how RDMA-based protocols expose a user-space **Verbs API** that bypasses the kernel's traditional TCP/IP stack.

8 **InfiniBand**: Formed in 1999 from the merger of two competing server I/O standards (Future I/O and NGIO), InfiniBand was originally designed to replace PCI as a general-purpose system interconnect. Its pivot to HPC networking preserved the credit-based, hardware-managed flow control that server I/O demanded—and this heritage is precisely why InfiniBand provides native losslessness without the PFC fragility that plagues Ethernet-based RDMA fabrics.

3.4.3 Losslessness and the go-back-n problem

The critical takeaway from figure 3.6 is that the Verbs API provides a uniform programming model, but the reliability guarantees beneath it differ fundamentally: InfiniBand enforces losslessness in hardware, while RoCE must construct it from Ethernet's best-effort foundations using PFC and ECN. ML collectives assume in-order, lossless delivery, but the hardware implementation of this reliability introduces a critical fragility. TCP handles packet loss gracefully via Selective Acknowledgement, retransmitting only the specific missing segment. RDMA protocols like RoCEv2, by contrast, typically rely on simpler recovery paths such as **Go-Back-N** retransmission when rare packet drops occur (Gangidi et al. 2024). The NIC's physical constraints drive this choice: implementing complex reassembly logic for out-of-order packets requires substantial on-chip SRAM, which consumes precious die area needed for SerDes blocks and packet processing engines. The NIC hardware is optimized for throughput, not state management.

The trade-off is a severe penalty upon failure. If a network switch drops a single packet 900 MB into a 1 GB gradient transfer, the receiver discards all subsequent packets, forcing the sender to retransmit the entire tail of the message, potentially 100 MB of data for a single missed frame. At 400 Gb/s, this retransmission triggers a latency spike orders of magnitude larger than the wire delay. In a synchronous training loop where thousands of GPUs wait for the slowest member, a single dropped packet idles the entire cluster. The network fabric must therefore behave as a lossless medium, pushing the complexity of flow control into the switches via Priority Flow Control (PFC) to ensure buffers never overflow.

Checkpoint 3.1: Protocol selection

Consider a 2,048-GPU training cluster that will run both large language model training (gradient messages of several gigabytes) and reinforcement learning (frequent small control messages).

- ☐ **Protocol fit**: Recommend a protocol and justify the choice for both the large gradient messages and the frequent small control messages.
- ☐ **RoCE requirements**: Specify the additional infrastructure RoCE needs compared to InfiniBand.
- ☐ **Shared fabric**: Reevaluate the protocol choice when the cluster also serves inference traffic to external users over the same physical network.

3.4.4 The α-β performance model

Protocol choice determines whether the fabric can behave as a lossless medium; performance modeling then asks how fast that medium can carry a given message. The α-β model decomposes message transfer time as $T(n) = \alpha + n/\beta$, where α is the fixed startup latency and β is the sustained bandwidth (Hockney 1994). Section D.1 develops the full derivation and works the model through concrete message regimes, separating latency-dominated from bandwidth-dominated transfers; section 6.2.1 later applies the same decomposition to collective algorithms. Topology choice directly

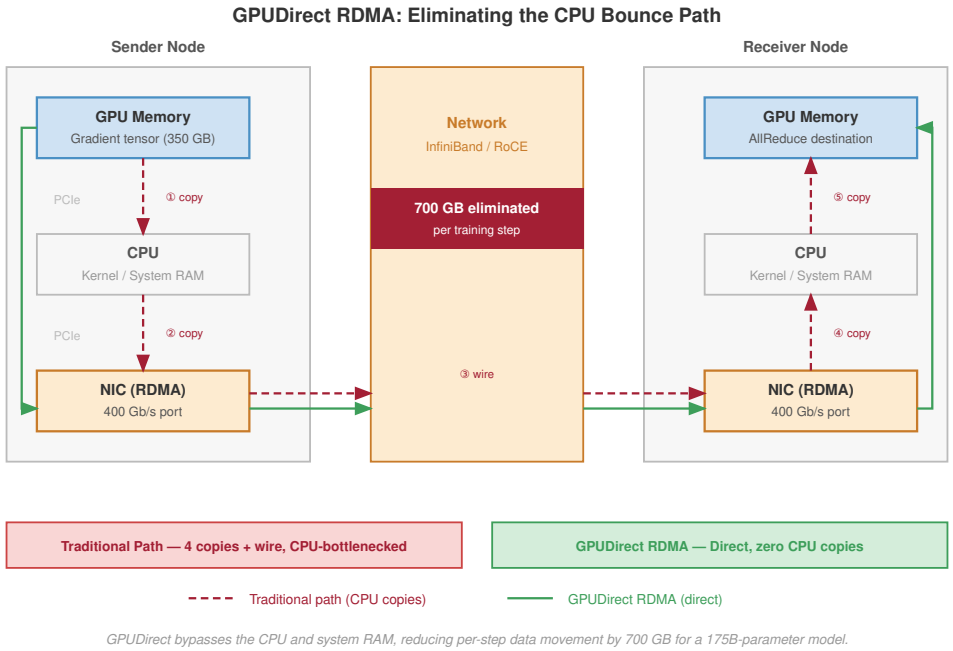


Figure 3.5: GPUDirect RDMA Data Path: Comparison of traditional vs. GPUDirect data paths. Traditional RDMA (top) requires data to be copied to host RAM before transfer to the NIC. GPUDirect RDMA (bottom) enables the NIC to access GPU memory directly via the PCIe bus, eliminating redundant copies and reducing latency for bulk gradient transfers.

shifts both parameters: a fat-tree minimizes α by providing short equal-cost paths, while a ring amplifies α with cluster size because messages traverse a hop count that grows with the number of participants N.

The model reveals two regimes that lead to different engineering responses. For messages with $n < \alpha\beta$, startup cost α dominates the transfer time; small control messages and pipeline bubbles fall in this latency-bound region. For messages with $n > \alpha\beta$, the n/β term dominates; gradient AllReduce falls in this bandwidth-bound region.

For NDR InfiniBand with $\alpha \approx 1.5$ µs and $\beta \approx 50$ GB/s, the crossover point $n^* = \alpha \cdot \beta \approx 75$ KB. Messages smaller than this gain little from more bandwidth; messages larger than this gain little from lower latency. That crossover separates two fundamentally different optimization strategies: reducing hop count to lower α, or adding link bandwidth to raise β. Applying the model to the concrete message sizes that our 175B training job generates on every iteration makes this distinction actionable.

Napkin Math 3.1: The alpha-beta crossover

Problem: A fabric designer is comparing InfiniBand NDR (1.5 µs, 50 GB/s) against a slower Ethernet baseline (5 µs, 12.5 GB/s). For a 4 KB control message and a 350 MB gradient shard, which part of $T(n) = \alpha + n/\beta$ dominates, and why does the faster fabric help for different reasons in each regime?

Math: Apply $T(n) = \alpha + n/\beta$ for a 4 KB control message and a 350 MB gradient shard.

1. **Small message** (4 KB):
   - InfiniBand: $1.5\,\mu\text{s} + 4\,\text{KB}/50\,\text{GB/s} = 1.5 + 0.08 = \mathbf{1.58}\,\text{s}$
   - Ethernet: $5.0\,\mu\text{s} + 4\,\text{KB}/12.5\,\text{GB/s} = 5.0 + 0.32 = \mathbf{5.32}\,\text{s}$
   - **Result**: InfiniBand is 3.4× faster purely due to lower α.

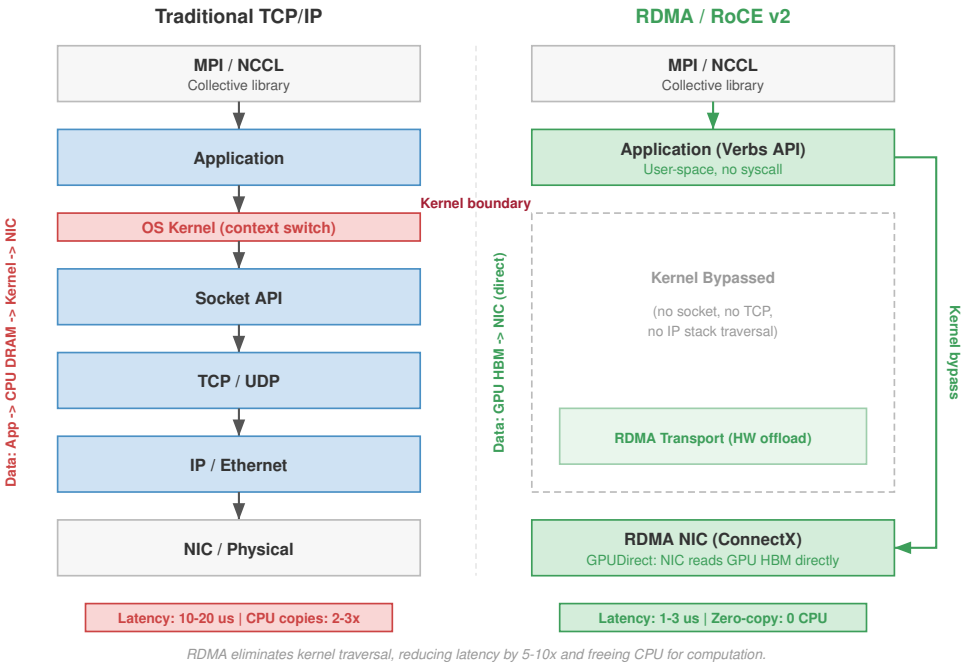


Figure 3.6: Traditional TCP/IP vs. RDMA/RoCE v2 Stacks: Side-by-side comparison. Traditional TCP/IP (left) traverses application, OS kernel (context switch), Socket API, TCP/UDP, IP/Ethernet, and NIC, with latency of 10–20 µs and 2–3× CPU copies. RDMA/RoCE v2 (right) exposes a user-space Verbs API that bypasses the kernel, offloading transport to the RDMA NIC (GPUDirect reads GPU HBM directly), with latency of 1–3 µs and zero-copy CPU overhead.

2. **Large message** (350 MB):
   - InfiniBand: $T = \alpha + n/\beta = 7.0$ ms
   - Ethernet: $T = \alpha + n/\beta = 28.01$ ms
   - **Result**: InfiniBand is 4× faster purely due to higher β.

Systems insight: For large-scale training, the crossover point ($n^* = \alpha\beta$) is typically around 75 KB. Since gradients are megabytes to gigabytes, we are almost always in the bandwidth-dominated regime. However, pipeline parallelism and distributed coordination rely on small messages in the latency-dominated regime, where wire-speed upgrades provide zero benefit and only topology and hop-count reductions matter.

To see *why* this distinction matters in practice, consider two messages that our 175B model training job sends every iteration. The first is a 4 KB pipeline-scheduling control message that coordinates microbatch handoffs between pipeline stages. Applying the model: $T(n) \approx 1.58$ µs for the control message. The bandwidth term contributes only 5.1 percent of the total. Doubling the link speed would save a negligible fraction of a microsecond. For this message, the most direct way to reduce transfer time is to reduce the hop count (which lowers α), not to buy faster links.

The second message is a 350 MB gradient shard for one layer's AllReduce. Now: $T(n) \approx 7.0$ ms. The latency term is invisible. Doubling bandwidth to 100 GB/s would halve this transfer time, a direct and proportional gain. These two cases illustrate *why* network design must address both α and β simultaneously: topology and hop count control the latency-dominated regime, while link speed and path diversity control the bandwidth-dominated regime.

Figure 3.7 makes these two regimes visible across the full range of message sizes. For InfiniBand, the crossover occurs at approximately 75 KB: messages smaller than this are latency-dominated (the flat region on the left), while larger messages are bandwidth-dominated (the linear region on the right). Ethernet RoCE crosses slightly earlier, at about 62.5 KB, because its lower per-link bandwidth makes the transfer term overtake the higher fixed latency at a smaller message size. The 5× latency gap between InfiniBand and Ethernet RoCE dominates for small messages but becomes irrelevant for the multi-megabyte gradient transfers that dominate training communication.

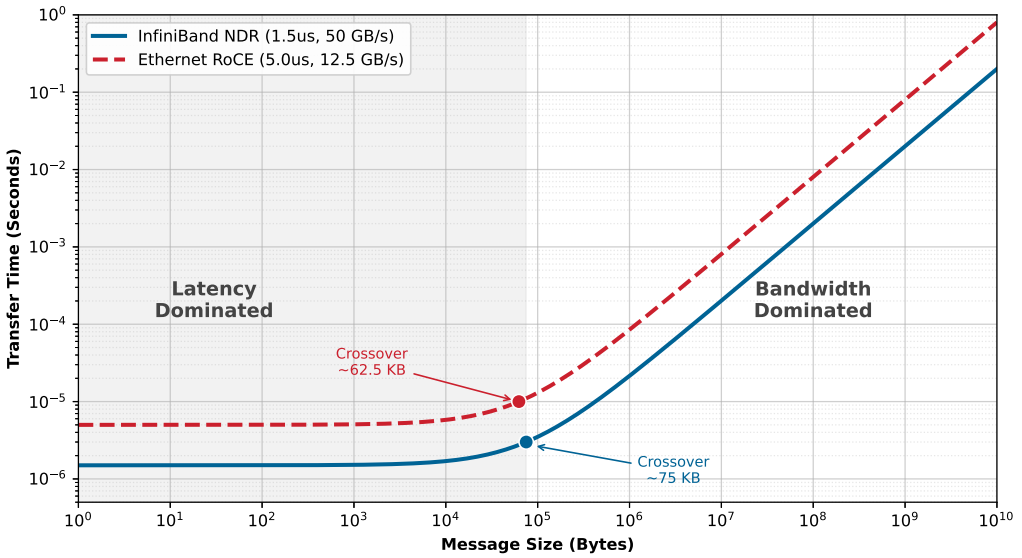


Figure 3.7: The α-β Crossover: Transfer time as a function of message size for InfiniBand NDR and Ethernet RoCE. Small messages are latency-dominated (flat region), while large messages are bandwidth-dominated (linear region). The crossover point marks where investing in bandwidth begins to pay off more than reducing latency.

Beyond classifying individual transfers, the same model identifies when communication overtakes computation. The α-β analysis above focused on the transfer time of individual messages across a single link, treating the network as a point-to-point channel. In a data-parallel training loop, however, the relevant question is whether the collective AllReduce across the full cluster finishes before the next compute step is ready to begin. When the gradient vector grows large enough, the aggregate transfer time for a ring AllReduce exceeds the per-step computation time, and the network becomes the pacing constraint for the entire training job.

Napkin Math 3.2: AllReduce bottleneck threshold

Problem: When does Ring AllReduce become the bottleneck for a 1024-GPU H100 cluster training models at GPT-2 scale and beyond?

Setup: The baseline cluster trains a GPT-2-scale model with 1.5 billion parameters (6 GB of FP32 gradients). Each GPU computes at 989 TFLOP/s FP16/BF16 peak. The network uses NDR InfiniBand ($\alpha = 1.5$ μs and $\beta = 50$ GB/s per link).

Step 1: Compute time per iteration. Assume each GPU processes a synthetic microbatch requiring 5.00×10^{13} FLOPs, chosen to produce a compute phase of about 101 ms for this bottleneck example. At 989 TFLOP/s with 50 percent utilization:

$$T_{\text{compute}} = 101.1 \text{ ms}$$

Step 2: Communication time for Ring AllReduce. With N = 1024 and a gradient payload M of about 6 GB:

- Ring cost model: $T_{\text{ring}} \approx \frac{2(N-1)}{N}\frac{M}{\beta} + 2(N-1)\alpha$; Chapter 6 derives the algorithmic steps behind this expression.
- Total Communication: $T_{\text{ring}} \approx 242.8$ ms

Step 3: Communication fraction.

$$\text{Comm. fraction} = 70.6\%$$

With overlap between communication and computation (possible because the backward pass produces gradients layer by layer), the effective overhead can be reduced, but the network is already a major contributor to iteration time for this configuration.

At 70 billion parameters (280 GB of gradients), the bandwidth term becomes the bottleneck if per-GPU computation stays similar through batch-size scaling:

$$T_{\text{ring}} \approx 11192.1 \text{ ms}$$

Now communication dominates computation under pure data parallelism. Our 175B model, far larger than this scale, would be even more severely bottlenecked. Models beyond a few billion parameters therefore require tensor and pipeline parallelism to partition the model, rather than relying solely on data parallelism, which must AllReduce the full gradient vector.

The α-β model quantifies the speed of a single link, but our 175B-parameter model requires 1,024 GPUs to work in concert. Scaling these transport primitives from a pair of nodes to a warehouse-scale supercomputer demands a **network topology**: a specific pattern of connections that maximizes $\text{BW}_{\text{bisect}}$ while minimizing the hop count and cabling cost for global collective operations.

3.5 Level 3: Switch and Topology

The physical arrangement of switches determines the bisection bandwidth $\text{BW}_{\text{bisect}}$ and whether the fabric is non-blocking. These two properties govern how well the network supports the global communication patterns that dominate distributed training, a constraint captured by the bisection bandwidth theorem (principle 4).

The first property, bisection bandwidth, quantifies the worst-case throughput ceiling that the topology imposes on global collectives. A cluster can have thousands of fast links at the edge and still starve its AllReduce operations if the cross-sectional capacity at the narrowest point in the switching hierarchy is insufficient.

Definition 3.3: Bisection bandwidth

Bisection Bandwidth is a network topology metric defined as the minimum aggregate link capacity crossing any partition that divides the cluster into two equal halves, representing the worst-case throughput ceiling for all-to-all communication patterns such as AllReduce.

1. **Significance**: Bisection bandwidth $\text{BW}_{\text{bisect}}$ directly sets the cluster-scale bandwidth ceiling for global synchronization. A 1,024 GPUs fat-tree with 400 Gb/s links at 1:1 subscription provides $\text{BW}_{\text{bisect}} = 512 \times 50\,\text{GB/s} = 25.6\,\text{TB/s}$ per direction; a 4:1 oversubscribed spine reduces this to 6.4 TB/s, making each AllReduce step 4× slower and turning the network into the dominant iron law bottleneck rather than the accelerator.
2. **Distinction**: Unlike aggregate bandwidth (the sum of all edge link speeds, which can be high even in a poorly connected topology), $\text{BW}_{\text{bisect}}$ measures global connectivity—a star topology with 1,000 edge links all meeting at one central switch has high aggregate bandwidth but $\text{BW}_{\text{bisect}}$ limited by that switch's backplane capacity.
3. **Common pitfall**: A frequent misconception is that adding more leaf switches always increases $\text{BW}_{\text{bisect}}$. In a three-tier fat-tree, oversubscribing the spine layer (using fewer up-

links than downlinks per pod switch) reduces BW_{bisect} below the edge-link total regardless of how many leaf switches are present.

For ML training, where all-to-all communication is standard, full BW_{bisect} is the foundational requirement. A fabric that falls short forces every synchronization step to bottleneck at the narrowest cross-section, and the entire cluster idles while gradients trickle through.

Bisection bandwidth is a metric; the topology property that delivers full BW_{bisect} under arbitrary traffic patterns is a non-blocking fabric. A non-blocking design guarantees that the uplink capacity at every switch tier matches or exceeds the downlink capacity, so no internal contention reduces the cross-sectional throughput below its theoretical maximum. When a fabric is oversubscribed, the effective BW_{bisect} drops by the oversubscription ratio, and every global collective slows proportionally. Section B.4.1 shows how to diagnose the regime where communication, rather than computation, becomes the binding constraint, so that an oversubscribed spine can be recognized as a fabric problem before it is mistaken for slow accelerators.

Definition 3.4: Non-blocking fabric

Non-blocking Fabric is an ML cluster network topology in which any permutation of input-output port pairs can communicate simultaneously at full line rate without internal contention, achieved by ensuring that uplink capacity at every switch tier equals or exceeds downlink capacity.

1. **Significance**: In ML fleets, a non-blocking fabric ensures that AllReduce traffic from any accelerator subset does not compete for shared links, preserving the full BW_{bisect} term for global collectives. A 2:1 oversubscribed spine halves the effective BW_{bisect}, doubling AllReduce time for global gradients and dropping scaling efficiency $\eta_{scaling}$ accordingly—in a 30 percent-communication workload, this costs roughly 23.1 percent of total cluster throughput.
2. **Distinction**: Unlike oversubscribed fabrics common in web data centers, where upper-tier links are shared among many lower-tier nodes, a non-blocking fabric provides dedicated path capacity for every possible pairing of senders and receivers simultaneously.
3. **Common pitfall**: A frequent misconception is that non-blocking means zero congestion. Endpoint congestion (incast) can still occur if multiple senders simultaneously target the same receiver port, regardless of how much internal fabric capacity is available.

Fat-trees build on Clos-style non-blocking network ideas to provide full BW_{bisect} (Clos 1953), rail-optimized networks trade some global flexibility for lower cost, and dragonflies reduce cabling while accepting workload-placement constraints (Kim et al. 2008). This trade-off between guaranteed bandwidth and economic scalability drives every topology decision in large-scale ML clusters; the topology comparison later in this section makes that bandwidth-cost trade-off concrete.

3.5.1 Top-of-rack (ToR) and the failure domain

The Top-of-Rack (ToR) switch serves as the fundamental physical aggregation point, defining both bandwidth limits and the minimum **failure domain** for the cluster. In a high-density AI configuration using standard DGX nodes, a single rack typically houses 4 nodes, each containing 8 GPUs, for a total of 32 accelerators. The ToR switch unites these devices but also creates a critical vulnerability: if the ToR fails, it instantly partitions 32 GPUs from the training job, forcing the global scheduler to halt and recover from the last checkpoint.

To mitigate congestion at this edge, network architects maximize the switch **radix**, the number of ports available. A high-radix switch with 64 ports allocates 32 ports downlink to the servers (ensuring full bandwidth for the 32 GPUs) and 32 ports uplink to the spine. This 1:1 subscription ratio guarantees non-blocking performance at the rack level. For our cluster of 1,024 GPUs, the physical topology comprises approximately 32 such racks.

The same rack boundary also shapes recovery. The job scheduler must be topology-aware for performance, while reliable replica placement must keep redundant state and replacement capacity outside the same rack failure domain so that a single lost rack does not take down the entire training run (see section 7.9.2.3).

3.5.2 Fat-tree (Clos) networks

The ToR provides non-blocking bandwidth within a single rack, but connecting 32 racks into a cluster that preserves full $\text{BW}_{\text{bisect}}$ across every possible communication pair requires a topology that scales cross-sectional capacity with cluster size. The fat-tree (also called a Clos network) achieves this by adding parallel spine switches at each tier, so the aggregate uplink capacity always matches the total edge bandwidth feeding into it. In this vocabulary, leaf switches attach servers or racks, spine switches connect leaves within a pod, and core switches connect multiple pods when a third tier is needed.

> **Definition 3.5: Fat-tree**
>
> ***Fat-Tree*** is a hierarchical ML cluster network topology in which the number of parallel paths (and therefore aggregate cross-sectional capacity) increases at each switch tier toward the spine, providing full $\text{BW}_{\text{bisect}}$ and multiple equal-cost routes between any two nodes (Al-Fares et al. 2008).
>
> 1. **Significance**: A k-ary fat-tree built from radix-k switches supports $k^2/2$ hosts in a two-tier (pod) configuration and $k^3/4$ hosts in a three-tier configuration with full $\text{BW}_{\text{bisect}}$. With $k = 64$, a two-tier pod supports 2,048 GPUs and a three-tier fabric supports 65,536 hosts. Because every AllReduce can use any available spine path, the fabric sustains simultaneous full-rate communication from all accelerators, meeting the $\text{BW}_{\text{bisect}}$ requirement for global gradient synchronization.
> 2. **Distinction**: Unlike a standard tree (where bandwidth at the root is a single bottleneck shared by all leaves), a fat-tree replaces each root with multiple spine switches whose combined uplink capacity matches the total edge bandwidth, eliminating the bottleneck.
> 3. **Common pitfall**: A frequent misconception is that fat-trees guarantee zero network cost. They require $\mathcal{O}(N \log N)$ switches and dense cabling: a non-blocking three-tier fat-tree at the 4,000-GPU scale needs hundreds of switches and tens of thousands of optical cables, costing $20–100 million in switching hardware alone.

Leaf, spine, and core tiers provide multiple equal-cost paths instead of a single oversubscribed root, which is how a fat-tree creates that capacity guarantee (figure 3.8).

The fat-tree[9] is a common default for ML clusters because a non-blocking design can provide full $\text{BW}_{\text{bisect}}$, a nonnegotiable requirement for the AllReduce collective, which demands simultaneous, all-to-all communication. The network is constructed in hierarchical tiers: Leaf switches (ToR) connect directly to servers, Spine switches interconnect all leaves within a locality domain known as a pod, and Core switches bind multiple pods together.

A fat-tree built from switches with radix[10] k supports $N_{\text{hosts}} = k^3/4$ hosts at three tiers, and two distinct two-tier framings, which differ only in whether the upper tier is treated as the cluster's edge or as an aggregation layer below a future core. The first framing counts every switch in the two tiers as line-rate capacity for hosts. With radix-64 switches, the fabric spends half of each leaf's ports on hosts and half on spine uplinks, giving $k^2/2 = 2{,}048$ host ports across the pod. The second framing reserves the upper tier as aggregation that will later uplink to a core layer rather than terminating hosts: each leaf still provides $k/2$ host ports, but the pod now contains only $k/2$ leaves, so it reaches $(k/2)^2 = 1{,}024$ hosts per pod. The two counts are the same switches accounted differently: whether the upper tier terminates as the cluster spine or as a pod aggregation stage. Either framing comfortably accommodates our 1,024 reference cluster with only leaf and spine layers.

A concrete bisection estimate shows how quickly oversubscription turns into synchronization delay.

[9] **Fat-Tree (Clos Network)**: The underlying multi-stage switching theory was invented by Clos (1953) at Bell Labs to minimize the number of electromechanical crosspoints in telephone exchanges. Leiserson (1985) at MIT generalized the concept as the "fat-tree," where the tree is "fat" because link bandwidth increases toward the root, proving it could emulate any network of equal hardware volume. The same cost-minimization logic that drove 1950s telephony now drives ML cluster design: minimize switch count while guaranteeing non-blocking connectivity for global AllReduce.

[10] **Switch Radix**: High-end switches (for example, NVIDIA Quantum-2) can feature a radix of 64 ports, each at 400 Gb/s. This density allows a two-tier fat-tree to support up to 2,048 GPUs with only two switch hops. As model scale pushes toward 100,000 GPUs, increasing switch radix (to 128 or 256), grouping accelerators into larger local domains, or accepting topology constraints are the main ways to avoid adding more tiers and the resulting latency/cost explosion.

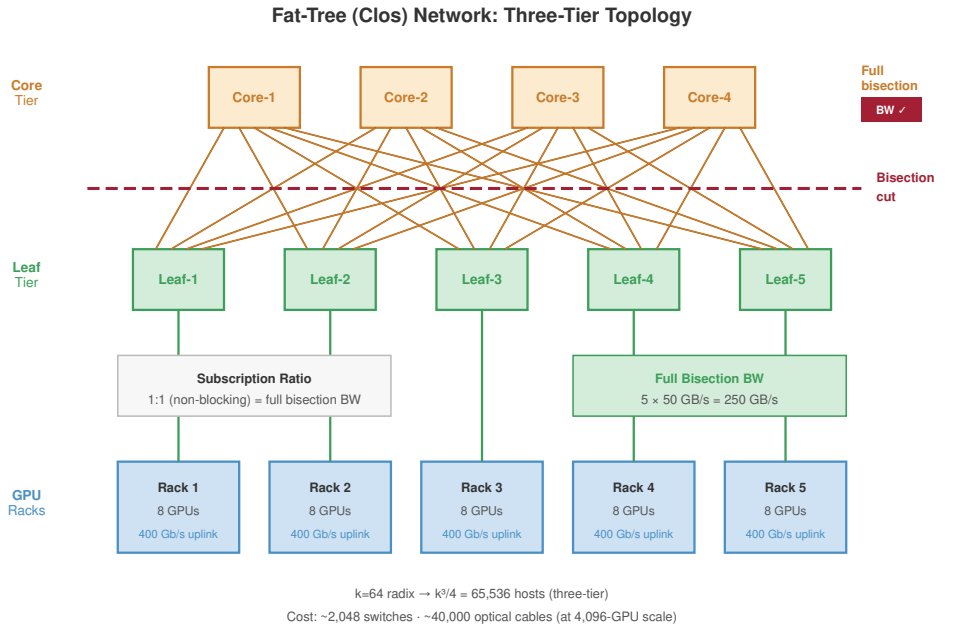


Figure 3.8: Non-Blocking Fat-Tree Topology: A three-tier Clos network built from radix-k switches. By ensuring that the number of uplinks at each level matches the number of downlinks, the topology provides full BW_{bisect} between any two pods. The multiple parallel paths between leaf and spine enable hardware-based adaptive routing to spray packets and avoid congestion.

Napkin Math 3.3: Bisection bandwidth: The cost of oversubscription

Problem: A cluster designer is choosing between a "Non-blocking" (1:1) fat-tree and a "Cost-optimized" (4:1) spine for a cluster of 1024 GPUs. How much slower will a 100 GB-per-GPU AllReduce be on the cheaper network?

Math: Bisection bandwidth (BW_{bisect}) is the minimum pipe diameter between halves of the cluster.

1. **Non-blocking (1:1)**: $BW_{bisect} = 512 \times 50$ GB/s $= 25{,}600$ GB/s per direction.
   - Time: 51,200 GB / 25,600 GB/s $\approx$ 2,000 ms.
2. **Oversubscribed (4:1)**: $BW_{bisect} = 25{,}600$ GB/s divided by 4 = 6,400 GB/s per direction.
   - Time: 51,200 GB / 6,400 GB/s $\approx$ 8,000 ms.

Systems insight: Saving money on core switches creates a 4× bottleneck for global synchronization. On a $300M supercomputer where training is 30 percent communication, that bottleneck wastes approximately $142.1M in idle compute time, matching the bisection-bottleneck model in section 3.5.2. For training, network oversubscription is a false economy.

This oversubscription calculation turns the topology definition into a design rule: the subscription ratio at each tier determines whether a fat-tree can sustain global collectives.

Checkpoint 3.2: Fat-tree topologies

These questions check whether hierarchical switch-fabric trade-offs are clear:

- ☐ **Radix and scale**: Compute the maximum number of GPUs a Radix-64 two-tier fat-tree can connect without core switches.
- ☐ **Non-blocking ratio**: Explain why a Non-blocking fabric requires a 1:1 subscription ratio at every tier of the network.
- ☐ **Tier vs. latency**: Determine how α **(startup latency)** changes when moving from a two-tier to a three-tier fat-tree.
- ☐ **Aggregate vs. bisection**: Distinguish aggregate bandwidth from $\text{BW}_{\text{bisect}}$ and identify the metric that matters for AllReduce.

Scaling beyond this requires a three-tier architecture with core switches, enabling the fabric to reach 65,536 hosts, but the added scale carries both cost and latency. A non-blocking $k = 64$ three-tier tree requires roughly $5k^2/4 \approx 5{,}120$ switches across the core, aggregation, and edge layers; at $10,000 to $50,000 per switch, the switching layer alone represents $50M to $250M. Typical **hop counts** also rise from 2 for intra-pod traffic (leaf-spine-leaf) to 4 for inter-pod traffic (leaf-spine-core-spine-leaf), adding serialization delay and switch processing time to the α latency term across thousands of synchronization steps. Rail-optimized topologies respond to both pressures by matching the physical cabling pattern to the collective communication pattern.

3.5.3 Rail-optimized topology

Rail-optimized topology begins from the communication pattern rather than from a generic switch hierarchy. In dense accelerator nodes, GPUs can be grouped by local slot position into rails; figure 3.9 makes that physical cabling pattern concrete.

The wiring pattern connects corresponding GPUs across nodes (GPU 0 to GPU 0, GPU 1 to GPU 1) through dedicated rail switches rather than a shared ToR switch. This matters because data parallelism creates a deterministic and highly stratified communication pattern that standard topologies fail to exploit. When tensor-parallel groups stay inside each node, each replica assigns the corresponding model shard to the same local GPU rank. A rank is the worker or GPU index assigned by the distributed runtime, so GPU 0 on one node synchronizes that shard's gradients with GPU 0 on other nodes, but rarely with GPU 1. A **rail-optimized topology** physically hardwires this logic by isolating these same-rank communication paths into dedicated networks. Instead of connecting all 8 GPUs in a node to a single ToR switch, the network connects all GPU 0s across the entire cluster to a dedicated "Rail 0" switch fabric, all GPU 1s to "Rail 1," and so on. A simple hop-count estimate shows the payoff.

Napkin Math 3.4: The rail-optimized dividend

Problem: A team is synchronizing per-rank data-parallel gradients across 128 nodes. In a standard fat-tree, each message between same-rank GPUs traverses its source leaf switch, a spine switch, and the destination leaf switch (3 switch traversals). In a rail-optimized network, all corresponding GPUs sit on the same rail switch (one traversal). How much "Latency Dividend" does the rail design earn?

Math: Communication latency (α) is proportional to the number of switch traversals.

1. **Standard latency**: 3 traversals $\times$ 0.6 µs = 1.8 µs.
2. **Rail-optimized**: 1 traversal $\times$ 0.6 µs = 0.6 µs.
3. **Result**: 3× lower latency.

Systems insight: For synchronous data-parallel AllReduce (which the training step waits on every iteration), a 3× latency reduction is the difference between 80 percent and 95 percent scaling efficiency. The scaling efficiency bound (principle 8) explains why this matters: physically

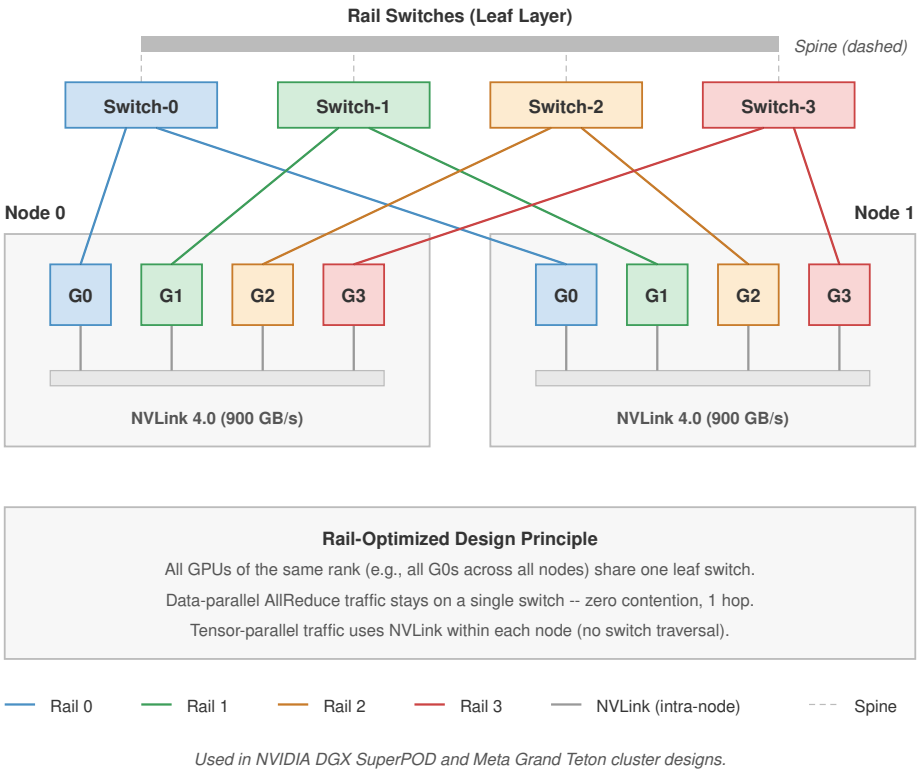


Figure 3.9: Rail-Optimized Physical Wiring: In dense accelerator nodes (for example, DGX H100), GPUs are partitioned into 'Rails' corresponding to their PCIe slot position. GPU 0 in every node connects to Switch Rail 0, GPU 1 to Switch Rail 1, and so on. This architecture ensures that data-parallel AllReduce traffic—which synchronizes between corresponding GPUs—never competes for BW_{bisect} with other parallel groups, minimizing hop count and congestion for the most latency-sensitive traffic.

aligning the network to the model's same-rank traffic pattern eliminates the "Spine Tax" for the most bandwidth-hungry communication. Archetype A clusters achieve their performance because they are structured, not merely large.

That same-rank latency dividend is why the rail pattern appears in large LLM training fleets rather than remaining a cabling optimization. Archetype A workloads exploit this structure directly, because 3D parallelism generates traffic patterns that align with the rail wiring.

Lighthouse 3.1: Archetype A (GPT-4/Llama-3): The rail-optimized fleet

Archetype A's use of 3D parallelism generates two distinct traffic patterns that pull in opposite directions: (1) massive, bandwidth-hungry gradient averaging for data parallelism, and (2) high-frequency, latency-sensitive activation exchanges for tensor parallelism. The rail-optimized design ensures that data-parallel traffic traverses only a single switch hop between nodes, minimizing the latency that would otherwise stall the synchronous training loop.

The engineering consequence of this alignment is measurable at cluster scale. Because each of the 8 GPU ranks in a node communicates only with the same rank on other nodes during data-parallel AllReduce, the rail topology partitions the cluster into 8 independent switch networks that can operate concurrently without contention.

For our cluster of 1,024 GPUs spanning 128 nodes, this creates 8 parallel networks of 128 GPUs each, allowing per-rank data-parallel AllReduce traffic to traverse a single switch hop rather than the multi-hop leaf-spine-leaf path required by a standard fat-tree. The latency benefit is significant for the frequent, bandwidth-hungry gradient exchanges that synchronous data parallelism demands. However, this architecture introduces a sharp trade-off: traffic that must reach GPUs of different ranks (such as expert routing in MoE, or pipeline-stage handoffs that do not align with the rail wiring) requires bridging across rails. Large clusters therefore often employ a hybrid approach, using rail-optimized leaves for same-rank data-parallel traffic while bridging the rails with a full fat-tree spine to support cross-rank communication patterns.

Checkpoint 3.3: Rail-optimized networks

These questions check whether workload-specific network-design trade-offs are clear:

- ☐ **Rail beneficiary**: Identify the dimension of 3D Parallelism (TP, PP, or DP) that benefits most from a Rail-Optimized design.
- ☐ **Switch hops**: Explain why a rail-optimized network reduces the number of switch hops for gradient exchanges compared to a standard fat-tree.
- ☐ **Traffic isolation**: Explain the "Traffic Isolation" benefit of placing all GPU 0s on one rail and all GPU 1s on another.
- ☐ **Core switch need**: Classify the claim as true or false: a rail-optimized network eliminates the need for high-bandwidth core switches connecting different rails.

3.5.4 Dragonfly and torus alternatives

A **Dragonfly**[11] topology organizes switches into high-radix groups, where each group functions as a fully connected island. These groups connect to one another via global optical links. The hierarchical structure minimizes the number of expensive long-distance cables required to scale, reducing cabling cost by roughly 50 percent compared to a fat-tree. However, bandwidth within a group is non-blocking (100 percent), while bandwidth between groups is typically only 25–50 percent of the aggregate injection rate. For ML workloads, this oversubscription creates a binary performance cliff based on job placement. A training job that fits entirely within a single dragonfly group sees full line-rate performance; a job spanning multiple groups is throttled by the limited global bandwidth, potentially suffering a 2–4× slowdown. Dragonfly fabrics therefore require topology-aware schedulers that rigidly pack jobs into groups to avoid crossing the oversubscribed global links.

A **Torus** topology connects each node directly to its neighbors in a multidimensional grid, most commonly a 3D torus where every node links to its six adjacent peers (up/down, left/right, front/back). The design offers full local bandwidth with minimal switching hardware, as connections travel only 1–2 hops to reach neighbors. However, global communication requires traversing the diameter of the mesh ($\mathcal{O}(N^{1/3})$ hops), making latency scale poorly with cluster size. Google adopted this architecture for its Tensor Processing Unit (TPU) pods because Transformer training is dominated by data parallelism and pipeline parallelism, both of which use nearest-neighbor communication patterns that map naturally onto the physical grid. The limitation becomes apparent with Mixture-of-Experts (MoE), which relies on AllToAll communication patterns. On a torus, these random permutations congest the limited $\text{BW}_{\text{bisect}}$ of the mesh, causing performance to degrade by 2–4× compared to a switch-based fat-tree.

The trade-offs between these topologies become stark when quantified for a large-scale deployment. Consider a 4,096-GPU cluster. A non-blocking fat-tree at this scale, built from the $k = 64$ switches assumed throughout this chapter, requires hundreds of switches (two 2,048-GPU pods plus a bridging core layer) and tens of thousands of optical cables to deliver 100 percent $\text{BW}_{\text{bisect}}$, enabling any GPU to communicate with any other at full speed. A 3D torus connecting the same nodes might use zero external switches (relying on direct host-to-host links) and only short copper cables, but offers only a fraction of $\text{BW}_{\text{bisect}}$ (scaling with $N^{2/3}$). The architecture choice follows from workload regularity. Google's TPU Pods have used torus topologies because their workloads,

[11] **Dragonfly Topology**: Introduced by John Kim and William Dally at ISCA 2008, the dragonfly uses high-radix routers grouped into fully connected "super-nodes" to reduce global cabling by 52 percent compared to a folded Clos of equivalent scale. The cost savings are real but create a rigid scheduling constraint: any training job that crosses group boundaries hits the oversubscribed global links, making topology-aware placement mandatory rather than optional.

primarily transformer training, are predictable and the structured grid efficiently supports collective communication algorithms (ring AllReduce, AllGather, ReduceScatter) that map onto the 3D mesh. General-purpose GPU clusters often favor fat-trees because their workloads are more diverse, ranging from recommendation systems to graph neural networks, and rely heavily on global AllReduce patterns that require the full BW_{bisect} only a tree can provide. A torus saves millions in switch costs but rigidly constrains the software; a fat-tree costs more but provides the universality needed for general-purpose AI research.

Figure 3.10 quantifies the bandwidth and cost differences between these common families. The Butterfly bar represents a multistage low-diameter switching network: it uses fewer stages than a full fat-tree, but its structured paths offer less bisection bandwidth for arbitrary ML collectives.

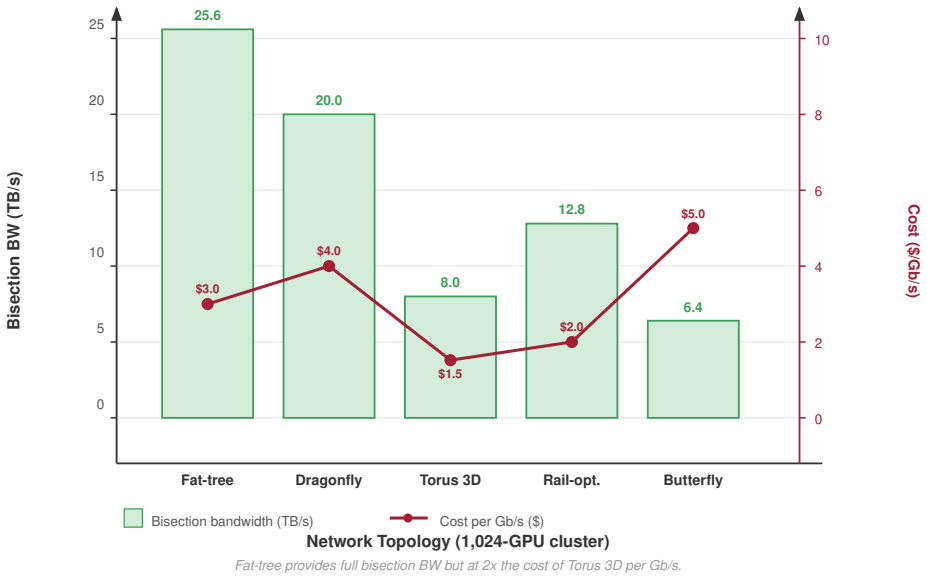


Figure 3.10: Network Topologies for ML: Bandwidth vs. Cost: Bar chart comparing BW_{bisect} (TB/s, left axis) and cost per Gb/s (right axis) for five topologies on a 1,024-GPU cluster: Fat-tree (25.6 TB/s, \$3.0/Gb/s), Dragonfly (20.0, \$4.0), Torus 3D (8.0, \$1.5), Rail-opt (12.8, \$2.0), Butterfly (6.4, \$5.0). Fat-tree provides full BW_{bisect} but at 2× the cost of Torus 3D per Gb/s.

Bisection analysis alone assumes a single workload type. In practice, clusters serve multiple workloads with different communication patterns, and topology selection must balance their competing demands.

Checkpoint 3.4: Topology selection for your workload

The choice of network topology dictates the upper bound of training efficiency. Warm up by matching each single-workload pattern to its ideal topology, then design a fabric that must serve several at once.

Single-workload picks

- ☐ **AllReduce bandwidth**: Select the topology that maximizes BW_{bisect} for a standard data-parallel job dominated by AllReduce.
- ☐ **MoE traffic**: Select the topology that minimizes hop count while maintaining congestion control for a Mixture-of-Experts model with AllToAll-dominant, bursty, sparse communication.

- ☐ **Mesh locality**: Select the topology that offers the best locality for a 2D mesh computation by mapping physical neighbor links directly to hardware.

Designing for a mixed cluster

You are designing the network for a new ML cluster that will run two primary workloads: (1) training a 175B-parameter language model using 3D parallelism (tensor, pipeline, and data parallelism), and (2) serving a Mixture-of-Experts model that relies heavily on AllToAll communication to route tokens to the correct experts.

- ☐ **Training topology**: Select the topology (fat-tree, rail-optimized, or hybrid) that best matches tensor parallelism within groups of 8 GPUs and data parallelism across all nodes, then justify the choice.
- ☐ **MoE topology**: Revise the topology choice for unpredictable AllToAll traffic where any GPU may need to communicate with any other GPU.
- ☐ **Oversubscription risk**: Identify which workload suffers more under 2:1 spine-layer oversubscription and propose a scheduling strategy to mitigate the impact.

Topology provides the structural capacity for our cluster of 1,024 GPUs to communicate, but structure alone does not guarantee performance. When all 1,024 GPUs simultaneously inject 350 GB of gradient traffic into the fabric, that theoretical capacity collides with the reality of **Fabric Behavior**: congestion control and routing dynamics determine whether this traffic flows smoothly or gridlocks under the strict synchronization of the BSP model.

3.6 Level 4: Fabric Behavior (Congestion, Routing)

Real fabrics deviate from theoretical full BW_{bisect} due to congestion, and the impact of this congestion is qualitatively different in ML clusters than in general-purpose networks. The BSP barrier introduced earlier is what makes the difference: web traffic is stochastic and asynchronous, so one user's 50 ms delay does not penalize the thousands of others, but under BSP the slowest flow in the fabric dictates the iteration time for the entire cluster. If a single link out of 10,000 becomes congested and doubles its latency, the effective throughput of the entire supercomputer drops for that step. In this synchronized regime, tail latency is the dominant performance constraint, not an outlier metric.

Definition 3.6: Bulk synchronous parallel (BSP)

Bulk Synchronous Parallel (BSP) is a parallel execution model in which every worker completes a local computation phase, exchanges data with all other workers, and then waits at a global barrier before any worker begins the next phase—making the slowest participant the pacing constraint for the entire cluster.

1. **Significance**: BSP makes system efficiency $\eta_{scaling}$ directly proportional to the slowest worker: if one GPU in a 1,024-GPU cluster runs 10 percent slower due to thermal throttling or network jitter, the barrier stalls the remaining 1,023 GPUs for that fraction of the step, wasting effectively 102 GPU-steps of compute per iteration. At $3/GPU-hour, a 5 percent straggler gap across a $50M training run wastes roughly $2.5M in idle accelerator time.
2. **Distinction**: Unlike asynchronous parallelism, which allows workers to proceed with stale weights from previous steps, BSP enforces a global state update at every barrier, providing mathematical equivalence to single-device training and predictable convergence behavior.
3. **Common pitfall**: A frequent misconception is that BSP is simply inefficient compared to async models. Asynchronous training often requires more total steps to converge because stale gradient updates introduce noise—in practice, BSP with careful straggler mitigation typically reaches the same loss in fewer wall-clock hours than async alternatives.

This is why **tail latency (P99)**, not average throughput, is the metric that governs a training fabric: one congested switch port stalls the barrier, and the barrier stalls the cluster. The mechanisms that keep tail latency bounded therefore become the central concern of fabric behavior.

3.6.1 Priority flow control (PFC)

The BSP weakest-link property means that even a single dropped packet can trigger a retransmission timeout that stalls the global barrier for milliseconds, an eternity in a training loop where each compute step finishes in tens of milliseconds. RoCEv2 fabrics mitigate this with priority flow control (PFC), operating in lossless mode: rather than allowing switch buffers to overflow and drop packets, the fabric uses a link-layer backpressure mechanism to pause upstream senders before any buffer saturates (Guo et al. 2016; Gangidi et al. 2024).

Definition 3.7: Priority flow control

Priority Flow Control (PFC) is a link-layer mechanism used by RoCEv2 ML cluster fabrics to prevent switch buffer overflow by sending PAUSE frames to an upstream sender when a port's queue depth crosses a configured threshold, throttling injection on a per-priority basis without dropping packets.

1. **Significance**: PFC is the foundation for lossless Ethernet required by RoCEv2 RDMA. A PFC PAUSE frame must reach the upstream sender within one round-trip time (roughly 1–5 µs at switch-to-switch distances) before the buffer overflows. In a 1,000-node cluster, a single slow receiver can trigger PAUSE frames that propagate 3–5 hops upstream within 10–50 ms, halting gradient traffic across thousands of unrelated GPU pairs and collapsing cluster throughput to near zero.
2. **Distinction**: Unlike standard IEEE 802.3 flow control, which pauses all traffic classes on a link, PFC operates per priority class, allowing latency-sensitive control messages to continue flowing while only pausing the congested gradient data queue.
3. **Common pitfall**: A frequent misconception is that PFC solves congestion. It transforms packet loss into congestion spreading: a backpressure cascade can freeze the entire fabric within 200 ms of a single faulty transceiver, because no mechanism limits how far PAUSE frames propagate across the network.

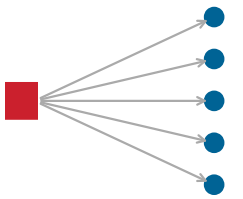
The danger of PFC lies in its cascading nature. When a switch port's buffer fills, it sends a PAUSE frame upstream, which causes that switch's buffers to fill, which triggers another PAUSE frame further upstream. In theory, this backpressure should throttle the source. In practice, a single slow receiver can propagate pauses across the entire fabric in milliseconds, freezing links that have no direct relationship to the original congestion point. This cascading behavior, known as **congestion spreading** or **victim flows**, is the primary operational risk of PFC-based lossless Ethernet. The root cause is incast, a many-to-one traffic pattern inherent to distributed synchronization, as illustrated in figure 3.11; this deterministic overload is what triggers the PFC backpressure cascades.

A production incident shows how this cascade can freeze a cluster.

War Story 3.1: Microsoft's PFC deadlock (2016)

Context: A team led by Chuanxiong Guo at Microsoft Research deployed RoCEv2 across Microsoft's data centers—a fleet operating at the scale of tens of thousands of servers—to support latency-sensitive services. RoCEv2 requires a lossless fabric, so the network used Priority Flow Control (PFC), and the team developed a DSCP-based PFC mechanism to extend the deployment beyond VLAN boundaries (Guo et al. 2016). DSCP marks packet priority, VLANs define layer-2 traffic boundaries, ARP resolves an IP address to a MAC address, and switches learn MAC-to-port forwarding entries.

Failure mode: PFC interacted with Ethernet flooding to create a cyclic buffer dependency. In Microsoft's reported example, a dead server with an incomplete ARP/MAC mapping caused



One paused receiver can freeze unrelated flows.

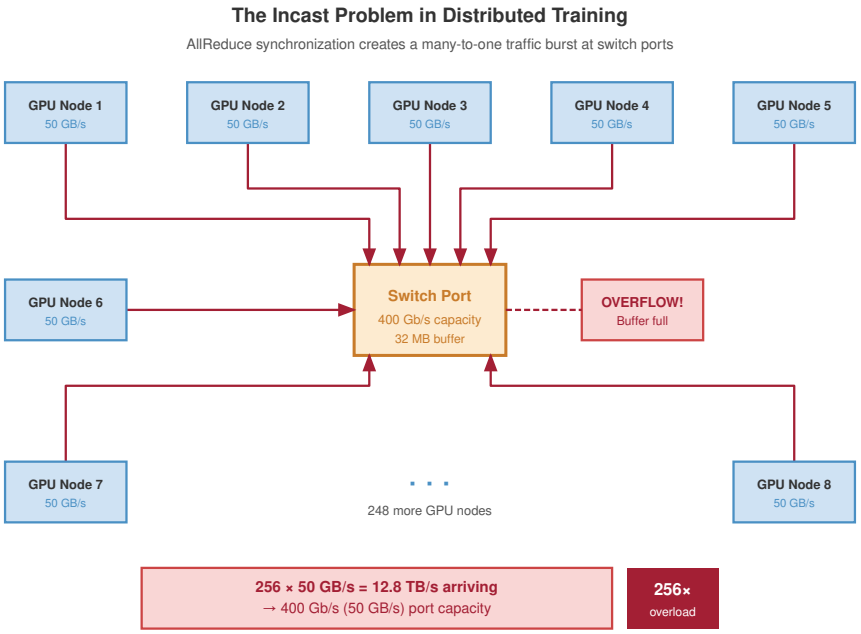


Figure 3.11: The Incast Problem in Distributed Training: During AllReduce synchronization, every GPU node sends line-rate traffic to a common aggregation switch port, producing a many-to-one traffic burst. The figure shows 256 GPU nodes each at 50 GB/s converging on a switch port with 400 Gb/s (50 GB/s) capacity and a 32 MB buffer: 12.8 TB/s of offered load against 50 GB/s of capacity oversubscribes the egress port by 256×. This deterministic overload is the root cause of the PFC backpressure cascades described later in this section.

upstream switches to flood lossless packets to every port; PFC PAUSE frames then propagated upstream in response to the flooded traffic, eventually forming a closed loop of paused buffers among switches.

Consequence: Once the PFC deadlock formed, restarting servers did not clear it. Microsoft's fix was architectural: block broadcast and multicast traffic from entering lossless traffic classes, and drop lossless packets when the corresponding ARP entry is incomplete rather than fall back to flooding.

Systems lesson: PFC congestion spreading is a structural vulnerability of lossless Ethernet at hyperscale. RoCE fabrics require packet-class discipline, PFC pause telemetry, and hardware-level safeguards because a mechanism designed to avoid packet loss can itself become the failure mode. ML training fabrics built on RoCE inherit this risk directly: a single stale ARP entry can cascade into a PFC storm that stalls a multi-thousand-GPU training run, with the cluster appearing healthy while no gradient flows between workers.

A simple probability calculation shows that such incidents are not rare edge cases but recurring operational risks at scale.

Napkin Math 3.5: The probability of a PFC storm

Problem: A 4096-GPU RoCE cluster is in operation. If the probability of a single transceiver degrading and triggering PFC pauses is just 0.001 percent per day, what is the chance of a cluster-wide "PFC Storm" occurring in a given day?

Math: In a lossless Ethernet fabric, one bad link can pause its neighbor, which pauses its neighbor, eventually freezing the entire tree.

1. **Total exposure**: 4096 × 3 link tiers = 12,288 links, or 24,576 transceivers.
2. **Probability of zero transceiver failures**: $\Pr(\text{safe}) = (1-0.00001)^{24,576} \approx 0.782$.
3. **Probability of at least one storm**: $\Pr(\text{storm}) = 1-0.782 = \mathbf{0.218}$ (about 21.8 percent).

Systems insight: Even with "five-nines" reliable components, a large-scale RoCE cluster faces a 21.8 percent daily risk of a fabric-wide freeze. That is not guaranteed to happen every day, but it is frequent enough to be a weekly-scale operational concern. RoCE deployments therefore require aggressive PFC watchdogs and hardware-level telemetry, because at scale, "rare" link degradations become recurring operational realities that must be handled automatically.

3.6.2 Proactive congestion control: DCQCN and HPCC

To avoid the blunt instrument of PFC pauses, high-performance fabrics rely on proactive congestion control to modulate injection rates before buffers overflow. In BSP workloads, proactive control is a *latency requirement*, not an *optional throughput optimization*: if a single packet is delayed by a congested switch queue, the entire cluster must wait for that straggler to complete the synchronization step. The tail latency of the network effectively becomes the average step time of the training job.

The first widely deployed solution, **DCQCN**[12], operates as a reactive feedback loop using Explicit Congestion Notification (ECN) (Zhu et al. 2015). When a switch's queue depth exceeds a configured threshold, it marks the ECN bit in the packet header. The receiver echoes this mark back to the sender via a Congestion Notification Packet (CNP), prompting the sender to reduce its injection rate using a multiplicative-decrease algorithm. DCQCN is widely supported, but production AI collectives expose a tuning problem: ECN thresholds and NIC firmware behavior can trade off PFC avoidance, throughput, visibility, and tail latency rather than offering one stable optimum (Gangidi et al. 2024).

A more precise alternative, **HPCC**[13], addresses this opacity by using In-Network Telemetry (INT) (Li et al. 2019). Instead of a simple bit mark, switches append precise metadata to every packet header: current queue depth, link utilization, and timestamps. The sender receives a full dashboard of the network state, allowing it to calculate the exact allowable transmission rate within a single round-trip time. By reacting to precise telemetry rather than binary signals, HPCC reduces queue buildup and tail-latency variance in the evaluated incast-heavy workloads. The primary trade-off is hardware support: DCQCN functions on standard ECN-capable Ethernet switches, whereas HPCC requires programmable switches capable of pushing INT metadata at line rate.

3.6.3 Adaptive routing

Static routing protocols like ECMP[14] (Equal-Cost Multi-Path) distribute traffic by hashing flow headers to fixed paths. While statistically sound for millions of small web requests, this approach fails for the elephant flows typical of ML training. Consider a scenario with 4 equal-cost paths and 8 large gradient flows. A perfect distribution would place 2 flows on each link. However, static hashing frequently results in collisions where one link carries 4 flows while another sits idle. In a synchronized training step, the completion time is dictated by the overloaded link, effectively halving the network's useful bandwidth.

Adaptive routing mitigates this problem by allowing switches to dynamically select the output port based on real-time queue depth rather than a static hash. The implementation is protocol-dependent: InfiniBand fabrics perform packet-level adaptive routing, spraying[15] individual packets across all available lanes because the hardware transport guarantees in-order delivery at the destination.

Ethernet fabrics typically employ *flowlet switching*, rerouting bursts of packets only when a sufficient time gap is detected, to avoid the performance penalties associated with packet reordering. This mechanism becomes essential for Mixture-of-Experts (MoE) models. Unlike the predictable Ring AllReduce pattern, MoE models use AllToAll communication, where every GPU sends data to every other GPU containing active experts. For a 175B-parameter model using expert parallelism with 64 experts, this generates a dense 64×64 traffic matrix—4,096 simultaneous flows. Under static

12 **DCQCN (Data Center Quantized Congestion Notification)**: Introduced at SIGCOMM 2015 by Microsoft and Mellanox, DCQCN was designed for large-scale RDMA fabrics (Zhu et al. 2015). Its binary ECN/CNP feedback loop is simpler to deploy than telemetry-rich schemes, but it can become a tuning surface for training collectives: Meta's SIGCOMM 2024 RoCE study reports that tighter ECN thresholds could reduce PFC in some settings while hurting collective completion time, and that 400G deployments exposed firmware and visibility issues (Gangidi et al. 2024).

13 **HPCC (High Precision Congestion Control)**: Developed by Alibaba and presented at SIGCOMM 2019, HPCC replaced binary ECN with per-packet in-network telemetry and reported large reductions in flow completion time under evaluated incast-heavy workloads. The trade-off is hardware dependency: HPCC requires programmable switch ASICs capable of appending telemetry metadata at line rate, limiting deployment to clusters with newer switching silicon (Li et al. 2019).

14 **ECMP (Equal-Cost Multi-Path)**: ECMP selects a path by hashing on the 5-tuple (source/destination IP, source/destination port, protocol), which means the same flow always takes the same path. This determinism is a feature for packet ordering but a liability for ML: because a single AllReduce ring produces a small number of persistent flows, hash collisions are not transient statistical events but permanent hot spots that persist for the entire training run.

ECMP, hash collisions are statistically guaranteed to create stragglers; adaptive routing is required to distribute this traffic evenly across the fabric's BW_{bisect}.

3.6.4 The incast problem in ML

Congestion control and adaptive routing manage flows that traverse the interior of the fabric. A different failure mode occurs at the edge, where the traffic pattern itself overwhelms a single port regardless of how much capacity the fabric interior provides.

Definition 3.8: Incast

Incast is a many-to-one ML cluster traffic pattern in which a large number of senders simultaneously transmit data to a single receiver port, concentrating line-rate traffic from multiple sources into a single switch queue and causing buffer overflow even when the rest of the fabric is uncongested.

1. **Significance**: In the reduce phase of AllReduce, every participating GPU simultaneously sends gradients toward the same aggregation points. With 256 senders each at 50 GB/s targeting one switch port, the instantaneous incast reaches 12.8 TB/s, orders of magnitude above a 400 Gb/s port's absorption capacity. A 32 MB switch buffer holds about 640 µs of one 400 Gb/s egress stream, but under 256× incast the net fill rate is roughly 12.75 TB/s after subtracting the single 50 GB/s egress port, so overflow arrives in about 2.5 µs. That sudden overflow triggers either PFC cascade or packet drops that elevate L_{lat} cluster-wide.
2. **Distinction**: Unlike general congestion (which occurs on shared internal links when aggregate traffic exceeds link capacity), incast is an endpoint bottleneck: it occurs even if every internal spine and leaf link is completely uncongested, because the bottleneck is the single destination port, not the fabric interior.
3. **Common pitfall**: A frequent misconception is that incast is rare or unpredictable. Because ML training synchronizes all GPUs at each backward pass, incast is a deterministic event at the end of every layer's gradient computation—it occurs hundreds of times per training step and must be architecturally mitigated through layer-staggered AllReduce or tree-based collective algorithms, not treated as an edge case.

ML training is structurally susceptible to incast because of its synchronized communication patterns. When a layer finishes backward computation, thousands of nodes simultaneously initiate AllReduce, targeting the same switch ports. In our 175B model training across 1,024 GPUs, each AllReduce involves every node injecting data simultaneously, creating a burst that can momentarily exceed the fabric's capacity at specific switch ports. Production clusters mitigate this through three complementary techniques:

- **Layer-staggering**: Trigger each gradient bucket's AllReduce as soon as backpropagation produces it. Final-layer gradients reduce while earlier layers are still computing, so communication is staggered instead of released as one burst.
- **Algorithm selection**: Choose collectives that limit the fan-in at any single port. Hierarchical and rail-local reductions aggregate gradients in stages so that no switch port must absorb all senders at once, in contrast to a flat reduction in which every node targets one aggregation point. Bandwidth-optimal ring AllReduce is itself point-to-point and creates no single aggregation hotspot; the incast pressure comes from many such flows converging on oversubscribed ports, which topology-aware placement and rail-optimized wiring relieve.
- **Quality of Service (QoS) classification**: Tag gradient traffic with the highest service class so background storage or management traffic does not delay the synchronization path.

Together, these mitigations spread the burst in time, space, and priority, but they do not eliminate congestion or tail latency. The question becomes how to build end-to-end clusters that perform despite these realities.

[15] **Packet Spraying**: Unlike standard Ethernet (which hashes flows to single paths to avoid reordering), packet spraying sends individual packets of a single flow across multiple paths. This can improve BW_{bisect} for AllReduce elephant flows when the transport can tolerate or repair reordering. UEC's Ultra Ethernet Transport design explicitly includes multi-path packet spraying and flexible ordering to reduce the ECMP collision problem for AI/HPC traffic (Ultra Ethernet Consortium 2025).

3.7 Level 5: Cluster Design and Case Studies

The final level of the stack is cluster design, where wires, transport, topology, and congestion control combine into a coherent system. The goal of cluster design is to provide an end-to-end Gradient Bus that makes thousands of distributed GPUs feel like a single machine. At this level, the abstraction layers collapse into concrete engineering decisions: which cables to buy, how to wire the racks, which protocol to deploy, and how to validate that the resulting fabric actually delivers the bandwidth the training job expects. Two representative large-scale architectures, one built on InfiniBand and one on Ethernet, illuminate the trade-offs that define ML infrastructure (NVIDIA 2023; Meta Engineering 2024; Gangidi et al. 2024).

3.7.1 The GPU-to-GPU "gradient bus"

In a well-designed cluster, the network fabric acts as a scheduler-aware extension of the system bus, not a passive pipe. The intra-node (NVLink) bandwidth is ~9× higher than inter-node (InfiniBand/RoCE) bandwidth, and the primary mechanism for hiding this cliff is **communication-computation overlap**. During the backward pass, gradients for the final layers are computed first. Instead of waiting for the entire backward pass to finish, the system triggers an asynchronous AllReduce for these gradients immediately while the GPUs continue computing gradients for earlier layers. If the backward pass requires 500 ms of computation and the AllReduce takes 300 ms, perfect overlap masks the entire communication cost behind computation, reducing the effective overhead to $\max(0, 300 - 500) = 0$. In practice, dependency chains and resource contention limit this efficiency to 60–80 percent, leaving a **last-mile problem**: the gradients for the very first layer (computed last) have no subsequent computation to hide behind, exposing their raw transfer time to the critical path.

The bandwidth hierarchy plotted in figure 3.2 dictates the parallelism strategy to mitigate this cliff. Tensor parallelism, requiring massive bandwidth for frequent activation exchanges, is confined to the NVLink domain within a node. Pipeline parallelism, involving point-to-point transfers of activations between pipeline stages, spans the InfiniBand links between nodes. Data parallelism, tolerant of lower bandwidth through gradient accumulation and overlap, stretches across the full fabric.

3.7.2 Case study: NVIDIA DGX SuperPOD

The NVIDIA DGX SuperPOD architecture connects DGX H100 nodes using an NDR InfiniBand network, serving as a concrete implementation of the Gradient Bus concept (NVIDIA 2023). Each node acts as a dense compute island, with eight H100 GPUs connected via NVSwitch to provide 900 GB/s of internal bandwidth. Externally, each GPU pairs with a ConnectX-7 NIC delivering 400 Gb/s of injection bandwidth. Across a standard scalable unit of 32 nodes (256 GPUs), this yields an aggregate injection bandwidth of 12.8 TB/s (256×50 GB/s), ensuring the fabric can ingest gradients as fast as the accelerators produce them.

This architecture explicitly instantiates the five-level model constructed in this chapter. At Level 1, it minimizes latency by using passive copper (DAC) within the rack and active optics only for spine connections. At Level 2, it relies on InfiniBand's native credit-based flow control to guarantee a lossless medium without the fragility of Ethernet PFC. Level 3 implements a rail-optimized fat-tree: GPU rails are aligned across a scalable unit so same-rail traffic stays close and cross-rail traffic traverses the spine layer. At Level 4, hardware-based adaptive routing can select among spine paths to improve bisection behavior for cross-rank communication. At Level 5, the design is modular: multiple SuperPOD scalable units connect through additional switching to form larger clusters (NVIDIA 2023). For our 175B-parameter model, the physical infrastructure would consist of approximately four SuperPOD scalable units wired together, allowing about 1,024 GPUs to function as a single synchronous instrument.

3.7.3 Case study: Meta Grand Teton

Meta disclosed two Grand Teton-based 24,576-H100 clusters: one using RoCE over an Arista/OCP Ethernet fabric and one using NVIDIA Quantum-2 InfiniBand, both with 400 Gb/s endpoints. Meta used both for Llama 3 training and reported ongoing Llama 3 training on the RoCE cluster (Meta Engineering 2024). The primary motivation for Ethernet is operational scale and supply chain

resilience: by using Ethernet, Meta can source switches from multiple vendors and use the same optical infrastructure and management tooling shared by their front-end serving fleet, avoiding the operational silo of a dedicated InfiniBand island.

Making Ethernet perform like a dedicated HPC fabric at this scale requires significant engineering at Level 4. Meta's RoCE study describes PFC watchdogs for long-duration pause events, iterative routing designs beyond plain ECMP, future-oriented flowlet switching experiments, and a 400G deployment experience in which DCQCN tuning proved difficult enough that Meta proceeded without DCQCN while relying on PFC plus higher-layer collective controls (Gangidi et al. 2024). While this architecture achieves high line-rate bandwidth for large gradient transfers, it accepts a trade-off: small-message latency can remain higher than InfiniBand because RoCE deployments rely on Ethernet buffering, QoS, PFC/ECN/DCQCN tuning, routing policy, and switch/NIC implementation choices rather than InfiniBand's native credit-based fabric semantics. For giant models where bandwidth dominates, this is an acceptable exchange; for latency-sensitive MoE routing, the penalty requires careful algorithmic compensation.



Systems Perspective 3.3: InfiniBand vs. RoCE: The industry verdict

The coexistence of InfiniBand (NVIDIA DGX SuperPOD) and RoCE (Meta Grand Teton, Google) in production reflects a genuine trade-off rather than a clear winner. InfiniBand can provide lower tail latency and simpler lossless configuration. RoCE can provide lower switch costs and multi-vendor flexibility. For *training runs* where iteration time is measured in seconds, the latency difference is often absorbed into the noise. For *inference serving* with tight SLOs, the latency difference may matter.

The design space is converging. NVIDIA's Spectrum-4 Ethernet switches incorporate InfiniBand-inspired adaptive routing and congestion control. Broadcom's Memory DCS chips add hardware support for RDMA-optimized switching. The distinction between the two ecosystems is narrowing, though it has not disappeared.

These case studies close the five-level stack by showing how the physical wire, transport protocol, topology, and congestion controls become one production fabric. The remaining operational problem is not how to build a single fast cluster, but how to share that expensive substrate across teams and workloads without sacrificing the predictable performance that training demands.

3.8 Network Virtualization

Production ML clusters are rarely dedicated to a single training job, creating a massive economic imperative for efficient multi-tenancy. A $300 million supercomputer that sits 30 percent idle because it cannot securely isolate concurrent workloads represents a $90 million waste of capital. To recover this utility, the network must support virtualization along three orthogonal dimensions: **bandwidth partitioning** (guaranteeing minimum throughput), **latency determinism** (preventing head-of-line blocking), and **security isolation** (preventing memory snooping between tenants). Technologies like SR-IOV (Single Root I/O Virtualization) and virtual lanes decouple the training job from the physical wire, much as hypervisors decoupled the operating system from the CPU. For our 175B model, this means the training job can reliably consume 80 percent of the cluster's BW_{bisect} while a high-priority inference service and a background data preprocessing job share the remaining 20 percent, with the fabric enforcing hard boundaries that prevent the preprocessor's bursty traffic from stalling gradient updates.

3.8.1 SR-IOV: Hardware NIC virtualization

Cloud providers can deliver near bare-metal RDMA performance to virtualized GPU instances through **Single Root I/O Virtualization (SR-IOV)**, a standard that allows a physical NIC to present itself as multiple independent **Virtual Functions (VFs)**. Each VF has its own hardware queues, doorbell registers, and DMA mappings. By assigning a dedicated VF to each VM or container, the hardware creates a direct path for DMA operations that bypasses the host kernel and hypervisor completely. The passthrough architecture is critical for ML training because it can reduce virtualization overhead to low levels, often within a few percent of bare metal when the NIC, hypervisor,

and placement policy are configured correctly. The cluster-management consequence is immediate: VFs become allocatable network resources, so placement logic must reserve NIC capacity and QoS policy alongside GPUs rather than treating the network as an unbounded shared pool.

However, hardware-level isolation still needs explicit bandwidth policy. SR-IOV exposes multiple VFs that share the NIC's physical resources; administrators can add per-VF or per-group QoS policies to cap or guarantee bandwidth, such as assigning eight VFs 50 Gb/s each on a 400 Gb/s NIC. For our 175B-parameter model training on a multi-tenant cluster, that configured partitioning can prevent a neighboring tenant from stealing bandwidth, but it also enforces a hard ceiling on peak throughput. The training job must be architected to operate within this slice, as no amount of software optimization can burst beyond the configured VF limit.

3.8.2 Traffic isolation and quality of service

Consider a worst-case contention scenario on a shared cluster: our 175B model is midway through a latency-sensitive AllReduce operation when a neighboring job initiates a massive checkpoint save. Without strict isolation, a bursty 100 GB write (taking 2 seconds at full 400 Gb/s line rate) could saturate the shared spine links, introducing queuing delays that increase the AllReduce time by 50 percent or more. To prevent this noisy-neighbor effect, high-performance fabrics rely on **Quality of Service (QoS)** mechanisms that enforce fairness at the packet level.

The primary tool is the Virtual Lane (VL) in InfiniBand (or **Traffic Class** in RoCE), which provides up to 16 independent logical channels on a single physical link. By mapping different traffic types to separate VLs, the network ensures that a saturation event in one lane does not block progress in another. Each VL maintains its own independent credit-based flow control: if the storage traffic for the checkpoint fills up its buffer, the switch pauses only that specific lane. Gradient updates, tagged with a high-priority service level, continue to flow through their reserved lane unimpeded. On the Ethernet side, **Enhanced Transmission Selection (ETS)** provides analogous bandwidth guarantees per traffic class, while advanced switch ASICs can partition their forwarding tables and buffer pools into isolated **network slices**, ensuring that congestion in one tenant's slice cannot trigger PFC pauses in another's.

Virtualization solves the sharing problem but makes performance diagnosis harder. When a training job slows down on a multi-tenant cluster, the cause could be a physical link degradation, a noisy neighbor exceeding its bandwidth allocation, or a misconfigured QoS policy. Systematic monitoring is essential to distinguish these cases.

3.9 Monitoring and Debugging

Network performance problems in ML clusters are insidious because they manifest as **silent waste** rather than explicit failures. A degraded transceiver causing a 10 percent reduction in effective bandwidth might slow each training iteration by only 2–3 percent, a drift easily masked by the natural variance of checkpointing or data loading. Over a 30-day training run on 1,024 GPUs, this invisible drag accumulates to roughly 15,000–22,000 wasted GPU-hours, burning about $45,000–$66,000 at $3/GPU-hour without triggering a single alarm. Traditional IT monitoring tools like SNMP or ICMP ping measure connectivity, not the sustained throughput required by RDMA. Effective observability requires a three-layer approach: **physical monitoring** (FEC errors, signal attenuation), **transport monitoring** (PFC pause frames, retransmission rates), and **application monitoring** (NCCL algorithmic bandwidth) (Jeaugey 2017; Gangidi et al. 2024). Only by correlating signals across these layers can operators detect that a "slow training run" is actually caused by a single degraded cable in one rack.

3.9.1 Link-level telemetry

At the physical layer, hardware counters answer whether the wire still behaves like the model assumed. Every RDMA NIC and switch maintains these counters, accessible via the `perfquery` utility (Linux man-pages project 2026). **PortXmitData** and **PortRcvData** provide the raw byte counts used to compute real-time link utilization, while **PortXmitDiscards** tracks packets dropped by the switch—in a properly configured lossless fabric, discards should be exactly zero. Physical signal integrity is tracked via the **SymbolErrorCounter**, which increments on bit-level errors caused by

loose cables or dirty optics, and the **LinkDownedCounter**, which logs link flaps often triggered by intermittent hardware faults or overheating transceivers.

These metrics are typically polled by a centralized monitoring system (for example, Prometheus with Grafana dashboards) at 10–30 second intervals to catch **silent degradation**. A common failure mode involves a QSFP56 cable negotiating at HDR speed (200 Gb/s) instead of NDR (400 Gb/s), or silently dropping from 4 lanes to 2 lanes due to a single bad connector pin. The link remains "up" and functional, but its bandwidth is halved. In a cluster of 1,024 GPUs with over 3,000 links, a 0.1 percent point-in-time component degradation rate implies about 3 degraded links in expectation, with a high probability that at least one link is degraded. Because distributed training algorithms like Ring AllReduce are synchronous, a single degraded link throttles the entire job to the speed of that straggler, transforming a minor hardware fault into a massive waste of idle compute cycles.

3.9.2 Bandwidth and latency validation

Once counters rule out visible link faults, validation must test the bandwidth and latency that the training job can actually use. A healthy NDR InfiniBand link can approach the raw 50 GB/s rate after encoding and protocol overheads, but the only useful number is the delivered payload bandwidth on the specific host pair. Operators rely on periodic health checks, often running perftest tools such as `ib_write_bw` between selected node pairs and assembling the results into an **all-pairs bandwidth matrix** (linux-rdma project 2026). This heatmap immediately visualizes cold spots in the fabric where specific spine switches or cable bundles are underperforming, allowing for targeted maintenance before jobs are scheduled.

For latency-sensitive synchronization, `ib_write_lat` measures round-trip times for small RDMA writes. Baseline NDR latency should remain below 2 μs for directly connected nodes. Latencies exceeding 5 μs suggest switch-buffer congestion or routing imbalances, while values spiking above 100 μs usually indicate the RDMA path has fallen back to TCP/IP emulation—a catastrophic configuration error. Before launching a massive training job, robust validation includes application-level tests such as `nccl-tests`, which verify that the fabric can sustain the expected AllReduce bandwidth across the specific collective topology (ring or tree) used by the workload. This ensures that the physical network reality matches the theoretical design before expensive compute resources are allocated.

3.9.3 Systematic debugging workflow

When a training job reports lower-than-expected throughput, the diagnostic order should preserve the layer model rather than starting with cables. The following sequence moves from application symptoms toward physical causes:

1. **Check GPU utilization**: Rule out compute bottlenecks using `dcgmi` or `nvidia-smi`. If SM utilization is 100 percent, the network is not the bottleneck. Low utilization does not automatically implicate the fabric, however: a starved input pipeline (data loading or storage) produces the same symptom, so confirm the data path is keeping the accelerators fed before investigating the network.
2. **Inspect NCCL logs**: Set `NCCL_DEBUG=INFO` to reveal which network transport was selected, the detected bandwidth between nodes, and any fallbacks to slower protocols.
3. **Run point-to-point tests**: Use `ib_write_bw` between specific nodes in the job. A single degraded link can bottleneck the entire ring in a Ring AllReduce.
4. **Check PFC/ECN counters**: Inspect switch counters along the path. Sustained PFC activity indicates persistent congestion that should be investigated at the scheduler or routing level.
5. **Validate Physical Layer**: Check symbol errors and CRC counts to identify failing transceivers or cables.

The diagnostic sequence preserves the chapter's layer model: application symptoms identify the failing path, transport counters show whether congestion or fallback is involved, and physical telemetry confirms whether the wire itself is degrading.

Checkpoint 3.5: Diagnosing a training slowdown

Your 175B model training job has been running for 3 days on 512 GPUs. You notice that the iteration time has gradually increased from 4.2 seconds to 4.8 seconds (a 14 percent slowdown). The GPU utilization reported by `nvidia-smi` has dropped from 92 percent to 85 percent.

- ☐ **First diagnostic**: Choose the first diagnostic step, name the tool that confirms whether the bottleneck is communication-bound or compute-bound, and cite the evidence that distinguishes the two.
- ☐ **Physical layer**: Identify physical-layer causes that could explain a 50 percent bandwidth drop when `ib_write_bw` finds one node pair at 24 GB/s instead of the expected 48 GB/s, and verify them using switch counters.
- ☐ **Collective impact**: Estimate the impact on overall AllReduce time when the degraded link lies in the Ring AllReduce path, then evaluate whether switching to a Tree AllReduce algorithm helps.

3.10 Network Technologies That Move the Ceiling

Monitoring keeps deployed fabrics honest, but observability cannot push past a physical ceiling that copper approaches at high signaling rates. The physical ceiling that monitoring exposes is the reason new fabrics deserve attention: they matter only when they move a bound the synchronization backbone cannot cross. As clusters scale toward 100,000 nodes, the durable design variables are power per bit, reach, and the ratio of compute to memory to network capacity, not the product names attached to a particular roadmap.

3.10.1 Protocol convergence and link density

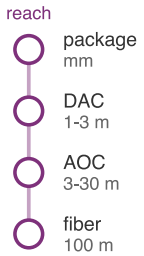
The first pressure point is reliability. The Ultra Ethernet Consortium (UEC) targets that constraint by trying to make Ethernet behave more like an HPC fabric without inheriting RoCEv2's PFC fragility. Its Ultra Ethernet Transport design combines packet spraying across multiple paths, flexible ordering, multiple transport delivery services, congestion-control changes, and telemetry intended for AI/HPC traffic (Ultra Ethernet Consortium 2025). The goal is to preserve Ethernet's ubiquity and multi-vendor economics while moving its failure behavior closer to InfiniBand's lossless fabric model. Specific mechanisms such as packet trimming or link-layer retry are implementation details of that broader direction, so they should be evaluated against the same durable criteria: whether the fabric can repair congestion locally before it becomes a synchronous-training straggler.

Higher port rates attack hop count as well as raw bandwidth. XDR InfiniBand pushes 800 Gb/s per port, and Ethernet roadmaps include 800GbE and 1.6TbE link speeds (NVIDIA 2026c; Ethernet Alliance 2025). Ethernet switch silicon with 1.6 Tb/s ports pushes aggregate capacity to 102.4 Tb/s in current Tomahawk 6-class designs (Broadcom Inc. 2025). A single 1.6 Tb/s port delivers the bandwidth of 4 400G ports. The increased density allows architects to flatten the topology: a cluster that previously required three tiers of switches may be served by two when radix, port rate, and placement constraints line up, halving the transceiver count and reducing tail latency by removing an entire hop of switching and FEC overhead.

3.10.2 Optical interconnects

Optical interconnects attack the power-per-bit and reach terms that copper makes worse as data rates increase. At 112 Gb/s per lane (the PAM-4 signaling rate used by NVLink 4.0 and InfiniBand NDR), copper SerDes transceivers consume approximately 7–10 pJ per bit and are limited to distances of 2–3 meters before signal integrity degrades beyond the point where equalization can recover the data.

The 224 Gb/s-per-lane class pushes copper even harder. At 224 Gb/s, the SerDes power consumption roughly doubles to 15–20 pJ per bit, and the reach shrinks to 1–1.5 meters over passive copper cables. Active electrical cables (which include retimer chips to regenerate the signal mid-cable) extend the reach but add latency and power.



Each extra meter pushes the fabric from copper toward optics.

Co-packaged optics (CPO) addresses these limitations by integrating optical transceivers directly into the switch or accelerator package, rather than placing them at the end of a pluggable cable module. In CPO, a silicon photonics chip sits on the same package substrate as the processor, converting electrical signals to light at the die boundary. The optical signal then travels through a fiber at the speed of light, with negligible attenuation over distances of tens of meters, and is converted back to electrical at the receiving package.

CPO changes the link budget through three independent physical effects:

- **Shorter electrical path**: Moving the optical transceiver from the switch faceplate to the package substrate reduces the SerDes-to-optics path from centimeters to millimeters, lowering the power consumed by electrical signal conditioning.
- **Distance-stable bandwidth**: Optical fiber has no distance-dependent bandwidth degradation over data center-relevant distances up to 100 meters, eliminating retimers and enabling more flexible physical layouts.
- **Lighter cable plant**: Fiber is lighter and thinner than copper cables, simplifying cable management in dense racks.

Together, these effects shift the fabric from a copper-limited layout problem toward an optics-limited packaging and power problem.

A quick estimate makes the power dividend concrete.



Napkin Math 3.6: The optical dividend

Problem: Calculate the power savings of moving a 51.2 Tb/s switch from pluggable transceivers to Co-Packaged Optics (CPO).

1. **Pluggable Architecture**: 128 ports $\times$ 20 W = 2.56 kW for optics alone.
2. **CPO Architecture**: 128 engines $\times$ 10 W = 1.28 kW.
3. **Savings**: The savings reach 1.28 kW of power per switch.

Systems insight: In a cluster with 1,000 switches, pluggable optics consume 2.56 MW just to move light. CPO halves this "network tax," saving 1.28 MW and redirecting enough power to fuel roughly 1,800 H100 GPUs. At the BW_{bisect} wall, sustainability is not a choice; it is an architectural requirement driven by the thermal limits of the faceplate.

For a 10,000-GPU cluster, eliminating pluggable modules could save more than a megawatt of power that can be redirected to computation. CPO also removes the transceiver as a discrete field-replaceable unit, eliminating a common point of mechanical failure.

For ML infrastructure, CPO could flatten the bandwidth hierarchy by narrowing the gap between intra-node and inter-node bandwidth. If inter-node links achieve bandwidth comparable to NVLink-class local fabrics (hundreds of GB/s per GPU), the constraint that confines tensor parallelism to within a single node would relax. This would enable new parallelism strategies where tensor parallelism spans two to four nodes rather than being confined to a single 8-GPU node, potentially improving scaling efficiency for very large models.

The product roadmap is less durable than the constraint it illustrates: as link speed rises, optics moves closer to silicon to control power and reach. Products that integrate optics into switch packages or accelerator packages are concrete attempts to pay less power and latency for the same physical distance. Whether a specific product generation succeeds is less important than the systems direction: the electrical path shrinks, the optical path begins closer to the die, and package-level thermals become part of network design.

However, CPO introduces new challenges. Optical components are sensitive to temperature (laser wavelength shifts with temperature, requiring active thermal management), and integrating photonics on the same package as a 1,000 W GPU creates a hostile thermal environment for the optical components. The manufacturing processes for silicon photonics and CMOS transistors are similar but not identical, requiring separate fabrication steps that increase package cost and complexity. These challenges are being addressed through hybrid integration approaches where the

photonics chiplet is placed on a cooler region of the package substrate, thermally isolated from the GPU die.

3.10.3 Disaggregated and composable architectures

The conventional node bundles a fixed ratio of compute, memory, and networking, and section 2.7.1 established why that ratio is wasteful when one workload needs more memory capacity per GPU and another needs more network bandwidth: CXL memory pooling lets a processor in one chassis reach memory in another with load/store semantics, and disaggregated designs decouple compute, memory, and networking into independently composable pools. The fabric-specific consequence is what concerns this chapter. Once memory pooling and resource composition cross the node boundary, the rigid distinction between intra-node NVLink domains and inter-node InfiniBand domains softens into a single memory fabric, and the synchronization backbone shifts from a fixed node boundary to a composable one.

That shift does not repeal the physics this chapter has developed. A composed virtual node still pays the same α-β costs on every cross-pool access, the same power-per-bit limits on the links that carry it, and the same observability requirements that keep an oversubscribed path from masquerading as slow accelerators. The systems lesson is the displacement of overhead: disaggregation relocates the synchronization backbone rather than eliminating it. Whether a composable fabric works is decided by the same fabric properties that decide whether a fixed-node fabric works.

3.11 Fallacies and Pitfalls

Designing and operating high-performance fabrics for ML requires unlearning assumptions from traditional data-center networking. The following fallacies and pitfalls capture the most common errors that stall training and degrade cluster productivity.

Fallacy: *More bandwidth always means faster training.*

Engineers assume upgrading from HDR (200 Gb/s) to NDR (400 Gb/s) will yield proportional gains, but the α-β model (section 3.4.4) reveals this is only true in the bandwidth-dominated regime. For small models or the small-message phases of pipeline parallelism, the latency term α, dominated by switch hops and FEC (~1 μs), dictates performance. If a workload is latency-bound, a 2$\times$ bandwidth increase may deliver only a small fraction of the line-rate gain while adding substantial power and transceiver cost.

Consider a 10 KB message (typical for control synchronization). On 200 Gb/s InfiniBand, it takes 1.90 µs. Upgrading to 400 Gb/s reduces this to 1.70 µs, a 10.5 percent improvement despite doubling the link rate.

Pitfall: *Treating InfiniBand as just fast Ethernet.*

Procurement teams compare InfiniBand and high-speed Ethernet on link rate alone and conclude they are equivalent at the same Gb/s. InfiniBand and Ethernet differ architecturally, not just in speed. InfiniBand provides kernel-bypass (RDMA), hardware-managed flow control, and credit-based congestion avoidance, all standardized and predictable. Ethernet relies on software-managed TCP/IP stacks with orders-of-magnitude higher latency and requires PFC/ECN approximations to behave losslessly under RDMA. The choice between InfiniBand and Ethernet is a system architecture decision, not a bandwidth selection: it determines whether failure modes are bounded by hardware credits or by software configuration discipline.

Fallacy: *Lossless Ethernet is as reliable as InfiniBand.*

RoCE over Ethernet can achieve throughput comparable to InfiniBand for large transfers, but the "lossless" property is an approximation maintained by PFC (section 3.6.1). A misconfigured switch or a firmware bug can trigger a **PFC storm**, where PAUSE frames propagate in a loop, freezing the entire fabric. InfiniBand's credit-based flow control is inherently immune to such cascades because it operates on per-hop buffer availability rather than reactive signals. Teams deploying RoCE must invest substantially more engineering effort in fabric testing and monitoring to avoid multi-day outages caused by "lossless" deadlocks.

Pitfall: *Accepting oversubscription because most traffic is local.*

In general-purpose clouds, 4:1 or 8:1 oversubscription at the spine is common because traffic is stochastic. However, ML training is bulk synchronous. When an AllReduce starts, the barrier's coordination forces every node to inject full bandwidth simultaneously. As shown in the bisection

bottleneck analysis (section 3.5.2), even a 4:1 oversubscription slows the entire synchronization by 4×. For a $300M cluster where sync accounts for 30 percent of time, this "cost optimization" wastes over $142.1M in idle GPU cycles.

Fallacy: *TCP bandwidth tests are sufficient for ML fabric validation.*

Standard benchmarks like `iperf` measure kernel-based TCP/IP performance. Because TCP requires CPU-managed buffer copies and context switches, it often caps at 20–40 Gb/s regardless of the wire speed. ML training uses RDMA, which bypasses the kernel entirely. A link that appears "broken" at 30 Gb/s in `iperf` might be perfectly healthy and deliver 390 Gb/s in `ib_write_bw`. Validation protocols must use RDMA-specific tools from the `perftest` suite to match the workload's data path.

Pitfall: *Relying on adaptive routing instead of topology-aware placement.*

Adaptive routing distributes traffic across available paths, but it cannot create bandwidth that does not exist. If a scheduler places a 1,024-GPU job across two oversubscribed spine groups, adaptive routing will balance the traffic, but it will still be throttled by the group-to-group links. Topology-aware placement, discussed in section 8.3, and adaptive routing are complementary: the former ensures bandwidth exists, while the latter ensures it is used efficiently.

Fallacy: *Network fabric problems always appear as hard failures.*

Network performance issues in ML clusters often manifest as subtle training slowdowns rather than errors. A 10 percent reduction in throughput due to intermittent congestion can waste thousands of GPU-hours before being noticed, because jobs continue to run while gradients arrive late.

Pitfall: *Treating PFC and ECN counters as optional production details.*

Operators must alert on PortXmitDiscards and PFC Pause frame rates. A gradual increase in these counters is often the leading indicator for a failing transceiver or a routing imbalance that will eventually lead to a job failure.

3.12 Summary

The Five-Level Model shown in figure 3.1 structures our analysis of high-bandwidth fabrics, ascending from the physics of signal transmission to the architecture of warehouse-scale clusters. The framework reveals that network performance is the product of interactions between physical reach, transport protocols, topology, and congestion control, not link speed in isolation. Level 1 (Wire) established that PAM4 encoding and FEC impose irreducible latency floors constraining cluster diameter. At Level 2 (Transport), InfiniBand's native credit-based flow control and RoCE's reliance on PFC yield different reliability guarantees, and the α-β model quantifies the bandwidth-latency trade-offs inherent in distributed collectives.

Level 3 (Topology) demonstrated how non-blocking fat-trees and rail-optimized designs provide the structural BW_{bisect} required by global AllReduce patterns. However, Level 4 (Behavior) showed that structure alone is insufficient: in the synchronous world of BSP training, tail latency is the dominant constraint, necessitating proactive congestion control mechanisms like DCQCN and HPCC to prevent incast-induced stalling. Level 5 (Cluster Design) integrated these layers into production architectures like the NVIDIA SuperPOD and Meta Grand Teton, illustrating how virtualization and multi-tenancy allow these massive instruments to be shared safely.

At the physical limit, the constraints of copper and the operational complexity of Ethernet drive new interconnect designs. Ultra Ethernet Consortium (UEC) standards, Co-Packaged Optics (CPO), and CXL memory pooling all aim to flatten topologies or reduce the power tax of moving data. Yet, as our monitoring workflow discussion emphasized, no technology eliminates the need for rigorous observability. Whether debugging a single degraded transceiver or optimizing a multi-tenant scheduler, the ability to correlate physical counters with application-level throughput remains the ultimate safeguard against silent waste in the machine learning fleet.

The practical value of this layered understanding is diagnostic precision. When a distributed training job underperforms, the complaint is invariably "the network is slow," but slowness has many causes: a failing transceiver degrading a single link, a PFC storm cascading across a subnet, a topology bottleneck starving one communication pattern while adequately serving another. Engineers who understand the Five-Level Model can isolate the layer at fault, correlate physical counters with transport behavior, and distinguish a topology limitation from a congestion control misconfiguration.

This capacity to reason across abstraction layers, from SerDes signal integrity to cluster-wide BW_{bisect}, is what separates routine troubleshooting from genuine systems engineering.

Equally important, the α-β cost model provides a quantitative vocabulary for making architectural decisions before hardware is purchased and racks are wired. Choices between InfiniBand and RoCE, between fat-tree and rail-optimized topologies, and between 400G and 800G link speeds are all decisions with multi-million-dollar consequences that hinge on the interaction between message size distributions, collective algorithms, and physical link characteristics. The framework developed in this chapter equips practitioners to evaluate these trade-offs with analytical rigor rather than vendor benchmarks alone.

Key Takeaways: The fabric decides useful compute

- **Link speed is not fabric speed**: A global AllReduce is limited by the narrowest bisection cut, not the fastest advertised port. Topology decides how much purchased accelerator throughput becomes useful training throughput and how much becomes idle silicon.
- **Latency and bandwidth bind differently**: The α-β model separates startup cost from per-byte transfer cost, with $n^* = \alpha \cdot \beta$ marking the regime change. Message-size distributions, not vendor peak numbers, determine whether to optimize software latency or hardware bandwidth.
- **Losslessness moves the risk**: RDMA needs a lossless fabric to avoid expensive retransmission stalls. InfiniBand supplies this natively, while RoCE depends on PFC, ECN, DCQCN, and HPCC, trading hardware flexibility for tail-latency and operational complexity.
- **Topology must match traffic**: Fat-trees buy flexible bisection bandwidth, rail-optimized designs accelerate same-rank AllReduce, and dragonfly or torus designs trade cabling and locality differently. The right fabric depends on whether the workload stresses AllReduce, AllToAll, or multi-tenant sharing.
- **Telemetry saves GPU-hours**: PFC counters, link error rates, bandwidth baselines, and application throughput must be correlated across layers. Without that observability, a degraded transceiver or congestion storm silently converts a high-end fleet into a queue of waiting accelerators.

Link speed is the number everyone quotes and the wrong one to trust. What decides whether a fleet computes as one machine is bisection bandwidth, the rate at which one half of the cluster can talk to the other, because a global AllReduce is only as fast as the fabric's narrowest cut. A topology that starves that cut turns expensive silicon into idle silicon, waiting on gradients the wires cannot deliver in time. The α-β model puts a number on the waiting, separating the latency a message pays to *start* from the bandwidth it pays to *finish*. That is why this chapter treats the interconnect as part of the computer and not the plumbing between computers: at fleet scale, the fabric decides how much of the silicon was worth buying.

What's Next: From wires to data pipelines

The network fabric now binds compute nodes into a fleet, and every byte of gradient data and activation tensor flows through it. The fleet, however, also needs to move massive datasets and enormous checkpoints. Chapter 4 examines the parallel storage systems and data-loading architectures that keep the fleet supplied with data.

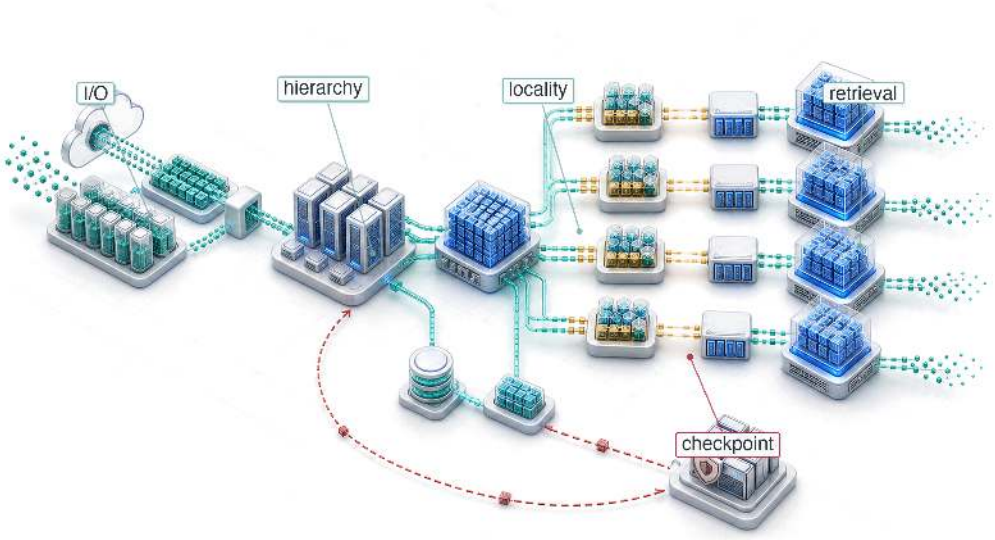
Research Questions: For further inquiry

- Why does bisection bandwidth, rather than advertised link speed, decide how much accelerator compute becomes useful?

- How should the α-β model change decisions about link upgrades, hop count, and topology?
- When is InfiniBand preferable to RoCE, and when might RoCE's operational flexibility justify its risks?
- How can one degraded link or congestion cascade silently waste an entire training fleet?

4

Data Storage



4.1 The Fuel Line
4.2 How ML Workloads Invert Storage Assumptions
4.3 The ML Storage Hierarchy
4.4 The Data Pipeline Equation
4.5 GPUDirect Storage and the CPU Bypass
4.6 Storage Economics
4.7 Checkpoint Storage
4.8 Retrieval Infrastructure: Vector Indexes
4.9 The Synthetic Fuel Line
4.10 Fallacies and Pitfalls
4.11 Summary

Purpose

Why does storage become the invisible bottleneck that prevents accelerators from reaching their potential?

Accelerators can compute faster than storage can feed them. A high-end accelerator processes data at terabytes per second internally, but individual local drives deliver gigabytes to low tens of gigabytes per second, and distributed storage systems add latency that compounds into idle accelerators waiting for data to arrive. This mismatch is invisible in benchmarks that measure accelerator performance in isolation but dominates real workloads where training data must stream continuously, checkpoints must be saved reliably, and model weights must be loaded at serving time. The gap between what accelerators can consume and what storage can deliver shapes system architecture at every level: it forces careful attention to data formats, caching strategies, and pipeline design that would be unnecessary if storage kept pace with compute. Organizations that optimize accelerator utilization without addressing storage discover that their expensive hardware runs at a fraction of capacity because nobody planned for the data path. In C^3 terms, data storage is a compute-communication co-design problem: the fastest compute in the fleet sits idle when the storage hierarchy cannot supply data at the rate the accelerators consume it.

Learning Objectives

- Explain how ML storage workloads invert database assumptions through streaming, checkpoint bursts, and metadata pressure
- Calculate required training bandwidth with the pipeline equation and target accelerator utilization
- Compare memory, flash, parallel file, object, and archive storage tiers by bandwidth, latency, capacity, and cost
- Design prefetching, sharding, caching, and locality strategies that prevent fleet-scale accelerator starvation
- Evaluate accelerator-direct and CPU-bypass storage paths for latency, augmentation, and locality bottlenecks
- Select checkpoint staging and replication strategies that balance pause time, recovery risk, and storage cost
- Assess retrieval indexes and synthetic-data pipelines as storage workloads with distinct latency, consistency, and governance constraints

4.1 The Fuel Line

Dense accelerator nodes can pack eight GPUs delivering petaFLOP/s of aggregate compute, and InfiniBand fabrics can connect thousands of such nodes at hundreds of gigabits per second. Within the fleet stack shown in figure 1.13, data storage completes that physical foundation by providing the fuel supply: training data, model weights, optimizer state, and intermediate checkpoints staged at the right distance from the accelerator. An engine without fuel is expensive sculpture. The engineering question is how to deliver that fuel fast enough that 1,000 accelerators never starve.

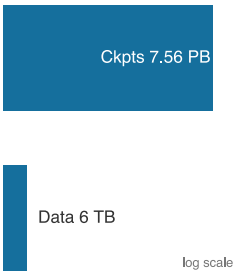


Storage is the Infrastructure axis of the fleet stack.

Consider the running example for the storage analysis. A 175-billion parameter language model trains on 1.5 trillion tokens of text: roughly 3 TB in compressed source form, or 6 TB once represented as 4-byte token IDs. Each training epoch reads every token once, in a shuffled order determined by the random seed. There is no "hot" subset of data that dominates access; every byte is consumed exactly once per pass. Meanwhile, each accelerator processes its local batch in roughly 200 ms, then waits for the next. If storage cannot deliver data within that 200 ms window, the accelerator sits idle, and the organization pays for silicon that produces heat instead of gradients.

The problem is deceptive because storage technology has improved substantially. NVMe drives achieve 7 GB/s of sequential throughput, a figure that would have seemed out of reach a decade ago. The accelerators, however, improved faster. An H100 GPU consumes data from its HBM at 3.35 TB/s, roughly 478.6× faster than a single NVMe drive can feed it. The gap between storage delivery and accelerator consumption is the central storage tension, and it cannot be solved by any single technology. Instead, it requires a hierarchy of storage tiers, each carefully matched to a specific phase of the ML lifecycle, connected by pipelines that hide latency through prefetching and pipelining.

The storage problem is fundamentally one of physics meeting economics. Physics dictates that data closer to the accelerator (in both physical distance and interconnect hops) can be delivered faster but in smaller quantities. Economics dictates that cheaper storage can hold more data but at greater distance. The engineering art is constructing a pipeline that bridges these constraints, keeping the expensive top tier full by drawing from cheaper lower tiers fast enough that the accelerator never perceives the delay. The resulting design problem is quantitative: how fast each tier must be, how deep the pipeline must run, and which bytes are worth keeping close to the accelerator.



Checkpoint writes dwarf training-data reads by about 1,000 times.

The canonical training-data footprint for this example is roughly 9 TB across the hierarchy, combining the compressed corpus and tokenized shards introduced above. Additional shuffled or packed variants can raise the staging footprint, but the per-epoch training read over 4-byte token IDs is 6 TB. The model generates roughly 1.75 TB checkpoints (1,750 GB total: 350 GB of weights plus 1.4 TB of Adam optimizer state) every 10 minutes. Over a 30-day training run on 256 nodes, the storage system must deliver 6 TB of tokenized training data per epoch, absorb 7.6 PB of checkpoint

writes, and stage model weights for evaluation runs. These numbers anchor the sections that follow, grounding abstract principles in concrete engineering constraints.

Those numbers force the storage path. ML access patterns first invert the assumptions behind conventional storage systems; that inversion forces a hierarchy from HBM to cold archive; the hierarchy then requires pipeline equations, direct data paths, and economics that decide which bytes belong at each tier. Checkpoints, retrieval indexes, and synthetic-data provenance are variations on the same fuel-line problem: the storage system must place the right representation at the right distance before the accelerator asks for it.

4.2 How ML Workloads Invert Storage Assumptions

A database administrator moving to an ML infrastructure team would find that ML storage workloads invert nearly every storage design principle they relied on. The 175B running example makes the inversion concrete. Each training epoch reads every token, in shuffled order, exactly once. The next epoch shuffles again and reads them all once more. There is no "hot data" in the traditional sense and no 80/20 rule where a small fraction of data accounts for most accesses. The standard storage optimizations fail precisely because they assume the opposite.

Traditional storage systems evolved to serve transactional databases, workloads characterized by small random accesses, strong consistency, and moderate bandwidth. A database server might issue thousands of 4 KB reads per second to serve user queries. The industry optimized for this pattern over decades, developing sophisticated caching algorithms, write-ahead logs, and RAID[1] configurations tuned for small-block random access. Each optimization assumed that the most recently accessed data would likely be accessed again soon.

ML workloads systematically invert these assumptions. Training data access is predominantly sequential, streaming through datasets that span hundreds of terabytes. Individual accesses are large (megabytes rather than kilobytes) because models consume batches of images or text sequences. Consistency requirements are relaxed, since slightly stale features rarely affect model quality. Bandwidth demands, however, are extreme: hundreds of gigabytes per second, sustained for days or weeks. The mismatch between what storage was optimized for and what ML actually needs creates what we call the **I/O wall** (principle 6): when storage throughput cannot deliver training data as fast as accelerators consume it, GPUs idle regardless of their computational power. A storage system that was adequate for 8 GPUs becomes the bottleneck at 64, making the *data pipeline*, not the *model*, the limiting factor. Section B.5.2 classifies this wall as a constraint that lives at the intersection of Communication and Compute in the fleet-scale diagnostic framework, so a storage engineer can confirm that the cure is more bandwidth rather than faster accelerators.

A simple shard-assignment calculation shows how this bottleneck can emerge even when aggregate storage capacity looks ample.

1 **RAID (Redundant Array of Independent Disks)**: A 1988 Berkeley taxonomy of drive-combining strategies, each trading redundancy against bandwidth (Patterson et al. 1988). ML training inverts the database-era default: RAID 0 (striping, no parity) maximizes sequential read throughput at the cost of zero fault tolerance, a safe trade-off[^fn-raid0-immutable] because training data is immutable and durably backed in object storage. Choosing RAID 5 or 6 instead wastes bandwidth on parity calculations that protect data already protected elsewhere.

Napkin Math 4.1: The thundering herd: Shard contention

Problem: A dataset is split into 1000 shards on a shared file system. If 32 workers each pick a shard at random to start their next epoch, what is the probability that at least two GPUs "collide" on the same storage server, causing a performance bottleneck?

Math: This is a variant of the "Birthday Problem" in probability.

1. **Probability of No Collision**: $\approx e^{-n^2_{\text{workers}}/(2K_{\text{shards}})} = e^{-32^2/2000} \approx 0.60$.
2. **Probability of Contention**: $1 - 0.60 = \mathbf{40}\%$.

Systems insight: Even with a large number of shards, the birthday-problem approximation $1 - e^{-n^2/(2k)}$ yields a 40.1 percent chance of storage shard contention with $n = 32$ workers and $k = 1{,}000$ shards: a "hot spot" where multiple workers land on the same storage server in the same epoch. In a distributed fleet, these collisions create tail latency: the entire cluster waits for the two GPUs sharing a disk to finish. To solve this, production data loaders use global shuffling and deterministic shard assignment to ensure that workers are perfectly distributed across the storage fabric, eliminating the "thundering herd" effect.

Shard collisions are one runtime symptom of the I/O wall; the hardware trend behind that wall is a widening gap between compute throughput and storage bandwidth.

> **Systems Perspective 4.1: The widening I/O wall**
>
> Between 2016 and 2024, advertised accelerator Tensor Core throughput grew sharply, but exact ratios depend on whether the comparison holds precision fixed or follows each generation's lowest supported training/inference precision. Over the same period, NVMe sequential bandwidth grew far more slowly, from roughly 3.5 GB/s to 14 GB/s per drive. The compute-to-storage bandwidth ratio has therefore worsened substantially. If that pattern continues, the storage hierarchy must add new tiers (persistent memory, Compute Express Link (CXL)-attached storage) or fundamentally change the data pipeline architecture (compute-near-storage, in-storage processing) to prevent the I/O wall from becoming the binding constraint on training throughput. The durable planning lesson is that compute improvements do not automatically carry the storage path with them.

Figure 4.1 makes this widening gap visually precise by tracking GPU throughput and storage bandwidth side by side on the same timescale.

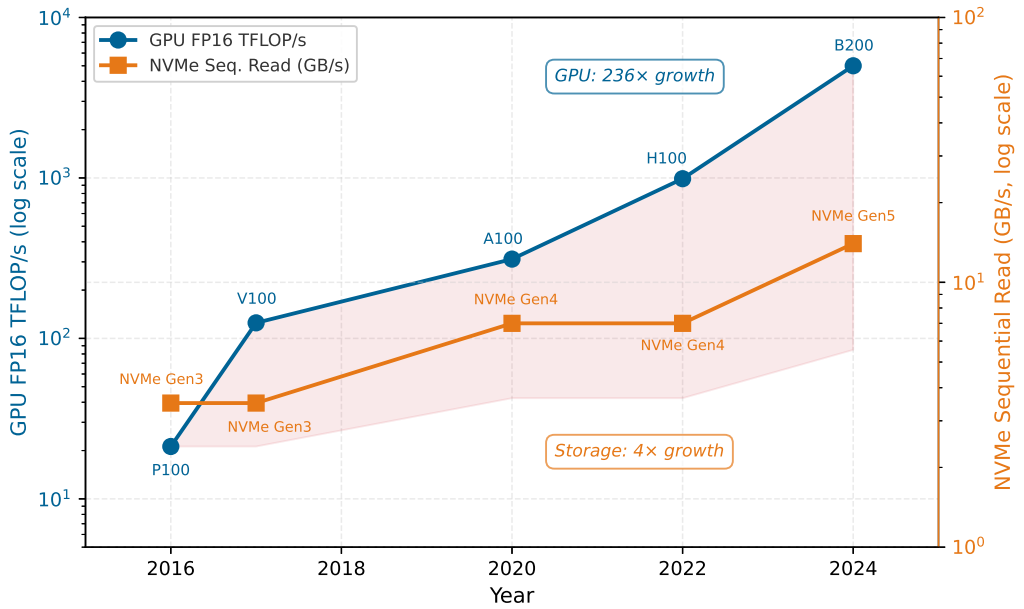


Figure 4.1: The Storage-Compute Chasm: GPU peak FP16 throughput (blue, left axis) and NVMe sequential read bandwidth (orange, right axis) from 2016 to 2024, both on logarithmic scales. GPU throughput has grown 236× while storage bandwidth has grown only 4× over the same period. The shaded region highlights the widening gap that data pipeline engineering must bridge through prefetching, caching, and format optimization.

Figure 4.1 illustrates *why* the I/O wall is the defining constraint of this chapter. The nearly 60× ratio between GPU throughput growth (236×) and storage bandwidth growth (4×) means that every new GPU generation increases the pressure on the data pipeline. This growth divergence, how fast the gap has widened over the years, is distinct from the static tier-to-tier bandwidth cliffs that later sections quantify, which measure how steep the gap is at a single moment. Without a multi-tier storage hierarchy, prefetching, and format optimization, the expensive accelerators at the top of the stack would spend more time waiting for data than computing on it. The remainder of this chapter examines how each tier of the storage hierarchy addresses a different facet of this chasm.

The first inversion is **access pattern**. Where database workloads exhibit *random access patterns* that benefit from seek-time optimization, ML training performs massive *sequential scans*. A training epoch

reads every sample once, in whatever order the shuffling algorithm produces. This pattern resembles video streaming more than database queries. Storage systems optimized for random IOPS[2] waste their capabilities on ML workloads, while systems optimized for sequential throughput excel. The distinction is quantitatively dramatic: a Gen4/Gen5 NVMe drive delivers roughly 7–14 GB/s for sequential reads, while small 4 KB random reads can fall near 0.5 GB/s, an order-of-magnitude to 30× penalty for the wrong access pattern. The penalty is even more severe on hard drives, where mechanical seek times impose a 100× throughput reduction for random access compared to sequential.

The shuffling that ML training requires adds a complication rooted in the requirements of stochastic gradient descent. Stochastic gradient descent requires that each mini-batch be drawn approximately uniformly from the training distribution; presenting samples in a correlated order (all samples from the same document, all images from the same class) biases the gradient estimates and slows or destabilizes convergence. True global shuffling would satisfy this requirement perfectly, but it requires random access across the entire dataset, destroying the sequential access pattern that storage hardware demands. For a petabyte-scale corpus, random per-sample seeks are I/O-prohibitive: a dataset of 1 trillion 4-byte tokens stored as individual elements on NVMe, accessed at random, would take thousands of hours to read at the drive's random-read IOPS rate rather than the few hours achievable with sequential streaming. The practical solution is therefore a compromise: shuffle the order of large data shards, then shuffle samples within each shard's local buffer. This achieves sufficient randomness for training convergence while preserving the sequential I/O pattern that storage hardware demands. The shard size determines the trade-off: larger shards provide more within-shard shuffle diversity but require more memory for the shuffle buffer.

The second inversion is **working set size**. Traditional applications exhibit temporal locality: a web server repeatedly accesses popular pages, and a cache holding the top 10 percent of content serves 90 percent of requests. ML training datasets are accessed uniformly. Our 6 TB serialized corpus has no "popular" tokens; each is consumed once per epoch. A cache of any practical size holds only a tiny fraction of the dataset, and every sample is effectively "cold" when accessed. This lack of temporal locality renders traditional caching strategies ineffective. Even an LRU[3] (Least Recently Used) cache, the workhorse of database systems, achieves a 0 percent hit rate on uniformly accessed data, because by the time a sample is accessed again in the next epoch, it has long been evicted to make room for other samples.

The exception to this uniformity is multi-task or curriculum learning, where certain subsets of the dataset are accessed more frequently during specific training phases. In curriculum learning, the trainer begins with "easy" examples and progressively introduces harder ones. This creates a temporary working set that does exhibit locality, and local caching at the NVMe tier can exploit this structure. For the majority of large-scale pretraining workloads, however, the access pattern is effectively uniform, and the storage system must be designed for full-dataset streaming rather than hot-subset caching.

The third inversion is **write pattern**. Transactional systems generate continuous streams of small writes, each immediately durable. ML systems generate occasional massive writes when saving checkpoints. A 175B parameter model checkpoint, including optimizer state, occupies roughly 1,750 GB. Saving it every 10 minutes generates concentrated bursts that saturate bandwidth for seconds, followed by long idle periods. Together, these bursts form a **checkpoint storm**: a synchronized checkpoint-write event that parallel file systems must absorb without disrupting ongoing training reads. The bursty write pattern is particularly challenging because all nodes in the cluster write their checkpoint shards simultaneously. If 128 nodes each write 14 GB, the parallel file system receives 1.8 TB of writes in a single burst, which must complete before the training pipeline can resume. Table 4.1 consolidates these inversions into the procurement rule: optimize for streaming and bursts, not database-style random IOPS.

2 **IOPS (Input/Output Operations Per Second)**: The metric that dominated storage procurement for decades because database workloads issue millions of small random reads. ML training inverts this priority: a pipeline streaming 256 MB shards sequentially needs sustained GB/s throughput, not per-operation speed. Provisioning storage by IOPS rating for an ML workload over-spends on random-access capability the pipeline never exercises.

3 **LRU (Least Recently Used)**: LRU's optimality proofs assume temporal locality, the property that recently accessed data will be accessed again soon. ML training's uniform-access-per-epoch pattern violates this assumption maximally: every sample is accessed exactly once, making LRU's eviction decisions no better than random. Teams that provision a large DRAM cache expecting database-like hit rates on training data discover 0 percent reuse, wasting memory that would be better allocated to prefetch buffers.

Table 4.1: ML Workloads Invert Traditional Storage Assumptions: Where databases optimize for random IOPS with cacheable working sets, ML training streams sequentially through datasets that exceed all cache levels.

Workload Pattern	Traditional Assumption	ML Reality
Access pattern	Random access	Sequential streaming

Workload Pattern	Traditional Assumption	ML Reality
Working set	Fits in cache	Exceeds all cache levels
Write pattern	Continuous small writes	Bursty large writes
Read/write ratio	Balanced	Phase-dependent (100:1 to 1:0)
Locality	Strong temporal locality	No locality (uniform sampling)

As table 4.1 shows, these inversions have a fourth, subtler dimension: the read/write ratio shifts dramatically by lifecycle phase. During training, reads dominate writes by 100:1 or more, as the system streams through data continuously and saves checkpoints occasionally. During checkpoint-heavy phases in fault-prone clusters, writes can briefly dominate. During data preprocessing, both reads and writes are heavy, and the access pattern more closely resembles a MapReduce job than a training loop: tokenization, deduplication, and shuffling scan the raw corpus once (large sequential reads), write intermediate artifacts (large sequential writes), and then scan those artifacts again for the next stage. A single 1 TB text corpus typically expands to 6 TB of tokenized shards before the first training epoch begins, loading each storage tier differently from the read-only training phase that follows. No single storage configuration optimizes for all phases, which is why ML systems require a multi-tier hierarchy rather than a single storage technology.

A fifth inversion emerges when comparing training and inference workloads. *Training* reads datasets sequentially and writes checkpoints in bursts. *Inference*, by contrast, reads model weights once at startup (a large sequential read of potentially hundreds of gigabytes), then performs no further storage I/O during normal operation because the model resides entirely in HBM. The storage challenge for inference is cold-start latency: the time required to load a model from storage to HBM when scaling up or recovering from failure. Because the model is read once and then resident, the binding constraint flips from sustained streaming throughput to a single bulk read whose duration depends entirely on which tier supplies the weights, with a parallel file system taking several times longer than local NVMe. Section 4.6.3 derives the concrete load times for a 175B model. For serving workloads with strict availability requirements, this cold-start time drives the design toward keeping warm replicas in host DRAM or using model sharding to parallelize the load across multiple storage devices.

When scaling from a single user to thousands of concurrent inference requests, the storage challenge shifts from a single-stream throughput problem to a massive fan-out distribution problem. A serving cluster with 100 replicas of our 175B parameter model requires 35 TB of model weights distributed across the cluster. When a new model version is deployed (a **model rollout**), all 100 replicas must be updated, triggering a 35 TB data distribution event that must complete within minutes to minimize serving disruption. This is analogous to the checkpoint storm in training but in reverse: Instead of many nodes *writing* to a central location simultaneously, many nodes are *reading* the same data simultaneously. The storage system must sustain this burst read bandwidth for model distribution while continuing to serve inference requests from the existing model version without degradation.

These inversions have direct consequences for system procurement and architecture. An organization provisioning storage for ML based on database-era heuristics will over-invest in random IOPS (which ML does not need), under-invest in sequential bandwidth (which ML desperately needs), and fail to account for the bursty write patterns that checkpointing creates.

Checkpoint 4.1: Storage workload analysis

You are designing the storage subsystem for a new ML training cluster with 512 GPUs. The primary workload will train large language models on a 10 TB text dataset.

- ☐ **Database heuristics**: Use the five storage workload inversions to name at least three database optimization strategies that would be counterproductive for this workload.

- ☐ **Epoch reads**: Calculate the total data volume read when each training epoch reads the full 10 TB dataset once and the cluster trains for 100 epochs, then compare it to a typical database workload.
- ☐ **Checkpoint bandwidth**: Calculate the minimum write bandwidth required when the model saves a 500 GB checkpoint every 15 minutes and each save must complete within 30 seconds.
- ☐ **Write duty cycle**: Calculate the percentage of time the storage system spends in "write mode" vs. "read mode" under bursty checkpoints every 15 minutes.

Understanding which storage tier can keep accelerators fed is essential to system design. Figure 4.2 plots the required I/O throughput to sustain full GPU utilization against model size, with horizontal ceilings for each storage tier overlaid. Below the NVMe ceiling, local SSDs can sustain the training feed; between NVMe and parallel-filesystem ceilings, only Lustre-class storage suffices; and beyond the parallel-filesystem ceiling, the workload enters the storage-bottleneck regime where no single tier can sustain the accelerators on its own.

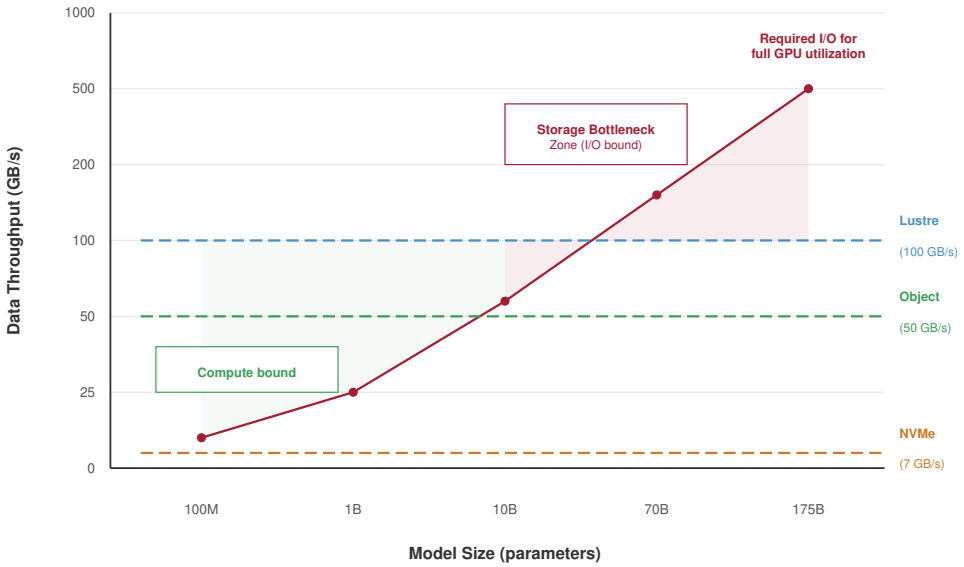


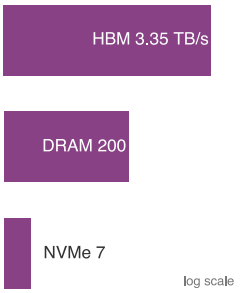
Figure 4.2: The Storage Bottleneck Zone: Required I/O throughput (GB/s) for full GPU utilization vs. model size (parameters, 100M to 175B). The red curve shows storage demand growing with model size; horizontal dashed lines mark the ceilings for common storage tiers (NVMe at 7 GB/s, Object at 50 GB/s, Lustre at 100 GB/s). Between the Compute-Bound and Storage-Bottleneck regions lies the transition where storage, not compute, limits training throughput.

The storage-bottleneck region in figure 4.2 widens with model size: as parameter counts grow, the required throughput rises faster than commodity storage tiers can supply, pushing larger models into regimes where only parallel filesystems or sharded object storage can sustain the training feed. At small request sizes, the gap also widens between sequential and random access: at 4 KB, sequential reads outperform random reads by 10×. This is *why* datasets stored as millions of small files (the "small file problem") perform catastrophically on ML workloads, even on high-bandwidth storage: the storage service spends its time on metadata rather than payload bytes, the same failure mode that large-scale file systems identify as a metadata bottleneck (Shvachko et al. 2010). The ML-specific response is to aggregate small samples into large sequential shards (Aizman et al. 2019).

The practical consequence for our running example is stark. The 1.5 trillion tokens of training data produce roughly 6 TB of sequential token-ID reads per epoch, even though the compressed source corpus is only 3 TB. If each token were stored as an individual file (as naive data collection might produce), the metadata overhead alone would throttle throughput to a fraction of what the storage hardware can deliver. Instead, the data must be preprocessed into large sequential shards so that each read operation amortizes the fixed overhead of file open, seek, and close across many tokens. This preprocessing step transforms the access pattern from metadata-heavy one-file-per-sample access to sequential shard reads (Aizman et al. 2019), keeping the workload below the storage ceiling shown in figure 4.2 and setting up the hierarchy that bridges accelerator appetite and storage capacity.

4.3 The ML Storage Hierarchy

A system architect must organize storage to serve workloads that simultaneously demand terabytes-per-second bandwidth for computation, petabyte-scale capacity for datasets, and extreme durability for checkpoints. No single technology satisfies all three requirements. HBM provides bandwidth but not capacity. Object storage provides capacity and durability but not bandwidth. The resolution is a multi-tier hierarchy that places small amounts of fast, expensive storage close to the accelerator and large amounts of slow, cheap storage at the periphery. Each tier exists because it resolves a specific tension between physics (bandwidth and latency are governed by distance from the accelerator) and economics (cost per bit decreases as capacity increases). The hierarchy extends the classic processor memory hierarchy (registers, L1/L2 cache, DRAM) that students encounter in computer architecture courses, adding tiers below DRAM that are unique to large-scale data systems. Table 4.2 reveals the extreme bandwidth disparities that ML systems must navigate.



Bandwidth drops roughly 479× across the top three tiers, from HBM to local NVMe.

Table 4.2: Extended Memory Hierarchy for ML Systems: The roughly 30× aggregate-to-aggregate bandwidth gap between HBM and object storage (and a much larger per-client gap once a single inference instance pulls from a shared object endpoint) drives the need for sophisticated prefetching and caching across multiple levels.

Storage Tier	Typical Capacity	Bandwidth	Latency	Cost ($/GB)
GPU HBM	80 GB	3.35 TB/s	~100 ns	~15.00
Host DRAM	512 GB–2 TB	200 GB/s	~100 ns	~3.00
Local NVMe SSD	4–30 TB	7–25 GB/s	~100 µs	~0.10
Parallel File System	100+ PB	1+ TB/s aggregate	~1 ms	~0.03
Object Storage	Very large pool	100 GB/s aggregate	~50 ms	~0.02
Archive/Cold Storage	Very large pool	1 GB/s	Minutes to hours	~0.004

One reading of table 4.2 is worth flagging: HBM's advantage over host DRAM is *bandwidth*, not *latency*. Random-access latency for both technologies sits in the ~80–150 ns range; HBM achieves its bandwidth lead (10–20×) through thousands of parallel traces on a silicon interposer and 3D stacking, not faster cells. The latency cliff between adjacent tiers only opens up at NVMe and below, where the physical distance from the accelerator grows from millimeters to meters.

The storage hierarchy principle (principle 7) governs every design decision in this chapter. Storage performance decreases and capacity increases as data moves further from the accelerator: each tier in table 4.2 drops bandwidth by 10–100× while increasing capacity by 10–100×. Data format choices, caching strategies, prefetch buffer sizing, and tiering policies all exist to manage the movement of data upward through the hierarchy so that the accelerator never starves.

Figure 4.3 maps these six tiers into a spatial hierarchy, showing how bandwidth decreases and capacity increases at each step away from the accelerator.

The pyramid in figure 4.3 encodes a fundamental trade-off: every step down the hierarchy trades bandwidth for capacity and cost. This trade-off is not arbitrary; it reflects the physics of data proximity. HBM sits on the same silicon interposer as the accelerator, connected by thousands

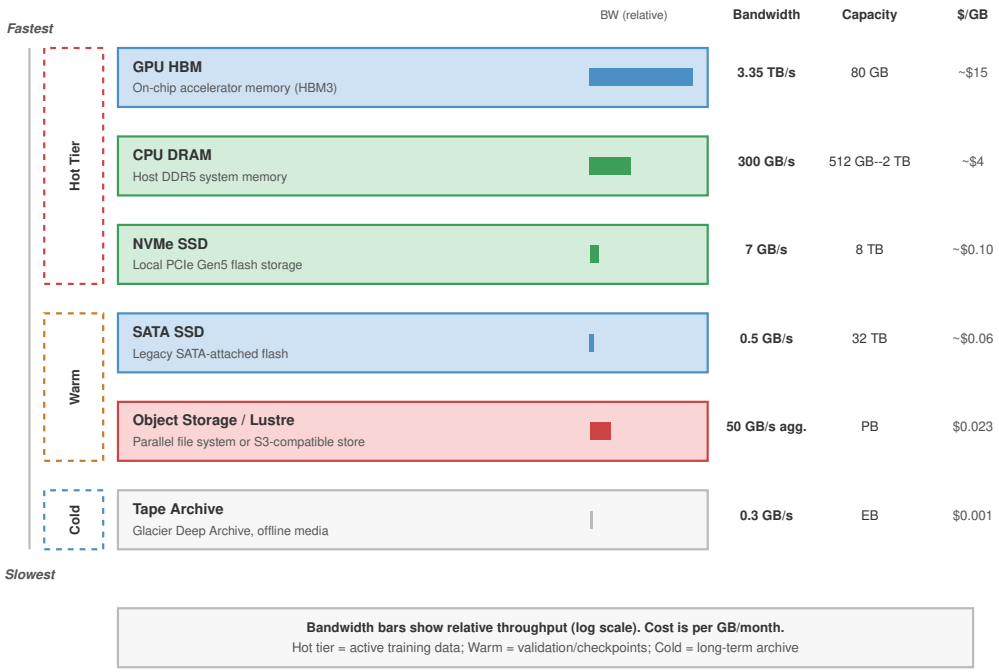


Figure 4.3: ML Storage Hierarchy: Storage tiers grouped into Hot, Warm, and Cold bands, from GPU HBM (terabytes per second, gigabytes of capacity) down to tape archive (sub-gigabyte-per-second bandwidth, very large capacity), with cost per gigabyte changing inversely. The figure adds two tiers beyond the five-level model the prose develops, a legacy SATA SSD tier and the tape archive, to show the full span of the trade-off; the shared object storage and parallel file system tier is filled red because it is the bottleneck that starves a large cluster, the tier whose bandwidth cliff the rest of the chapter works to hide. The figure uses illustrative values to make the bandwidth-vs.-capacity trade-off visible at a glance; the canonical per-tier numbers used throughout this chapter come from table 4.2, which is the source of truth where the figure and the table disagree.

of parallel traces measured in millimeters. Host DRAM communicates over PCIe lanes spanning centimeters. NVMe reaches across a circuit board via a PCIe connector. The parallel file system traverses meters of cable and network switches. Object storage may span kilometers of fiber between data centers. At each level, the increasing physical distance translates directly into increased latency, decreased bandwidth per connection, and decreased cost per byte (because the same medium can store more data at lower density).

The engineering challenge is to ensure that data flows upward through the pyramid fast enough that the top tier (HBM) is never empty when the accelerator needs it. Return to our running example: the durable corpus lives in object storage (Tier 4) as a 3 TB compressed source copy plus 6 TB of tokenized training shards, but the accelerator needs each batch in HBM (Tier 0) within 200 ms. The data must be promoted through intermediate tiers, staged in progressively faster storage, so that by the time the accelerator requests a batch, it is already waiting in host DRAM, one PCIe transfer away from HBM.

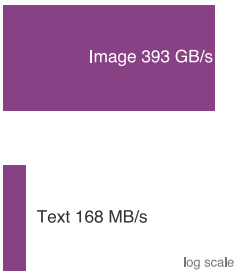
4.3.1 Bandwidth cliffs between tiers

The bandwidth ratios between adjacent tiers reveal the severity of each transition in the hierarchy. Between HBM and host DRAM, the ratio is roughly 16.8× (3.35 TB/s vs. ~200 GB/s). Between host DRAM and NVMe, the ratio is roughly 7.1× (200 GB/s vs. ~28 GB/s from a 4-drive RAID-0). Between NVMe and a parallel file system, the ratio depends on the per-node allocation: if a 1 TB/s aggregate PFS serves 256 nodes, each node receives roughly 4 GB/s, a 7× reduction from local NVMe. Between the parallel file system and object storage, the ratio is typically 10× or more, depending on the number of concurrent clients and network bandwidth.

These bandwidth cliffs have a critical implication: the pipeline cannot simply "stream through" the hierarchy in real time. If the accelerator consumes data at 3.35 TB/s from HBM, and the next

tier down delivers only 200 GB/s, then HBM can be emptied in under 50 ms but takes 400 ms to refill from host DRAM. The only way the accelerator avoids stalling is if the batch it needs next is already in HBM before it finishes the current batch. This is *why* every tier in the hierarchy serves as a prefetch buffer for the tier above it: host DRAM buffers data for HBM, NVMe buffers data for host DRAM, the parallel file system buffers data for NVMe, and object storage is the ultimate source of truth. Each buffer must be deep enough to absorb the latency and bandwidth variance of the tier below it.

The bandwidth arithmetic also explains *why* increasing cluster size creates storage pressure. A single node with 8 GPUs needs roughly 4 to 40 GB/s of storage bandwidth (depending on workload). A cluster of 256 such nodes needs 1,000 to 10,000 GB/s. A cluster of 10,000 nodes needs 40 to 400 TB/s. At the upper end, even a world-class parallel file system with 1,000 object storage servers (the data-serving nodes introduced in the parallel file system tier below) delivering 1 TB/s aggregate cannot satisfy the demand, and the architecture must rely on local NVMe caching to reduce the load on shared storage. The severity of that cluster-level pressure depends sharply on the data modality.



Storage bandwidth demand swings wildly with data modality.

Napkin Math 4.2: Text vs. image bandwidth

Problem: How much aggregate storage bandwidth does a 2,048-GPU cluster require for text training versus image training?

Setup: The bandwidth demand depends entirely on the data modality.

For **text training**, the demand is surprisingly low. With a typical batch size of 4,096 tokens per GPU and a 200 ms step time, the aggregate bandwidth is:

$$2{,}048 \text{ GPUs} \times 4{,}096 \text{ tokens/GPU} \times 4 \text{ bytes/token} \div 0.2\text{s} \approx \mathbf{167.8MB/s}$$

This is easily served by a single network-attached storage node.

For **image training**, the picture changes dramatically. Using a common batch size of 256 per GPU (ImageNet at 224×224, roughly 150 KB/image), the aggregate bandwidth explodes:

$$2{,}048 \text{ GPUs} \times 256 \text{ images/GPU} \times 150 \text{ KB/image} \div 0.2\text{s} \approx \mathbf{393.2GB/s}$$

This is about 2,300× higher than the text workload and requires a high-performance parallel file system. This fundamental difference drives hierarchy design: text training is *volume-heavy* but *bandwidth-light*, bottlenecked by total dataset size and checkpointing; image training is *bandwidth-heavy*, bottlenecked by the storage system's ability to feed the accelerators.

The bandwidth cliff between tiers also has implications for the data format at each level. At the HBM tier, data must be in the format the accelerator can directly compute on: float16 tensors, packed token IDs, or preprocessed feature vectors. At the NVMe tier, data can be in a more compact format (compressed JPEG, tokenized text with dictionary encoding) because the CPU has time to decode it while the accelerator processes the previous batch. At the object storage tier, maximum compression is desirable to minimize both storage cost and transfer time, even if decompression adds CPU overhead. The format transition from compressed storage to compute-ready tensors is part of the pipeline's "value-added" work, transforming raw bytes into the representation that the accelerator needs. This transformation happens in host DRAM, which is *why* host DRAM serves as the critical staging area for the pipeline.

The format challenge intensifies for multi-modal training, which combines text, images, audio, and video in a single model. Each modality has a dramatically different data profile: a text token is 4 bytes, a high-resolution image is 150 KB, and a short video clip is 10 MB. They also have different compression characteristics and require different augmentation pipelines. A multi-modal training job must manage multiple parallel data streams, each with its own bandwidth profile and prefetch requirements. The storage hierarchy must be provisioned for the sum of all modalities' bandwidth demands, not the dominant one alone. For a training job combining 3 TB of text with 50 TB of images and 200 TB of video, the video modality overwhelmingly dominates both storage capacity and I/O bandwidth requirements, even though the text modality may contribute more to model quality.

This asymmetry between storage cost and training value is a recurring challenge in multi-modal system design.

The descent starts where computation actually consumes bytes: HBM, the tier fast enough to keep accelerator arithmetic fed but far too scarce to hold the full training corpus.

4.3.2 Tier 0: GPU HBM

The most constrained resource in the entire system is also the most scarce. High Bandwidth Memory (HBM) is the only storage tier where weights and activations can reside during active computation. As established in section 2.3.1, HBM is a 3D-stacked memory technology that places DRAM dies vertically atop the accelerator, connected by thousands of through-silicon vias that provide aggregate bandwidth of 3.35 TB/s on an H100. This bandwidth is roughly 17× higher than the chapter's 200 GB/s host-DRAM figure and roughly 480× higher than a single 7 GB/s NVMe drive. That bandwidth is what makes large-scale deep learning feasible: a matrix multiplication involving billions of parameters requires reading those parameters from memory every forward and backward pass.

The constraint at this tier is *capacity*, not *bandwidth*. An H100 provides 80 GB of HBM, enough to hold a 40-billion parameter model in FP16 (2 bytes per parameter), but nowhere near enough for the 175B parameter running example. To understand the severity of this constraint, consider the memory budget for training our 175B model. The model weights in FP16 consume 350 GB. The Adaptive Moment Estimation (Adam) optimizer maintains two additional states (momentum and variance) in FP32, consuming $175 \times 10^9 \times 4 \times 2 = 1{,}400$ GB. Activations for a single batch, depending on sequence length and batch size, can consume another 100 to 400 GB. The total memory footprint reaches roughly 1.85–2.15 TB depending on activation size, approaching or exceeding 25× the capacity of a single H100's HBM. That 25× is the un-partitioned footprint, which is precisely why the partitioning in table 4.3 is mandatory. From the storage hierarchy perspective, HBM is the destination that every lower tier exists to serve. The data pipeline's purpose is to ensure that the 80 GB of HBM always contains the data the accelerator needs *next*, not the data it needed *a second ago*.

Because HBM capacity is so limited relative to both model size and dataset size, the accelerator processes data in batches. Each batch occupies a fraction of HBM for the duration of one forward-backward pass, then is discarded to make room for the next. The rate at which batches must be supplied sets the bandwidth requirement for all lower tiers.

That batch lifecycle creates a "spill" dynamic down the hierarchy. When weights alone do not fit in one HBM pool, the system has only two basic choices: split the live weights across accelerators, or keep some state outside HBM and fetch it when needed. For our 175B parameter model, FP16 weights occupy 350 GB, requiring at least five 80 GB H100 accelerators just for weight storage, with no room left for activations or optimizer state. Adam's FP32 momentum and variance add another 1.4 TB. Sharding this footprint across the cluster makes the run possible, but every shard creates more lower-tier traffic and more coordination. The storage point is that HBM scarcity is what forces the rest of the hierarchy to exist.

The razor-thin margin is a defining feature of large model training. Table 4.3 shows the HBM memory budget for training our 175B parameter model on a single H100 GPU with 80 GB of HBM. The table treats partitioning as a storage layout: model weights are split eight ways inside the node, while optimizer and gradient state are sharded across 256 nodes. The purpose is not to derive the partitioning algorithm, but to show the storage pressure HBM sees.

Table 4.3: HBM Memory Budget for 175B-Parameter Training: Per-GPU allocation when splitting a 175B model eight ways inside the node and sharding optimizer and gradient state across 256 nodes. The optimizer and gradient rows are per-GPU shards after both levels of partitioning. The resulting budget occupies most of a 80 GB HBM device, leaving limited buffer for the incoming data pipeline. Any delay in fetching the next batch from host memory risks starving the accelerator.

Component	Size per GPU
Model Weights (FP16, 8-way shard)	43.75 GB
Optimizer State (node-sharded)	0.7 GB
Activations (variable by sequence length)	10–20 GB
Gradient Buffers (FP16, node-sharded)	0.2 GB

Component	Size per GPU
Communication Buffers (NCCL)	2–4 GB
Total Occupied	**56.6–68.6 GB**

Even with aggressive partitioning, the total memory footprint reaches 56.6–68.6 GB on a decimal gigabyte basis—about 65.9–79.9 percent of the 80 GB HBM pool when measured against its binary capacity. This leaves a limited buffer for the incoming data pipeline, especially once fragmentation and framework workspaces are included. Any delay in fetching the next batch from host memory risks starving the accelerator, forcing it to sit idle while the most expensive resource in the system produces heat instead of gradients.

The batch lifecycle within HBM illustrates how transient storage at this tier truly is. When a new training batch arrives from host DRAM via PCIe, it is placed in a preallocated input buffer in HBM. The forward pass reads the input data, reads the model weights (which persist across batches), and writes activations to HBM. The backward pass reads the activations, computes gradients, and writes gradient updates. The optimizer step reads gradients and model weights, computes updated weights, and writes them back. After the optimizer step, the input batch and activations are no longer needed and their HBM regions are freed for the next batch. The entire lifecycle of an input batch in HBM, from arrival to deallocation, spans a single training step: typically 100 to 500 ms. Model weights and optimizer state, by contrast, persist in HBM for the entire training run, occupying a fixed allocation that cannot be reclaimed for batch data.

From the data pipeline's perspective, Tier 0 is not a storage tier to be managed but a constraint to be satisfied. The pipeline's purpose is to ensure that the batch the accelerator needs *next* is already resident in HBM before the *current* batch's computation completes. If it arrives late, the accelerator stalls. If it arrives early, it consumes HBM that could hold activations. The tension between "just in time" and "just too late" defines the pipeline's buffer management strategy, which we quantify in section 4.4.

4.3.3 Tier 1: Host DRAM

One level below HBM, host DRAM serves as the staging area for the data pipeline. Every byte of training data that reaches the accelerator passes through host DRAM first (unless GPUDirect Storage bypasses it, as described in section 4.5). A typical training node contains 512 GB to 2 TB of system memory shared across the host CPU and its peripherals. While the bandwidth between host DRAM and the accelerator is limited to what PCIe Gen 5 (about 64 GB/s per direction, 128 GB/s bidirectional) or NVLink (900 GB/s) can provide, host DRAM plays three critical roles in the ML storage hierarchy.

The data loader pipeline that runs in host DRAM follows four ordered stages:

1. **Read**: I/O threads read compressed data from NVMe or network storage into read buffers.
2. **Decode**: Decode threads decompress the data, such as JPEG decoding for images or decompression for text.
3. **Augment**: Augmentation threads apply transformations, including random cropping, flipping, and normalization for images or tokenization and sequence packing for text.
4. **Collate**: The collation stage assembles individual samples into batches and places them in **pinned memory** for efficient DMA transfer to the accelerator.

That final placement matters because pinned, or page-locked, memory (exposed in PyTorch as `pin_memory=True` on the `DataLoader`) prevents the operating system from paging out the buffer before the DMA transfer completes, allowing the NVMe controller to write directly into a physical address that the PCIe DMA engine can reach without an extra copy. Each stage runs concurrently, forming a pipeline that overlaps I/O, CPU computation, and data transfer. The efficiency of this pipeline determines whether host DRAM can keep up with the accelerator's appetite.

Host DRAM's most critical function is serving as a **prefetch buffer**. The CPU data loader reads data from lower tiers (NVMe or network storage), decodes compressed formats (JPEG, gzip), applies augmentations (random crops, flips, color jitter), and assembles tensors in host DRAM. By the time

the accelerator finishes processing batch i, batch $i+1$ should already be assembled in host memory, ready for transfer to HBM. The depth of this prefetch buffer determines how much I/O variance the pipeline can absorb without stalling.

Recommendation workloads place a different demand on host DRAM: hosting **embedding tables** that can exceed 100 GB, far too large for HBM. These tables reside in host DRAM and are accessed through lookups that fetch only the rows needed for the current batch. The bandwidth between host DRAM and HBM becomes the critical bottleneck for these workloads, which is *why* some systems use CPU-side DRAM with RDMA to serve embedding lookups across the network.

Host DRAM also provides the **augmentation workspace** that the CPU pipeline requires. Data augmentation operations (resizing images, tokenizing text, applying noise) execute on the CPU and require temporary memory for intermediate results. A training pipeline that applies five augmentations to a 256-image batch at 150 KB per image needs tens of megabytes of working space for each augmentation stage. Although modest per-batch, this memory accumulates when multiple data loader workers run in parallel.

Some augmentation pipelines have moved from CPU to GPU execution, using libraries like NVIDIA Data Loading Library (DALI) to perform image decoding and augmentation on the accelerator itself. This approach eliminates the CPU augmentation bottleneck and reduces the host DRAM bandwidth demand, because compressed data (smaller) is transferred to the GPU instead of decoded data (larger). The trade-off is that augmentation on the GPU consumes HBM capacity and compute cycles that would otherwise be available for training. For *compute-bound* workloads (where the GPU is already saturated with matrix multiplications), GPU-based augmentation is counterproductive. For *I/O-bound* workloads (where the GPU waits for data), it can improve overall throughput by shifting work from the bottleneck (CPU) to the resource with spare capacity (GPU).

The three roles interact in subtle ways. The prefetch buffer and the augmentation workspace compete for the same physical DRAM, and embedding tables consume capacity that could otherwise serve as deeper prefetch queues. A node with 512 GB of DRAM hosting a 200 GB embedding table has only 312 GB remaining for prefetching and augmentation. If the data loader uses 8 workers, each maintaining a decode buffer of 1 GB, the effective prefetch capacity drops further. System architects must balance these competing demands by profiling the actual memory consumption of each pipeline stage and provisioning DRAM accordingly.

The physical layout of DRAM has performance implications that are invisible in single-socket benchmarks but critical at production scale. Multi-socket servers exhibit Non-Uniform Memory Access (NUMA) topology where each CPU socket has "local" DRAM that it can access at full bandwidth and "remote" DRAM attached to the other socket at roughly half bandwidth. In a dual-socket DGX node with eight GPUs split four per socket, a data loader thread running on socket 0 that allocates its prefetch buffer in socket 1's DRAM pays a roughly 2× bandwidth penalty on every buffer access. The fix is **NUMA-aware allocation**: pin each data loader worker to the same CPU socket (the same NUMA domain) as the GPUs it serves, and use `numactl` or `libnuma` to ensure memory allocation stays local. Proper NUMA pinning can improve data loading throughput by 30–50 percent on dual-socket systems, a gain that is invisible in development environments but essential when every percentage point of utilization translates to thousands of dollars per day.

The gap between host DRAM bandwidth and HBM bandwidth is the first major cliff in the hierarchy: roughly 16.8× using the chapter's 200 GB/s host-DRAM reference and 3.35 TB/s HBM figure. The host-to-GPU interconnect adds a separate cliff, with effective transfer bandwidth depending on whether the node uses PCIe or NVLink. Any failure to keep host DRAM populated from lower tiers cascades immediately to accelerator starvation, because the accelerator cannot fetch directly from NVMe or network storage. In our running example, the 256-node cluster requires each node's host DRAM to sustain a continuous flow of decoded, augmented batches ready for PCIe transfer. If the NVMe-to-DRAM read pipeline falls behind by even a few hundred milliseconds, the prefetch buffer drains and the accelerator idles until the next batch arrives.

4.3.4 Tier 2: Local NVMe

When the working set exceeds host DRAM, the system falls to Tier 2: the local NVMe drives attached directly to the compute node. NVMe[4] provides a high-performance protocol designed specifically for solid-state drives, achieving 7 GB/s of sequential throughput per drive. With four drives in a

4 **NVMe (Non-Volatile Memory Express)**: A storage protocol designed for SSDs, connecting directly to the CPU via PCIe lanes. NVMe replaced Advanced Host Controller Interface (AHCI)'s single queue of 32 commands with 64K queues of 64K commands each, a 130-million-fold increase in maximum outstanding I/O. This deep queue parallelism is what allows a multi-worker ML data loader to saturate the drive's bandwidth; with AHCI, 32 workers would serialize on the single command queue regardless of flash speed.

RAID-0 configuration, a single node can sustain roughly 28 GB/s of sequential reads before overhead, sufficient to stream the 6 TB serialized dataset from local disk in about 3.6 minutes under ideal sequential conditions.

In ML training, local NVMe acts as a **warm cache**, storing data shards fetched from distributed storage. This design allows workers to re-read samples across multiple epochs without re-traversing the network. For multi-epoch training on petabyte-scale datasets, the network egress cost of re-fetching from object storage each epoch would be prohibitive (as we quantify in section 4.6). Populating local NVMe from shared storage at job start and reading locally thereafter eliminates both cost and latency.

The warm cache pattern requires careful capacity planning. A training node with four 7.68 TB NVMe drives provides approximately 30.7 TB of local storage. For our running example, the 6 TB serialized dataset fits comfortably on a single node's local storage, with room for checkpoint staging and temporary augmentation buffers. A multi-modal training job combining 20 TB of images, 10 TB of text, and 5 TB of audio totals 35 TB, exceeding local capacity by about 4.3 TB and forcing the pipeline to stream from the parallel file system for at least part of the dataset. The design trade-off is between provisioning more local NVMe (which increases node cost) and accepting network-dependent reads (which risks latency spikes).

NVMe's internal parallelism is key to its throughput advantage over traditional storage. The NVMe specification supports up to 65,535 I/O queues, each with up to 65,536 outstanding commands. A data loader with 32 workers, each issuing asynchronous reads, can keep the NVMe controller's internal pipeline saturated. In contrast, the legacy AHCI protocol that NVMe replaced supported a single queue of 32 commands, throttling parallelism at the protocol level regardless of the underlying medium's capability. This architectural difference explains why NVMe delivers $10\times$ to $50\times$ the throughput of SATA SSDs with identical NAND flash, even though the storage medium is the same.

Data format design for sequential I/O

The choice of data format on local NVMe has a dramatic impact on effective throughput. Consider three approaches to storing a 1.28 million image dataset.

The first approach stores each image as a separate JPEG file in a directory hierarchy. This format is natural for data collection (download one image, save one file) but adversarial for training I/O. Each `open()` system call has a fixed overhead of roughly 10–50 µs in the kernel's Virtual File System (VFS) layer. At 8,000 images per second, the overhead alone consumes 80–400 ms of CPU time per second. Worse, the directory structure forces the file system to maintain an inode for each file, consuming metadata resources that would otherwise be available for data reads.

The second approach packs all images into a small number of large binary files (such as HDF5, LMDB, or raw concatenated tensors with an index file). Each file contains thousands of images stored contiguously, and a separate index maps sample IDs to byte offsets within the file. The data loader seeks to the desired offset and reads the sample directly. This eliminates the per-file metadata overhead and enables sequential access within each binary file. The disadvantage is that the dataset is no longer human-readable, and modifying a single sample requires rewriting the entire file.

The third approach uses the tar-based archive format popularized by WebDataset. Each sample is stored as a group of related files (image, label, metadata) within a standard POSIX tar archive. The tar format supports sequential iteration without a separate index, because each file's header contains its size, allowing the reader to skip forward to the next sample. This format combines the simplicity of individual files (each sample is self-describing) with the sequential I/O efficiency of large binary files. The tar archives are also valid HyperText Transfer Protocol (HTTP) byte-range targets, making them directly streamable from object storage without local staging.

For our running example, the 1.5 trillion token dataset is typically stored as a collection of 256 MB to 4 GB binary shards, each containing a contiguous sequence of tokenized text. The data loader opens a shard, reads it sequentially into a buffer, and iterates over tokens within the buffer. When the buffer is exhausted, the loader opens the next shard. The total number of `open()` calls per epoch is the number of shards (roughly 2,000 if the 6 TB serialized corpus is split into 3 GB shards), not the number of tokens (trillions). This approximately 750,000,000× reduction in metadata operations is what makes streaming from both local NVMe and remote storage feasible at training scale.

At the NVMe tier, compression represents a critical trade-off between I/O bandwidth and CPU cycles. An I/O-bound pipeline, where the NVMe drives cannot keep up with accelerator demand,

benefits from aggressive compression: zstd at level 9 achieves roughly 4:1 compression but decompresses at only 0.5 GB/s per CPU core. A CPU-bound pipeline, where decode and augmentation already saturate the host processor, prefers lighter compression: zstd at level 1 offers roughly 3:1 compression but decompresses at 1.5 GB/s per core. On a 28 GB/s four-drive NVMe RAID array, zstd-1 delivers an effective throughput of 84 GB/s of uncompressed data, while zstd-9 delivers 112 GB/s but requires 3× more CPU cores dedicated to decompression. The optimal compression level is therefore not a property of the data but a property of the pipeline's bottleneck, and it can change when the cluster configuration changes (adding more GPUs shifts the bottleneck toward I/O, favoring heavier compression).

A standard ImageNet training pipeline makes the cost of the wrong format visible.

Napkin Math 4.3: The ImageNet bottleneck analysis

Problem: A ResNet-50 training job on ImageNet (1.28M, ~150 KB average) targets 1000 images/s. The question is whether to use individual JPEG files on an HDD or NVMe.

Math:

1. **Raw Bandwidth**: 1000 images/s × 150 KB = 150 MB/s.
2. **HDD reality**: A 7200 RPM hard disk drive (HDD) delivers 100 IOPS at random reads. Sustaining 1000 images/s requires 10× more IOPS than the disk provides.
3. **Result**: Shuffling individual files on an HDD will starve the GPU, capping utilization at about 10 percent before decode, seek, and file-system overheads.

Systems insight: The pipeline must either use NVMe (~100 µs random access, three orders of magnitude faster than object storage) or convert the dataset to a sequential format (TFRecord/WebDataset) to achieve sequential throughput.

The central challenge at this tier is the I/O wall: a single NVMe drive is roughly 500× slower than HBM, and even a four-drive RAID-0 stripe remains roughly 120× slower. Bridging this gap requires pipelining (overlap I/O with compute, detailed in section 4.4) and, increasingly, GPUDirect Storage (detailed in section 4.5) to bypass CPU overhead entirely. The I/O wall at this tier is particularly insidious because NVMe performance is excellent by historical standards. A storage engineer accustomed to HDD-era throughput of 100 MB/s might view 28 GB/s from a local RAID-0 stripe as superabundant. Relative to the accelerator's appetite, however, 28 GB/s is a trickle. The only way to bridge the gap is to overlap storage reads with computation so thoroughly that the accelerator never perceives the storage delay.

Local NVMe is also the primary tier for **local checkpoint staging**. When saving a model checkpoint, the fastest strategy is to write to local NVMe at full bandwidth (minimizing the time the training pipeline is paused), then asynchronously replicate to shared storage for durability, since a per-node shard written directly to a contended parallel file system takes several times longer than the same shard written to local drives. The optimal checkpoint frequency depends on cluster failure rates and checkpoint write time, but the storage-side design goal is already clear: minimize T_{write} through tiered staging by writing to local NVMe at full bandwidth, then background-copying to shared storage. Section 4.7 works the timings through in full.

The same local NVMe tier also pays for itself by recovering utilization lost to remote-read latency.

Napkin Math 4.4: ROI of local NVMe caching

Problem: A vision-model training pipeline runs each step in 800 ms. Fetching data from a shared Parallel File System adds 150 ms of I/O wait because of network congestion. How much does adding local NVMe SSDs to each node improve GPU utilization?

Math: GPU utilization (η_{hw}) is the fraction of step time spent in computation.

1. **Remote Only**: 800 ms / (800 ms + 150 ms) ≈ 84.2 percent.

2. **Local Cache**: Using prefetching into local NVMe reduces the exposed I/O wait to near zero.
   - New Util: 800 ms / (800 ms + 10 ms) $\approx$ 98.8 percent.

Systems insight: Local storage is a *GPU utilization multiplier*. In this scenario, adding a $500 NVMe drive to a $30,000 GPU node recovers 14.6 percent of the GPU's capacity that was previously wasted on I/O wait. Across a 1,024-GPU cluster, that utilization gain is equivalent to adding 149 GPUs worth of useful work without buying that many more accelerators. In ML infrastructure, local NVMe is not auxiliary; it is the physical buffer that decouples expensive compute from unpredictable shared storage.

A practical concern at this tier is **SSD endurance**. NAND flash memory can sustain a limited number of write-erase cycles before the cells degrade. Enterprise NVMe drives are rated for 1 to 3 Drive Writes Per Day (DWPD) over a 5-year lifespan. For a 7.68 TB drive at 1 DWPD, this means the drive can absorb 7.68 TB of writes per day, or roughly 14 PB total over its lifetime. ML training workloads are predominantly read-heavy (the dataset is written once and read many times), which is favorable for SSD endurance. However, checkpoint writes can be intensive: if each node saves a 4 GB checkpoint shard every 10 minutes, that is 576 GB per day of checkpoint writes, well within the 1 DWPD budget. The risk emerges when local NVMe is used as a staging buffer for both checkpoint writes and dataset caching: the combined write volume from initial dataset staging plus repeated checkpoint saves must remain within the drive's endurance rating.

Local NVMe provides high bandwidth and low latency within a single node, but distributed training requires every node to access the same datasets and see the same checkpoints. This shared-namespace requirement cannot be satisfied by node-local storage alone and motivates the next tier in the hierarchy.

4.3.5 Tier 3: Parallel file systems

Beyond the single node, the workload requires a shared namespace where all workers can access the same datasets and where durable checkpoints are globally visible. This is the role of the parallel file system (PFS).[5]

[5] **PFS (Parallel File System)**: A family of distributed file systems (Lustre, GPFS/Spectrum Scale, BeeGFS, WekaFS) whose defining property is that a single client reads from multiple storage servers simultaneously, aggregating their bandwidth into one logical stream. For ML training, this means a single data shard striped across 100 servers can deliver 100$\times$ the bandwidth of any individual server, the architectural feature that makes petabyte-scale dataset access feasible within training-iteration time budgets.

Definition 4.1: Parallel file system

Parallel File System (PFS) is a distributed storage architecture for ML training clusters that stripes data across many storage servers to provide aggregate throughput exceeding the capacity of any single device.

1. **Significance**: A PFS aggregates BW_{io} linearly with the number of storage servers (Object Storage Servers). A Lustre cluster with 20 OSS nodes each delivering 10 GB/s provides 200 GB/s aggregate, vs. a single NAS server capped at 10 GB/s, enabling a training job to load a 10 GB striped shard in about 50 ms rather than 1 second. This aggregate bandwidth directly reduces the D_{vol}/BW term in the iron law.
2. **Distinction**: Unlike Network Attached Storage (NAS), where every I/O request routes through a single server, a PFS client receives stripe location metadata from a dedicated Metadata Server (MDS) and then reads data directly from multiple OSS nodes in parallel: the MDS and OSS paths are architecturally separated, so data bandwidth scales with OSS count while metadata operations scale with MDS count.
3. **Common pitfall**: A frequent misconception is that a PFS has unlimited throughput if enough OSS nodes are added. In reality, a Lustre MDS handles roughly 100,000–300,000 metadata operations per second; at 10,000 workers each opening one small file, the MDS saturates in under 1 second and becomes the serialization point that idles the entire cluster regardless of how many OSS nodes are present.

The architecture of a Lustre-style parallel file system separates two concerns that traditional file systems handle together (Schwan 2003). **Metadata Servers (MDS)** manage the namespace: file creation, directory listings, permission checks, and lock management. **Object Storage Servers (OSS)**[6] manage the actual data blocks, each serving a stripe of every large file. When a client opens a 10 GB training shard, the MDS tells the client which OSS nodes hold which stripes, and the client reads from all of them in parallel. A Lustre[7] deployment with 100 OSS nodes, each providing 10 GB/s, delivers an aggregate 1 TB/s.

This abstract architecture has concrete implications for performance tuning. When a training job creates a new dataset directory on Lustre, the administrator configures the **stripe count** (the number of OSS nodes a file is spread across) and **stripe size** (the chunk size written to each OSS) for that directory (Lustre Wiki 2017). For large sequential-read training shards, administrators may choose wider striping and multi-megabyte stripe sizes so a single file can draw bandwidth from multiple OSS nodes; Lustre's striping guide treats wide striping as useful for very large files or files accessed by many clients, while smaller files often use fewer stripes to reduce overhead (Lustre Wiki 2019). A 4 GB data shard striped across 100 OSS nodes with 4 MB stripes places roughly 10 stripes on each OSS. When a data loader reads large contiguous ranges, the Lustre client can issue parallel read requests to the OSS nodes holding those stripes, aggregating their bandwidth. The client also maintains its own read-ahead buffer, prefetching the next several stripes while the application processes the current ones. This file-system-level read-ahead is distinct from the data loader's application-level prefetch buffer; the two layers of prefetching compound to provide deep latency hiding, making the physical distance to the OSS nodes nearly transparent to the training process.

The separation between metadata and data paths is what enables the aggregate bandwidth that ML workloads demand. In GPFS[8] (IBM Spectrum Scale), the architecture takes a different approach: it stripes data and metadata across shared disks and coordinates concurrent access through token-based distributed locking (Schmuck and Haskin 2002). The result is not a simple MDS/OSS split; it is a shared-disk design whose metadata and data placement can be distributed while the lock protocol preserves consistency across clients. The trade-off is greater complexity in lock management, which must scale to thousands of nodes.

This separation creates a critical bottleneck: the small file problem. If a dataset consists of millions of 10 KB images stored as individual files, the metadata load overwhelms the MDS long before the data links saturate. Each `open()` system call requires a metadata lookup, lock acquisition, and attribute fetch. With 10,000 workers simultaneously calling `open()` on different files, the MDS becomes the serialization point. The throughput of the storage system collapses to the rate at which the MDS can process metadata operations, typically a few hundred thousand per second, far below what the data path could deliver.

Definition 4.2: Small file problem

Small File Problem is an ML data-loading pathology where millions of individually small files overwhelm the metadata server of a storage system.

1. **Significance**: It reduces effective I/O Bandwidth (BW_{io}) to a fraction of its theoretical rating because each file requires its own metadata operations (`open`, `stat`, `close`). With 10,000 workers simultaneously accessing small files, the metadata server becomes a Serialization Point that idles the entire cluster.
2. **Distinction**: Unlike bulk data throughput (which measures bit-rate), the small file problem is a metadata latency (L_{lat}) issue: the bottleneck is the frequency of requests, not the size of the data.
3. **Common pitfall**: A frequent misconception is that this is "fixed" by buying faster SSDs. In reality, it is a Format Problem: the fix is to pack samples into large sequential containers (for example, TFRecord, Parquet) to Amortize metadata operations across thousands of samples.

At production scale, the metadata bottleneck can collapse an otherwise high-bandwidth file system.

6 **OSS (Object Storage Server)**: In parallel file system terminology, "object" means a chunk of striped file data managed by an Object Storage Target (OST), a usage that predates cloud object storage (S3, GCS) by over a decade. Confusing the two is a common source of miscommunication: when a Lustre administrator says "add more OSS nodes," they mean adding data-serving capacity to the parallel file system, not provisioning cloud buckets.

7 **Lustre**: A portmanteau of "Linux" and "cluster," developed at Carnegie Mellon University and first deployed in production in 2003. Lustre is common in HPC and ML infrastructure because its architecture scales aggregate bandwidth linearly with the number of OSS nodes, the same property that makes it a natural choice for training clusters where a single job may demand hundreds of GB/s of sustained read throughput.

8 **GPFS (General Parallel File System)**: Developed by IBM Research starting in 1998, now marketed as IBM Spectrum Scale. Unlike Lustre's dedicated metadata-server/object-server organization, GPFS is a shared-disk parallel file system that can stripe data and metadata and uses token-based locking to coordinate concurrent client access. For ML checkpoint writes, this distributed coordination model can reduce reliance on a single dedicated metadata path, but it also makes lock behavior part of the performance envelope.

Example 4.1: The metadata meltdown

Scenario: A large-scale training cluster migrates from a preprocessed sequential dataset to raw images stored as individual files.

Setup: The parallel file system is provisioned for 500 GB/s of aggregate data bandwidth, but it delivers less than 1 percent of rated throughput because the metadata servers cannot sustain the millions of `stat()` and `open()` calls per second generated by ML data loaders. Converting the 200-million-image dataset into 50,000 large tar files reduces metadata operations by four orders of magnitude.

Systems lesson: At scale, metadata operations can become the first bottleneck even when the storage fabric has ample data bandwidth.

The design response to the small file problem is **striping** and **aggregation**. Striping distributes a single large file across multiple OSS nodes so that a client can read from all of them in parallel. The stripe size (typically 1 to 4 MB per OSS) determines the granularity: a 4 GB file striped at 1 MB across 100 OSS nodes places 40 MB on each node, and a sequential read saturates all 100 data paths simultaneously. The stripe count (how many OSS nodes participate) can be configured per-file or per-directory, allowing administrators to tune bandwidth for different access patterns. Training data shards benefit from maximum striping; small configuration files benefit from minimal striping to avoid the overhead of coordinating across many nodes.

The interaction between stripe size and workload access pattern determines realized throughput. If the training data loader reads sequentially through a shard, the PFS client reads stripe 1 from OSS-A, stripe 2 from OSS-B, stripe 3 from OSS-C, and so on, naturally distributing the load across all OSS nodes that hold stripes of that file. If the read size is smaller than the stripe size, each read is served by a single OSS node, and the client does not benefit from parallel reads. If the read size spans multiple stripes, the client issues parallel reads to multiple OSS nodes simultaneously. For ML workloads that read multi-megabyte chunks (an entire batch of images, or a 4 MB block of tokenized text), the read size typically exceeds the stripe size, achieving full bandwidth aggregation.

Aggregation complements striping by reducing the number of files that the MDS must track. Small samples are bundled into large sequential files (such as WebDataset[9] tar archives or TFRecord sequences) to amortize metadata costs across thousands of samples. A single `open()` on a 4 GB tar file gives access to 40,000 samples, reducing metadata load by 40,000$\times$ compared to individual files.

Parallel file systems provide the coordinated shared namespace and lock management required for reliable checkpoints (Schmuck and Haskin 2002). When a checkpoint save completes, every node that subsequently reads that checkpoint must see the same completed state. This guarantee is essential for fault recovery: if a node fails and restarts, it must read the checkpoint that the surviving nodes wrote, not a partially flushed version. Achieving this consistency at scale requires careful coordination between the MDS lock manager and the distributed OSS write paths, which is one reason that checkpoint writes are more expensive than training reads.

The consistency model has subtleties that matter for ML workloads. For training data reads, strong consistency is unnecessary because the data is immutable: once a dataset is preprocessed and uploaded to the parallel file system, it is never modified. Multiple workers reading the same shard simultaneously can do so without locking, because there are no concurrent writers to create conflicts. This read-only access pattern allows the PFS to serve training data at near-theoretical bandwidth. Checkpoint writes, by contrast, require exclusive locks to prevent partial reads during the write, and the lock acquisition and release add latency to every checkpoint operation. Some systems mitigate this by writing checkpoints to a new file rather than overwriting the previous one, trading storage space for reduced lock contention.

The performance of a parallel file system depends not only on the *number* of OSS nodes but also on the *balance* of load across them. If a training job reads a single large shard that is striped across 10 OSS nodes while the remaining 90 nodes are idle, the job achieves only 10 percent of the system's aggregate bandwidth. Conversely, if 100 training jobs each read different shards striped across all 100 OSS nodes, the aggregate bandwidth approaches the theoretical maximum. The scheduling of

[9] **WebDataset**: A data format that repurposes standard POSIX tar archives for ML training. The design insight is that tar's sequential header-then-data layout, invented in 1979 for tape backup, maps perfectly onto the streaming access pattern that ML data loaders need. Because tar archives are valid HTTP byte-range targets, a data loader can stream training samples directly from object storage without local staging, reducing the I/O path from five tiers to two.

data access patterns across the cluster is therefore a system-level optimization opportunity that can dramatically affect realized throughput.

At the scale of thousands of nodes, tail latency (Dean and Barroso 2013) dominates system performance. The mathematics is sobering. If a training step requires data from 100 storage servers and each has a 1 percent chance of being slow (due to background maintenance, garbage collection, or network jitter), the probability that all 100 respond quickly is $0.99^{100} = 0.366$, meaning over 63 percent of training steps will experience at least one slow response. The slowest response determines the step's completion time, because data-parallel training requires all workers to complete their batch before the collective communication phase begins.

Systems mitigate tail latency through **hedged requests**: after waiting for a configurable timeout (typically the median response time), the client issues a redundant read to a different replica of the same data stripe. The first response to arrive is used; the second is discarded. If the storage system replicates each stripe across two OSS nodes, the probability that both replicas are slow is $0.01^2 = 0.0001$, reducing the fraction of slow steps from 63 percent to less than 1 percent. The cost of the redundant request, one additional network read, is negligible compared to the cost of an idle accelerator consuming hundreds of watts while waiting.

The effectiveness of hedged requests depends on the replicated data layout. If both replicas reside on OSS nodes that share the same network switch, a switch failure makes both copies simultaneously unavailable. Effective hedging requires **failure-domain-aware placement**: replicas should reside in different racks, connected through different top-of-rack switches, so that the failure of any single component affects at most one replica.

In production clusters, the parallel file system is a shared utility, simultaneously servicing dozens of concurrent training jobs with different I/O patterns. At any given moment, a large language model job might be streaming sequentially through text shards, a computer vision job might be reading random image shards, and a checkpoint storm from a third job could be saturating write bandwidth. This concurrency gives rise to the **noisy neighbor problem**, where one job's I/O pattern severely degrades performance for all other jobs. A checkpoint save that consumes a disproportionate share of the PFS's internal network bandwidth causes read latency for other jobs to spike, potentially stalling their training pipelines. PFS administrators mitigate this through I/O scheduling policies (quality-of-service tiers, bandwidth quotas per job), but these policies add administrative complexity and can reduce peak single-job throughput.

A related challenge is **namespace isolation**. Different teams and workloads require different storage configurations. A team training on millions of 100 KB images needs a directory with high stripe count and small stripe size to maximize metadata performance. A team working with large video files needs fewer, larger stripes to optimize for sequential bandwidth. Misconfigured striping for one team's workload can create hotspots that degrade performance for the entire shared file system. The impact of sharing is easy to quantify: a PFS with 1 TB/s aggregate bandwidth, shared across 10 concurrent training jobs, provides only 100 GB/s per job on average. If one job's checkpoint storm consumes 400 GB/s for 10 seconds, the remaining nine jobs share 600 GB/s, a 33 percent reduction that can trigger data stalls in their accelerator pipelines.

Checkpoint 4.2: Parallel file system design

Consider a training cluster with 512 nodes, each running 8 GPUs. The training job requires 400 GB/s of aggregate read bandwidth.

- ☐ **OSS count**: Calculate the number of OSS nodes required when each Lustre OSS delivers 10 GB/s.
- ☐ **Metadata cost**: Calculate the time required to open 200 million 150 KB image files sequentially when each `open()` call takes 50 µs on the MDS, then state the practical implication.
- ☐ **Shard count**: Calculate the number of metadata operations needed to read the full dataset when images are bundled into 4 GB tar files.

Parallel file systems solve the shared-namespace problem for active training clusters, but their cost per byte is too high for the organization's complete data holdings. The petabytes of raw training data, historical checkpoints, and preprocessed datasets that an ML organization accumulates over years require a tier optimized for capacity and durability rather than bandwidth.

4.3.6 Tier 4: Object storage

At the scale of petabytes, object storage provides the durable, cost-effective foundation of the hierarchy. Services like Amazon S3 and Google Cloud Storage organize data in object namespaces where objects are retrieved by keys rather than by POSIX directory paths (Amazon Web Services 2026c; Google Cloud 2026). Unlike file systems, which organize data in hierarchical directories with metadata-rich operations (rename, link, permission inheritance), object storage treats each object as an opaque blob identified by a unique key. This simplification eliminates the metadata complexity that plagues parallel file systems at scale and enables horizontal scaling to very large capacity pools.

Object storage targets "eleven nines"[10] of durability (99.999999999 percent) through erasure coding[11] and geographic replication. That target makes object storage the natural source-of-truth tier for training data and checkpoint archives, while still requiring versioning, lifecycle policy, access control, and recovery testing for operational failures outside the storage service's durability model.

> **Definition 4.3: Erasure coding**
>
> ***Erasure Coding*** is a storage protection scheme for ML datasets and checkpoint archives that fragments an object into k data and m parity blocks, ensuring the original is recoverable from any k remaining fragments.
>
> 1. **Significance**: It achieves extreme data durability (for example, "eleven nines") with significantly lower storage overhead than replication. For example, a 4+2 code stores four data fragments and two parity fragments, so it tolerates two unavailable fragments at 1.5× storage overhead instead of the 3× overhead of triple replication.
> 2. **Distinction**: Unlike replication (which stores complete copies), erasure coding uses mathematical encoding (for example, Reed-Solomon) to distribute redundant information across different failure domains (disks, racks, or sites).
> 3. **Common pitfall**: A frequent misconception is that erasure coding is "free durability." In reality, it is a latency-compute trade-off: reconstructing data after a failure requires additional CPU cycles and increases the tail latency (L_{lat}) of the storage read.

The engineering behind erasure coding illustrates a recurring theme in storage systems: achieving extreme durability through redundancy that is invisible to the application. When an ML training job reads a shard from object storage, the storage service transparently reads k fragments from available nodes, reconstructs the original shard, and returns it to the client. If one fragment is on a failed disk, the service reads additional coded fragments and reconstructs the missing data. The client never observes the failure, but the storage system pays extra I/O, network, and decoding work, which contributes to the higher tail latency of object storage compared to local NVMe.

The cost advantage of object storage is significant: at a representative \$0.02/GB/month, it is 750× cheaper per byte than HBM and 5× cheaper than local NVMe. For a 100 TB training dataset, object storage costs roughly \$24,000 per year under that price assumption, compared to \$120,000 for local NVMe. This cost advantage makes object storage a common home for the organization's training data lake, where raw data, preprocessed datasets, and archived model artifacts can reside durably.

The latency disadvantage is equally significant. A GET request to object storage can incur tens of milliseconds of latency; the design examples in this chapter use 50 to 100 ms as a representative planning range rather than a universal service guarantee. That latency is a consequence of the multi-layer indirection that provides durability. When a client requests an object, the storage service must look up the object's location in a distributed metadata index, identify which erasure-coded fragments to read, retrieve fragments from potentially different storage nodes, and reconstruct the original object. This overhead is invisible for large objects (where transfer time dominates) but catastrophic for small ones (where per-request latency dominates).

[10] **Eleven Nines (99.999999999 Percent Durability)**: A design target of 11 nines annual durability corresponds to an expected loss probability of about 10^{-11} per object-year; for one million objects, that is about one expected object loss per 100,000 years. This extreme durability is why object storage is the only tier where training data and checkpoint archives can be treated as permanent. The same service indirection, erasure coding, replication, and metadata lookup that support durability also make object storage a higher-latency tier than local NVMe or a warm parallel file system, so real-time data pipelines need prefetch buffering.

[11] **Erasure Coding**: Built on Reed-Solomon codes (Reed and Solomon 1960), which use maximum-distance-separable coding to recover the original data from any sufficient subset of encoded fragments. In storage systems, Reed-Solomon-style codes and later local-reconstruction variants trade storage overhead against repair bandwidth and degraded-read cost (Plank 1997; Huang et al. 2012). For ML storage, the trade-off is concrete: a 4+2 code tolerates two missing fragments at 1.5× storage overhead, while triple replication costs 3×; degraded reads then pay extra network I/O and reconstruction work when fragments are unavailable.

The solution is the same aggregation pattern used in parallel file systems: samples are bundled into multi-gigabyte shards using formats like WebDataset (Aizman et al. 2019) or Mosaic Streaming (Databricks 2026) that transform high-latency random access into high-bandwidth sequential streaming. Rather than issuing billions of small GET requests, the data loader issues hundreds of large reads in parallel, each fetching an entire shard that contains thousands of samples. With sufficient parallelism and colocated network capacity, object storage can sustain high aggregate throughput, enough to feed large training clusters for some workloads. The streaming pattern works by assigning each data loader worker a subset of shards and having it read them sequentially. Within each shard, samples are stored contiguously, so the worker decodes them in order and shuffles them in a local buffer. This approach achieves pseudo-random access across the dataset while maintaining sequential I/O at the storage level.

The architecture of a streaming data loader for object storage differs from a local-storage loader in important ways. A local-storage loader can use memory-mapped I/O to treat the dataset as a virtual memory region, relying on the operating system's page fault mechanism to load data on demand. This approach is elegant but incompatible with object storage, which does not support the POSIX file system interface that memory mapping requires. Instead, an object-storage loader must explicitly manage HTTP connections, issue ranged GET requests for specific byte ranges within shards, handle retries on transient failures, and manage a local buffer pool for decoded samples. Libraries like WebDataset and Mosaic Streaming encapsulate this complexity, presenting a simple iterator interface to the training loop while managing the HTTP transport and buffering internally.

Applying this streaming architecture to our running example makes the design concrete. The tokenized training set is stored in an S3-like object store as roughly 2,000 shards of 3 GB each. The streaming architecture assigns each of the 256 compute nodes a unique subset of roughly 8 shards. Within each node, 8 data loader worker processes issue concurrent HTTP `Range` requests to fetch 256 MB chunks from their assigned shards. Across the cluster, this results in 2,048 concurrent requests, allowing aggregate throughput in the 50–100 GB/s range when the object store and network path are provisioned for it, easily saturating the roughly 167.8 MB/s required for text model training. In this regime, the bottleneck often shifts from the object store's bandwidth to the network bandwidth between the compute VPC and the object-storage endpoint. Organizations that co-locate compute and storage in the same cloud availability zone can reduce cross-network latency, directly reducing the required prefetch buffer depth and improving overall pipeline efficiency.

Object storage exposes large request parallelism. Each GET request is independent and can be served by a different storage node within the cloud provider's infrastructure. A training cluster with 1,024 nodes, each running 8 data loader workers issuing concurrent requests, generates 8,192 simultaneous GET requests. If each request fetches a 256 MB shard, the aggregate throughput depends on the cloud provider's network capacity, internal bandwidth, and request-rate quotas. The practical limit is often the network bandwidth between the compute cluster and the object storage service, not the storage service's internal media bandwidth.

Many object storage services provide strong read-after-write consistency, which means that once a checkpoint is written to object storage, a subsequent read sees the complete data. This was not always the case: until December 2020, Amazon S3 provided eventual consistency for overwrite PUTs and DELETEs, meaning a read immediately after updating or deleting an existing object might return stale data. New-object PUTs were always strongly consistent, but checkpoint workflows that update a "latest" pointer or overwrite an existing artifact were vulnerable: a node recovering from failure might read a stale checkpoint and corrupt the training state. The December 2020 transition to strong read-after-write consistency for all operations on Amazon S3 closed this gap for S3-style checkpoint retention (Amazon Web Services 2020), while the broader design lesson is to verify the consistency semantics of the object store used by the pipeline. This consistency property makes object storage suitable for both training data source (the original dataset) and long-term checkpoint retention (the archival copy after local NVMe staging), provided the pipeline validates the service semantics it relies on.

The immense scale of ML datasets makes them vulnerable to **silent data corruption**: subtle bit flips in the storage medium or during network transfer that can introduce training artifacts nearly impossible to diagnose. Object storage services provide per-object checksums that verify integrity on upload, copy, or download; Amazon S3, for example, documents checksum options including

CRC-family and cryptographic hashes (Amazon Web Services 2026a). Parallel file systems often rely on underlying RAID or erasure coding for physical integrity but may not provide end-to-end checksums visible to the application. A robust large-scale ML pipeline computes and stores a checksum for each data shard during preprocessing and verifies it when the shard is first loaded by a training worker. The verification overhead is negligible compared to the cost of training on corrupted data. For model checkpoints, integrity verification is even more critical: a single bit flip in a weight or optimizer state can corrupt the model, causing training to diverge silently after recovery. Frameworks and checkpoint libraries can include integrity verification, but the pipeline should not assume it without testing the checkpoint format.

Storage security is a first-class concern in production ML infrastructure. Training datasets often contain sensitive information (personally identifiable data, proprietary text, licensed images) that requires access control. Object storage provides fine-grained security policies through Identity and Access Management (IAM) roles, per-bucket policies, and integrated encryption for data at rest and in transit. Parallel file systems provide POSIX permissions and access-control lists (ACLs) but often lack the audit logging that compliance requires. For organizations subject to data protection regulations such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA), the storage architecture must ensure that data access is logged, that deletion requests can be honored (a task complicated by immutable preprocessed shards), and that model checkpoints do not inadvertently memorize protected information. Storage design therefore has to preserve both throughput and governance metadata.

The transition from Tier 4 to Tier 5 is driven primarily by economics: data that is accessed rarely should move to archive storage when retrieval latency and governance requirements allow it, because the storage cost can be an order of magnitude lower.

4.3.7 Tier 5: Archive and cold storage

The final tier provides long-term preservation for compliance and auditability. Archive services (such as S3 Glacier) are designed for data that is rarely accessed: training logs from previous years, superseded model checkpoints, audit trails for regulatory compliance. At a representative \$0.004/GB/month, archive storage costs roughly \$4,800 per year for 100 TB, which is $5\times$ cheaper than the \$0.02/GB/month object-storage assumption used above. The trade-off is retrieval latency measured in minutes to hours, making this tier unsuitable for operational access.

The primary value of archive storage is organizational memory. When a model trained two years ago produces unexpected behavior in production, the ability to recover the exact training data, hyperparameters, and intermediate checkpoints for forensic analysis depends on archive storage. Organizations that aggressively purge old data to save costs discover, often at the worst possible moment, that they cannot reproduce or explain the behavior of deployed models.

Automated **lifecycle policies** govern the transition of data between tiers based on access recency. As one illustration of how such automation is expressed, a time-based rule for ML infrastructure might move checkpoints as follows: the most recent three checkpoints remain on the parallel file system for fast recovery; checkpoints older than 72 hours are transitioned to object storage; checkpoints older than 90 days are transitioned to archive storage; and checkpoints older than two years are deleted unless flagged for regulatory retention. S3 Lifecycle rules provide the same general transition/archive/delete mechanism for object lifetimes (Amazon Web Services 2026b). The running example later adopts a count-based variant of the same idea (section 4.7), where the transition rule keys on checkpoint count rather than age. Without automation, teams either hoard data on expensive tiers (wasting budget) or delete data too aggressively (losing reproducibility). The lifecycle policy encodes the organization's cost-recovery trade-off in a declarative rule that requires no manual intervention.

For our 175B model training run, the archive tier holds the complete training lineage: the raw 3 TB compressed source corpus, the 6 TB serialized shard set, all preprocessing scripts, the final checkpoint, and a sampled subset of intermediate checkpoints. If the organization budgets a conservative 100 TB for this retained lineage, the two-year archive cost is roughly \$9,600 (100 TB at \$0.004/GB/month $\times$ 24 months), well under one day of GPU time on the training cluster. The asymmetry between archive cost and compute cost makes retention inexpensive relative to the run

it documents; failures to archive usually reflect missing lifecycle ownership, unclear retention policy, or retrieval-process neglect rather than raw capacity cost.

Archive storage services offer multiple retrieval speed tiers, each with different pricing. In an S3 Glacier-style service, standard retrieval may complete within hours at moderate cost, expedited retrieval may complete within minutes at a higher cost, and bulk retrieval may trade still more latency for the lowest retrieval price (Amazon Web Services 2026d). The choice of retrieval tier depends on the urgency of the use case. Forensic analysis of a production incident warrants faster retrieval; annual compliance audits can use slower bulk retrieval. Designing the lifecycle policy requires understanding the storage cost, the expected retrieval frequency, and the urgency for each class of data.

Compliance requirements can require organizations to retain training data provenance records for deployed models. The EU AI Act, for instance, requires documentation for high-risk AI systems, including information about training data composition and preprocessing. Archive storage is usually the economically viable tier for retaining the complete lineage of a model, from raw data through preprocessing to the final checkpoint, for a required retention period. The cost of compliance storage is often small compared to the operational and legal cost of insufficient documentation.

4.3.8 Data format landscape

Data format choices determine whether the storage hierarchy delivers its theoretical bandwidth or collapses under metadata overhead. The same multi-terabyte dataset stored in different formats can yield I/O throughput that differs by over 100× on identical hardware. For large-scale machine learning, data must be structured for high-throughput sequential access, not for human readability or transactional convenience.

The useful design question is which access cost the workload must pay: per-file metadata, unnecessary feature reads, or sequential shard traversal. **Row-oriented formats**, such as CSV, JavaScript Object Notation (JSON) lines, or a directory of individual image files (JPEG, PNG), are the most intuitive and human-readable. They are easy to inspect and debug, making them suitable for initial data exploration or for datasets small enough to fit entirely in memory (typically under 10 GB). For training at scale, they are catastrophic for I/O performance. Each sample is a separate file, requiring a distinct file system operation (`open`, `stat`, `read`, `close`) that incurs significant per-sample overhead, often dominating the actual time spent reading data.

A second choice solves a different storage problem: **columnar formats** like Apache Parquet and Apache Arrow avoid unnecessary reads when tabular features can be selected before transfer. They organize data by column rather than by row, which allows for excellent compression as data within a column is typically of the same type and has lower entropy. This structure also enables **predicate pushdown**, where the storage engine reads only the specific columns (features) required by a query, avoiding unnecessary I/O. These formats are well-suited for tabular ML tasks and feature engineering but are less natural for unstructured data like images or audio, which are treated as atomic blobs.

A third choice addresses the training access pattern directly: **sequential streaming formats** reduce per-sample metadata to the cost of opening a shard. TFRecord, Mosaic's StreamingDataset format (Databricks 2026), and WebDataset (based on standard `tar` archives) group many samples into large, contiguous binary files called shards. This amortizes the cost of opening a file over thousands of samples. A data loader opens a single shard and streams its contents sequentially, maximizing bandwidth. The `tar`-based formats have the added advantage of being valid targets for HTTP byte-range requests, making them ideal for streaming data directly from object storage without downloading the entire file first.

A key trade-off within streaming formats is the presence of an **index**. TFRecord and HDF5 often use a separate index file to record the byte offset of each sample within a shard. This enables efficient random access within a shard but introduces a metadata dependency: the index must be read and held in memory. WebDataset is indexless: each record in the `tar` archive contains its own header specifying its size, allowing for purely sequential streaming without any external metadata. The trade-off is that indexless formats cannot efficiently seek to an arbitrary sample within a shard, a capability that matters for curriculum learning or active learning workloads that require nonsequential data access.

Compression integration also varies across formats. Parquet integrates compression at the column level, exploiting type homogeneity for high ratios. WebDataset and TFRecord typically leave compression to the individual sample: a WebDataset of images stores already-compressed JPEG files, while a text dataset might apply zstd compression per shard. This choice affects the CPU-I/O trade-off discussed in the context of NVMe reads, as per-sample decompression must be handled by the host CPU.

For our running example, the 1.5 trillion token dataset is stored as roughly 2,000 shards of approximately 3 GB each in a custom binary format. Each shard contains a small header (token count, vocabulary size, byte order) followed by a contiguous array of 4-byte token IDs. When staged on POSIX storage such as local NVMe or a parallel file system, the data loader memory-maps the shard and indexes directly into the token array by position, achieving near-peak read bandwidth with zero per-sample metadata overhead. When the same shards are read directly from object storage, the loader uses HTTP ranged GET streaming and an explicit local buffer rather than memory mapping. Table 4.4 compares the format choices by the access cost they impose on the storage path.

Table 4.4: Data Format Comparison for ML Training: Per-sample overhead determines whether the storage hardware's bandwidth is realized or wasted on metadata operations.

Format	Overhead/Sample	Random Access	Streaming	Compression
Individual files	~50 µs (open/stat/close)	Yes	Poor	Per-file
Parquet	~1 µs (row group seek)	Yes (row group)	Good	Column-level
TFRecord	~0.1 µs (index lookup)	With index	Excellent	Per-record
WebDataset (`tar`)	~0 (sequential)	No	Excellent	Per-sample
Raw binary (tokens)	0	Yes (byte offset)	Excellent	None needed

The comparison in table 4.4 sets the data-volume term in the pipeline equation, turning format selection into a bandwidth-sizing problem.

4.4 The Data Pipeline Equation

Every training job eventually raises a concrete sizing question: how much storage bandwidth must the pipeline deliver before accelerators begin to stall? The required rate depends on the training configuration, and getting it wrong in either direction is expensive. Under-provisioning storage bandwidth starves accelerators. Over-provisioning wastes budget on storage capacity that sits partially idle. The **data pipeline throughput equation** in equation 4.1 quantifies the bandwidth the pipeline must sustain so that accelerators never wait for data:

$$\text{BW}_{\text{required}} = N_{\text{GPU}} \times \eta_{\text{target}} \times \frac{D_{\text{vol,batch}}}{T_{\text{iteration}}} \tag{4.1}$$

where N_{GPU} is the number of accelerators, η_{target} is the target utilization (typically 0.8 to 0.95), $D_{\text{vol,batch}}$ is the size of one local batch in bytes, and $T_{\text{iteration}}$ is the time for one forward-backward pass.

The equation reveals four levers for controlling storage demand:

- **Accelerator count**: Reducing N_{GPU} directly reduces bandwidth requirements, but fewer accelerators also reduce training throughput.
- **Target utilization**: Lowering η_{target} reduces the bandwidth needed, but accepting lower utilization wastes expensive hardware.
- **Batch volume**: Decreasing $D_{\text{vol,batch}}$ reduces per-iteration data volume, but smaller batches may harm model convergence.
- **Iteration time**: Increasing $T_{\text{iteration}}$ gives storage more time to deliver each batch, but slower iterations extend total training time.

In practice, none of these levers is free; the pipeline must be engineered to deliver the bandwidth that the training configuration demands.

As a separate image-training example, consider 256 GPUs training with ImageNet-scale images, 200 ms iterations, and 80 percent target utilization. This configuration requires 39.3 GB/s of aggregate storage throughput. The bandwidth must be sustained continuously for the duration of training, which may last days or weeks. If at any point the storage system delivers less than 39.3 GB/s, accelerators begin to idle. For the chapter's 175B language-model running example, tokenized text has a much lower per-batch bandwidth demand, but the total data volume and checkpoint traffic over the run are much larger, shifting the bottleneck from peak training-data bandwidth to sustained throughput and checkpoint movement over weeks.

The consequences of falling short are captured by the **data stall ratio**, defined in equation 4.2 as the fraction of each training step where the accelerator waits for data:

$$\text{Data Stall \%} = \frac{T_{\text{step}} - T_{\text{compute}}}{T_{\text{step}}} \times 100 \tag{4.2}$$

where $T_{\text{step}} = T_{\text{compute}} + T_{\text{I/O}}$ when I/O and compute are not overlapped, or $T_{\text{step}} = \max(T_{\text{compute}}, T_{\text{I/O}}) = T_{\text{compute}} + \max(0, T_{\text{I/O}} - T_{\text{compute}})$ with pipelining.

Napkin Math 4.5: The data stall ratio

Problem: A GPU processes a batch in 200 ms, but storage takes 250 ms to deliver the next batch. What fraction of each training step is the accelerator idle, and how much does pipelining I/O with compute reduce that stall?

Without pipelining (sequential I/O then compute):

$$T_{\text{step}} = T_{\text{I/O}} + T_{\text{compute}} = 250 + 200 = 450 \text{ ms}$$

$$\text{Stall \%} = \frac{250}{450} = 55.6\%$$

With pipelining (I/O overlapped with compute):

$$T_{\text{step}} = \max(200, 250) = 250 \text{ ms}$$

$$\text{Stall \%} = \frac{250 - 200}{250} = \mathbf{20}\%$$

Systems insight: Even with pipelining, the accelerator is idle 20 percent of the time. To eliminate the stall entirely, the next batch must arrive within the 200 ms compute window, either by increasing effective storage bandwidth, hiding tail latency with deeper prefetch buffers, or using more parallel I/O streams.

The data stall ratio provides a diagnostic metric that storage engineers can use to identify whether a training job is compute bound or I/O bound. A stall ratio below 2 percent indicates that the storage pipeline is healthy: the accelerator spends virtually all its time computing. A stall ratio between 2 percent and 10 percent suggests that the pipeline is marginally adequate but will degrade if the training configuration changes (more GPUs, smaller batches, faster model). A stall ratio above 10 percent indicates a clear storage bottleneck that wastes significant compute budget. Section B.5.1 maps these same green, yellow, and red bands onto the broader fleet diagnosis, so a storage engineer can read a high stall ratio as a red-light signal that computation is sitting idle and place storage alongside the other axes that starve an accelerator. Profiling tools like PyTorch's DataLoader profiler and NVIDIA Nsight Systems can measure data stall ratio directly by comparing the time the accelerator spends waiting for data vs. computing.

A storage system cannot be a "fire-and-forget" component; it requires continuous monitoring to prevent silent bottlenecks from eroding training efficiency. Key metrics to track in real time include per-tier bandwidth utilization (NVMe, network, object storage), prefetch buffer depth (how many batches are queued and ready), the data stall ratio per GPU, the parallel file system's metadata operation rate, and NVMe drive health indicators (wear level, temperature, error counts). Alert

thresholds should trigger before stalls become visible in training loss curves. An undetected 5 percent increase in data stall ratio across a 1,000-GPU cluster for just one week wastes over $16,800 in idle compute (1,000 GPUs $\times$ $2/GPU-hour $\times$ 168 hours $\times$ 5 percent), making comprehensive storage monitoring a first-order economic concern rather than an operational nicety.

This high-level monitoring must be complemented by fine-grained profiling to pinpoint the exact source of a bottleneck. Profiling occurs at three levels. At the application level, NVIDIA Nsight Systems provides a timeline view that shows exactly when and for how long the accelerator is idle waiting for data, with microsecond precision. At the pipeline level, PyTorch's built-in `DataLoader` profiler reports per-worker throughput, batch processing times, and data queue depth, identifying slow workers or insufficient prefetching. At the device level, `iostat` and `nvme-cli` provide raw hardware bandwidth and latency metrics. For parallel file systems, Lustre's `lctl get_param` and GPFS's `mmpmon` give real-time statistics on storage server utilization, metadata operation rates, and client-side cache hit ratios. The most effective debugging approach combines all three levels: correlating an application-level accelerator stall with a specific pipeline worker and a saturated underlying storage device identifies the bottleneck tier and guides optimization effort to where it will have the greatest impact.

A common and costly failure mode is to develop and test a data pipeline on a small local dataset (1 GB) and only discover at full production scale (1 TB+) that the pipeline cannot sustain the required throughput. The root cause is often a bottleneck invisible at small scale but crippling under load: a Python Global Interpreter Lock (GIL) contention in a data augmentation function, a file descriptor leak that accumulates over millions of samples, or a memory fragmentation issue that causes the prefetch buffer to slow down after hours of continuous operation. Robust teams benchmark the data pipeline in isolation, measuring sustained throughput over at least 10 to 20 minutes at the target data volume, before connecting it to the training loop. This **pipeline stress test** catches system-level bottlenecks that would otherwise manifest as mysterious training slowdowns days into a production run.

Different ML workloads have dramatically different bandwidth profiles, even when using the same cluster. Image classification with ResNet-50 on ImageNet produces a high bandwidth demand because each batch contains hundreds of large images (150 KB each) and iteration times are short (100 to 200 ms). Language model pretraining produces a lower per-batch bandwidth demand because tokenized text is compact (each token is a 4-byte integer, and a batch of 4,096 tokens occupies only 16 KB per sequence), but the total data volume over the training run is enormous (trillions of tokens). Recommendation model training produces a mixed bandwidth demand: embedding lookups require high-IOPS random access to the embedding table, while the dense layers consume standard sequential training data. Each workload's bandwidth profile determines which storage tier is the bottleneck and where optimization effort should focus.

Figure 4.4 visualizes this relationship between storage bandwidth and GPU utilization. The S-curve reveals a sharp transition: below the stall threshold, even modest bandwidth shortfalls cause dramatic utilization drops, while above it, additional bandwidth yields diminishing returns. The plotted technologies show why no single tier suffices: a hard disk and a network file system sit deep in the stall zone, a single NVMe drive lands almost exactly at the knee where utilization is still rising steeply, and only an NVMe RAID array clears the threshold into the compute-saturated regime. This spread is why tiered storage architectures are essential, because no single tier provides both the capacity and bandwidth needed to keep a large cluster saturated.

The pipeline equation also reveals a scaling challenge that worsens with cluster size. As N_{GPU} increases, the required bandwidth $\text{BW}_{\text{required}}$ increases linearly, but the storage system's aggregate bandwidth does not automatically scale to match. Adding more compute nodes to a cluster that shares a parallel file system increases the demand on a fixed storage resource. Eventually, the storage system saturates and every additional GPU beyond the saturation point contributes zero additional training throughput while increasing cost. This saturation point is the practical upper limit on cluster scaling for a given storage configuration, and identifying it quantitatively requires applying equation 4.1 to the specific storage system's measured bandwidth.

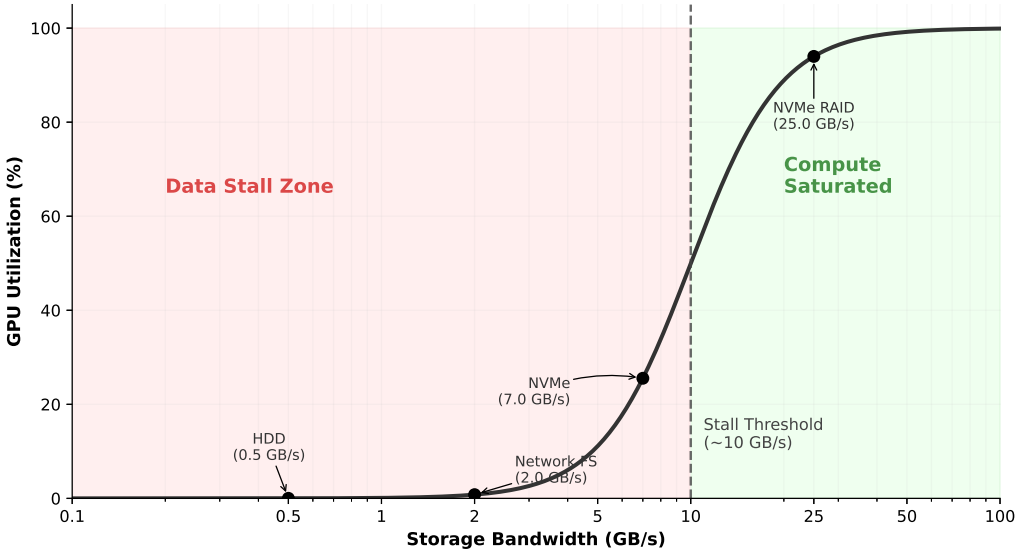


Figure 4.4: The Data Stall Frontier: GPU utilization follows a steep S-curve as a function of storage bandwidth. Around the stall threshold (~10 GB/s for a typical 8-GPU node), utilization crosses from waiting-dominated to compute-dominated. The position of common storage technologies on this curve explains why tiered architectures are essential.

Example 4.2: The metadata tax

Scenario: A large GPU training cluster has enough raw storage bandwidth on paper, but measured Model FLOPs Utilization (MFU) remains far below target.

Failure mode: Profiling reveals that GPUs are frequently idle even though network and storage bandwidth metrics look healthy. A common root cause is metadata amplification: the data loader issues a `stat()` system call for each file before opening it, multiplying metadata-server load across thousands of workers.

Consequence: Across thousands of workers, per-file metadata checks can generate millions of metadata requests per minute, overwhelming the parallel file system's metadata server. GPUs then stall on file-open operations, a delay invisible to standard I/O bandwidth counters.

Systems insight: Precompute a file manifest at job start and have loaders read it once, eliminating per-file metadata checks on the hot path. Storage bottlenecks are often metadata bottlenecks, not byte-throughput bottlenecks.

4.4.1 Pipelining and prefetching

The primary weapon against data stalls is **pipelining**: the CPU prepares batch $i+1$ while the GPU processes batch i. Figure 4.5 illustrates this overlap. When I/O time is less than compute time, pipelining hides the storage latency entirely, and the accelerator never stalls.

The key insight from figure 4.5 is that pipelining converts a sequential bottleneck into a parallel overlap, but only when the prefetch buffer is deep enough to absorb variance in I/O latency. Pipelining works perfectly when I/O times are consistent. In practice, they are not. Storage systems exhibit **I/O jitter**: variation in read latency caused by a multitude of factors. NVMe drives experience occasional latency spikes during internal garbage collection (when the controller reorganizes NAND flash blocks) or wear-leveling operations. Parallel file systems exhibit latency spikes when multiple training jobs contend for the same OSS nodes, when the MDS processes a burst of metadata operations, or when background scrubbing detects and repairs bit errors. Object storage latency can spike during cross-region replication, garbage collection of versioned objects, or load-balancer rebalancing.

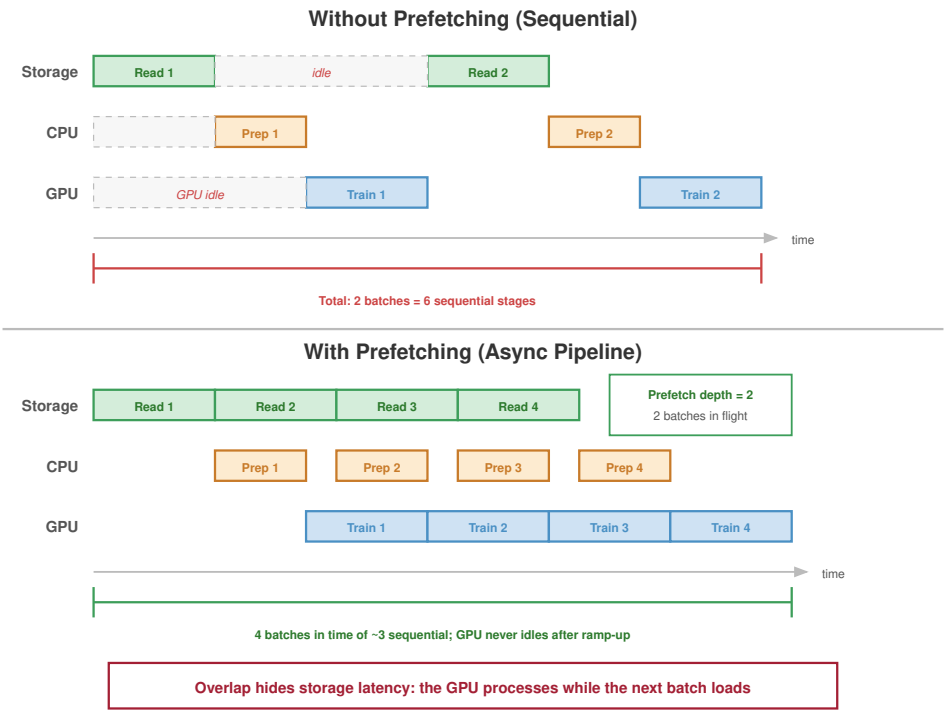


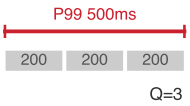
Figure 4.5: Pipelined Data Loading: The CPU prepares Batch $i+1$ while the GPU processes Batch i. Pipelining hides storage latency, but only if the prefetch buffer is deep enough to absorb variance.

A single slow I/O can create a "bubble" in the pipeline that propagates forward, stalling the accelerator. If the CPU was preparing batch $i+1$ and the read took twice as long as expected, batch $i+1$ is not ready when the GPU finishes batch i. The GPU idles until the read completes, and the pipeline falls one batch behind. If subsequent reads are also slow, the pipeline never recovers. The defense is a prefetch buffer: rather than preparing just one batch ahead, the CPU maintains a queue of Q_{prefetch} preloaded batches.

The minimum buffer depth Q_{prefetch} must absorb the worst-case I/O latency without the queue draining:

$$Q_{\text{prefetch,min}} = \left\lceil \frac{T_{\text{I/O,p99}}}{T_{\text{compute}}} \right\rceil \tag{4.3}$$

The prefetch depth equation in equation 4.3 tells us how many batches must be in flight simultaneously to hide I/O latency behind computation. The equation assumes that the long-run data supply rate is at least as high as the accelerator's consumption rate; prefetching absorbs latency variance, not a sustained bandwidth deficit. When average throughput is sufficient but P99 latency exceeds compute time, deeper prefetching prevents stalls. When sustained I/O throughput is lower than demand, the fix is faster storage, more parallel streams, smaller batches, or local staging. The equation uses P99 I/O latency rather than average latency because a single slow read can drain the buffer and stall the accelerator; sizing for the average guarantees frequent stalls at scale.



P99 I/O latency sets the required prefetch depth.

If I/O at the 99th percentile takes 500 ms and compute takes 200 ms, then $Q_{\text{prefetch,min}} = 3$ batches, with a safety margin of 5. In practice, data loaders like PyTorch's `DataLoader` use `prefetch_factor` and `num_workers` parameters to control this depth. Setting `prefetch_factor=2` with 4 workers creates a buffer of 8 batches, which is typically sufficient for NVMe-backed pipelines but may be inadequate for object-storage-backed pipelines where P99 latency can exceed 500 ms.

To illustrate the memory cost of prefetch buffer depth, consider a large-batch text pipeline where each GPU processes a packed token batch in roughly 200 ms and the collated batch occupies about 40 MB per GPU. Reading from local NVMe, the P99 I/O latency for that batch is approximately 50 ms. The minimum prefetch depth is $\lceil 50/200 \rceil = 1$ batch, and a safety margin of 2 is adequate. Reading from a parallel file system, the P99 I/O latency rises to roughly 200 ms due to network jitter and contention, requiring a minimum depth of $\lceil 200/200 \rceil = 1$, with a safety margin of 3 to account for occasional multi-hundred-millisecond outliers. Reading from object storage, the P99 latency can exceed 500 ms, requiring a depth of at least 3, with a safety margin of 5 or more. These numbers translate directly into host DRAM consumption: at 40 MB per batch, a depth-5 prefetch buffer per GPU consumes 200 MB, and 8 GPUs per node consume 1.6 GB. At 40 MB per batch with a depth of 1, the same node needs only 320 MB. The storage tier directly determines the memory cost of the prefetch buffer.

The cost of deep prefetching is memory: each buffered batch occupies host DRAM. A batch of 256 images at $224 \times 224 \times 3$ bytes (after decoding) occupies roughly 37 MB. A prefetch buffer of 8 such batches consumes 300 MB, which is negligible for a node with 512 GB of DRAM. For large-batch language model training where each batch contains millions of tokens, however, the prefetch buffer can grow to several gigabytes, competing with embedding tables and other DRAM consumers.

The interaction between prefetching and the storage hierarchy creates a layered defense against stalls. The first layer is the prefetch buffer in host DRAM, absorbing I/O variance from NVMe reads. The second layer is the local NVMe warm cache, absorbing network variance from the parallel file system. The third layer is the parallel file system read-ahead cache, absorbing variance from disk seeks across the OSS cluster. Each layer adds latency tolerance at the cost of memory or storage capacity. The system designer's task is to ensure that the combined depth of all layers exceeds the worst-case latency spike at the lowest tier in regular use. For our running example, if the parallel file system occasionally exhibits a 500 ms latency spike and compute takes 200 ms per batch, the NVMe warm cache eliminates this risk entirely by serving reads locally, so the prefetch buffer only needs to absorb the much smaller NVMe latency variance of microseconds rather than milliseconds.

Checkpoint 4.3: Data pipeline design

A training cluster runs 1,024 GPUs with 128-image batches (150 KB per image after compression, 150 ms per iteration).

- ☐ **Bandwidth requirement**: Using equation 4.1, calculate the required aggregate storage bandwidth at 90 percent target utilization.
- ☐ **Checkpoint contention**: Determine whether 800 GB/s aggregate parallel-file-system bandwidth is sufficient, then recalculate when checkpoint writes consume 20 percent of that bandwidth.
- ☐ **Prefetch depth**: Calculate the minimum prefetch buffer depth in batches needed to absorb 300 ms P99 object-storage latency.
- ☐ **Host memory**: Calculate total prefetch buffer memory when each decoded batch occupies 10 MB in host DRAM at the depth calculated in (3).

4.4.2 Multi-worker data loading

A single CPU core cannot keep a high-throughput accelerator fed, even when the storage hardware is fast enough, because of the CPU work required between the storage read and the GPU transfer. Consider the throughput required for image training: decoding a 150 KB JPEG image into a $224 \times 224 \times 3$ raw tensor requires entropy decoding of Huffman-coded coefficients, inverse discrete cosine transform (DCT), chroma upsampling, and pixel-format conversion. This process takes roughly 1 to 5 ms per image on a server CPU core. After decoding, augmentation adds further CPU work: a random crop requires computing crop coordinates and copying the sub-region; horizontal flip requires copying with reversed column order; color jitter requires per-pixel multiplication and addition. Each augmentation adds 0.5 to 2 ms per image.

At 1,000 images per second per GPU and 8 GPUs per node, a single core would need to decode and augment 8,000 images per second, a throughput $10\times$ to $40\times$ beyond what a single core can sustain. The solution is **multi-worker data loading**, where W worker processes each read from storage, decode, augment, and enqueue batches independently. The effective I/O throughput scales approximately linearly with W until one of three bottlenecks is reached: the storage device's bandwidth saturates, the PCIe bus between host and device saturates, or the workers exhaust host CPU cycles.

In PyTorch's `DataLoader`, each worker is a separate process with its own file descriptors and memory space. The `num_workers` parameter controls W. Setting W too low leaves the GPU starved; setting W too high wastes CPU resources on context switching and contention. A good heuristic is to start with $W = 4 \times G$, where G is the number of GPUs per node, and profile the data stall ratio, adjusting until stalls are below 2 percent. For a node with 8 GPUs, this yields 32 workers, which on a system with 64 CPU cores leaves 32 cores for the PyTorch runtime, GPU-to-GPU communication threads, and operating system overhead. The division of CPU resources between data loading and training orchestration is itself a capacity planning exercise.

The interaction between multi-worker loading and the storage hierarchy matters. When reading from local NVMe, workers can issue concurrent reads to the same RAID array without contention, because NVMe's deep command queues (up to 64K outstanding commands) handle parallelism in hardware. When reading from a parallel file system, workers distribute their reads across different OSS nodes, naturally aggregating bandwidth. When reading from object storage, workers issue concurrent GET requests, each to a different shard, achieving parallelism at the HTTP level.

A subtle pitfall in multi-worker loading is **shuffle quality vs. I/O efficiency**. Perfect shuffling requires that each batch contain samples drawn uniformly from the entire dataset, which implies random access across all shards. Random access, however, defeats the sequential I/O patterns that storage systems optimize for. The practical compromise is *shard-level shuffling* combined with *within-shard shuffling*: the loader shuffles the list of shards at epoch start, assigns contiguous groups of shards to each worker, and then shuffles samples within each shard's local buffer. This approach provides sufficient randomness for training convergence while maintaining sequential I/O at the storage level. Empirical studies confirm that shard-level shuffling produces training loss curves indistinguishable from full random shuffling for most workloads, as long as the shard size is large enough (typically 256 MB or more) to provide adequate within-shard diversity.

The shuffle buffer size creates a memory-randomness trade-off. A larger buffer provides better randomness (because samples are drawn from a larger pool) but consumes more host DRAM. For our running example with text data, a shuffle buffer of 10,000 sequences at 2,048 tokens each, with each token represented as a 4-byte integer, consumes roughly 80 MB. This is negligible relative to the 512 GB of host DRAM. For image data, where each decoded image occupies 150 KB, a shuffle buffer of 10,000 images consumes 1.5 GB, still manageable but a nontrivial fraction of the prefetch budget. The data loader designer must balance shuffle buffer size against prefetch depth, since both compete for the same host DRAM capacity.

The data loading pipeline, from storage read through decode, augmentation, and transfer to accelerator, represents the complete fuel delivery system for the training engine. When any stage becomes the bottleneck, the accelerator starves.

4.4.3 Data locality and placement

In a distributed training cluster, the placement of data shards across the storage hierarchy determines pipeline performance. Each training worker needs efficient access to its assigned portion of the dataset. The core trade-off is between the flexibility of shared storage and the raw speed of local storage. Reads from a node's local NVMe drives are typically 10–$100\times$ faster and have lower latency than reads that must traverse the network to a parallel file system or object store.

The simplest strategy is **static placement**. At the beginning of a training job, the orchestrator assigns a fixed subset of data shards to each node. The node stages this data by copying its shards from the shared PFS to local NVMe drives. For the remainder of the training run, all data reads are local, maximizing I/O bandwidth. This approach is highly effective for single-dataset training runs but introduces inflexibility: if the dataset changes or the job requires a different subset of data, the shards must be re-staged.

The alternative is **dynamic placement**, where shards are fetched on demand from shared storage as needed. This provides maximum flexibility, as any node can access any shard at any time. Dynamic placement is essential when the total dataset size exceeds the aggregate local NVMe storage of the cluster, or when the job involves data sampling strategies that change the active subset over time. The cost is performance: every read incurs network latency and consumes shared storage bandwidth.

A more advanced approach is **locality-aware scheduling**. The scheduler keeps track of which data shards are cached on which nodes' local NVMe drives. When it starts a new job or replaces a failed node, it prefers a node that already has the required data cached, rather than treating all nodes as interchangeable. This uses the principle of data gravity[12], co-locating compute with data to minimize transfer times. The storage point is simple: cached data only saves time if the placement system knows where the cache is. For teams running repeated experiments on the same dataset, locality-aware scheduling reduces staging time from minutes to zero.

For our 6 TB serialized dataset distributed across 256 nodes, each node is responsible for approximately 23.4 GB of data. If a node fails, its replacement must stage that shard subset from the PFS. At an uncontended PFS read speed of 4 GB/s, this staging takes about 5.9 seconds; under recovery contention, an effective 400 MB/s per-node rate stretches the same copy to about 58.6 seconds. If the orchestrator can instead schedule the replacement workload on an idle node that already has the data cached from a previous run, staging time is zero and training resumes instantly. At scale, this optimization compounds: in a 10,000-node cluster experiencing 10 node failures per day, locality-aware scheduling can save roughly a minute of staging time per failure, or about 10 minutes per day of aggregate cluster idle time, in the contended case.

12 **Data Gravity**: A metaphor treating large datasets as massive objects that attract compute. The gravitational analogy is apt: moving a 100 TB dataset across cloud regions costs over $9,000 in egress fees and takes hours, while launching compute next to the data is near-instantaneous. In ML fleet design, data gravity means the storage location of the training dataset often dictates the physical placement of the entire training cluster.

4.4.4 Data versioning and pipeline orchestration at scale

Knowing where data lives is only half the story; at scale we must also pin which version of the data each run uses and how it flows through the pipeline. Versioning turns the storage hierarchy into a reproducibility system: the fleet must know not only which bytes it read, but which preprocessing code, shuffle seed, schema, and integrity checks produced those bytes before the run began.



Lighthouse 4.1: Archetype B (DLRM at scale): feature store latency

While Archetype A (GPT-4) deals with static, versioned datasets of trillions of tokens, Archetype B (DLRM at Scale), the DLRM workload, deals with dynamic, high-velocity feature streams. For a recommendation system, the "ground truth" changes every second as users click and interact. This forces a move from simple file-based storage to a feature store architecture that must solve the point-in-time correctness problem: ensuring that the features retrieved for training exactly match what the model would have seen at the moment of inference, without "leaking" future information.

That same reproducibility requirement becomes harder when the inputs are live features rather than immutable text shards. Feature store architectures address point-in-time correctness[13] by maintaining an immutable feature ledger or a "time-travel" query engine, significantly increasing storage complexity.

At the 10,000-GPU scale, the dataset is no longer a *static collection of files* but a *dynamic stream* that must be captured with absolute precision. Foundation-model runs can last weeks or months; in our 30-day running example, the data seen near the end of the run must be identical in composition and order to the data seen on day 1. Otherwise, we cannot distinguish between a model regression and a silent change in the input distribution. Dataset versioning provides the necessary "Git for data" capability by snapshotting the cryptographic hashes[14] of every data shard, the exact preprocessing code used for tokenization, and the random seeds governing the global shuffle. This level of rigor is essential for audit trails and for debugging the subtle "loss spikes" that characterize large-scale training, where we must be able to replay the exact batch that caused a divergence to determine if the cause was a corrupted data sample or a numerical instability in the optimizer.

At petabyte scale, the data pipeline has to make every transformation reproducible. Raw documents are cleaned, tokenized, packed into training examples, and sharded across storage tiers. Each

13 **Point-in-Time Correctness**: In recommendation systems, features (for example, user click history) must be retrieved exactly as they existed at the moment of the historical interaction. Retrieving current features for a past event – a "temporal leakage" – inflates training accuracy but causes the model to fail in production. Solving this at petabyte scale requires an immutable feature ledger or a "time-travel" query engine, significantly increasing storage complexity.

14 **Hash Cost (SHA-256)**: Verifying the integrity of a 100 TB dataset using SHA-256 consumes approximately 1,501.5 CPU-core hours under a pipeline-level throughput assumption of 18.5 MB/s per core. For a 30-day foundation model training run, this metadata validation step must be pipelined or performed during ingestion to avoid stalling the fleet's startup phase.

stage writes an immutable artifact, and each artifact records the input version, code version, schema, and statistical checks it satisfied. Once those dependencies are explicit, the pipeline forms a Directed Acyclic Graph (DAG): later artifacts depend on earlier artifacts, and no stage is allowed to rewrite its own history. When a training job starts, the cluster manager verifies that each worker's local NVMe shards match the data snapshot declared for the run. By treating the data pipeline as a graph of versioned artifacts rather than a simple file transfer, the system eliminates the silent data drift that is a primary source of failure in foundation model training.

Beyond pipeline orchestration, a hardware optimization eliminates one of the most common bottlenecks: the CPU's role as intermediary between storage and accelerator.

4.5 GPUDirect Storage and the CPU Bypass

The storage hierarchy can provide enough aggregate bandwidth and still leave GPUs waiting when the CPU mediates thousands of small transfers. In the traditional path, every shard read bounces through host DRAM and software layers before reaching GPU HBM; GPUDirect Storage (GDS) removes that CPU-mediated path when local or RDMA-attached NVMe is the bottleneck. Understanding the traditional path first makes the GDS optimization clear.

The traditional data path for loading training data follows three hops: storage → host DRAM → GPU HBM. Data is first read from NVMe into a kernel buffer in host DRAM (via a DMA transfer initiated by the NVMe controller), then copied to a user-space buffer (the data loader's tensor, via a `memcpy` that the CPU executes), and finally transferred to GPU memory via the PCIe bus (using a `cudaMemcpy` or `cudaMemcpyAsync` call that programs the GPU's DMA engine). Each hop adds latency and consumes CPU resources. The CPU must orchestrate every transfer, manage buffer allocation, and handle interrupts from both the storage device and the GPU. For a single transfer, this overhead is negligible. When eight GPUs each demand thousands of small transfers per second, however, the aggregate CPU load for data movement alone can saturate multiple cores, leaving insufficient CPU capacity for the data augmentation that is also essential to the pipeline.[15]

[15] **GDS (GPUDirect Storage)**: Part of NVIDIA's GPUDirect family, which also includes GPUDirect RDMA (network adapters writing directly to GPU memory) and GPUDirect Peer-to-Peer (inter-GPU memory access). GDS extends this CPU-bypass philosophy to NVMe storage via the cuFile API. The combined effect of RDMA and GDS is that data can traverse the path remote storage → network → GPU memory without a single CPU instruction on the critical path, freeing the host processor entirely for augmentation and pipeline orchestration.

> **Definition 4.4: GPUDirect Storage**
>
> ***GPUDirect Storage (GDS)*** is a technology that enables a direct DMA path between NVMe storage devices and GPU memory, bypassing the host CPU and system DRAM.
>
> 1. **Significance**: It eliminates the "Bounce Buffer" through system memory. In the small-transfer model below, this reduces per-transfer latency by 75 percent and makes the local transfer path 4× faster. It allows the GPU to saturate the NVMe link speed (for example, 7 GB/s) while reducing CPU utilization for I/O.
> 2. **Distinction**: Unlike Traditional I/O, where every byte must be processed by the CPU and stored in kernel buffers, GDS provides Direct Memory Access between the storage controller and the accelerator.
> 3. **Common pitfall**: A frequent misconception is that GDS makes all storage "faster." In reality, it only accelerates Local or RDMA-attached NVMe; it does not eliminate the physical latency of network-attached file systems or object storage.

The software layers in the traditional I/O path contribute to the latency overhead. When a data loader calls `read()` on an NVMe-backed file, the call traverses the application's runtime, the Python/C++ boundary, the operating system's Virtual File System (VFS) layer, the file system driver (ext4, XFS), the block layer, and finally the NVMe driver. Each layer adds a few microseconds of overhead for parameter validation, lock acquisition, and buffer management. On the return path, the NVMe controller raises an interrupt, the interrupt handler wakes the blocked thread, and the data is copied from the kernel buffer to user space. The total round-trip overhead for a small read is 10–50 µs, dominated by the context switches and buffer copies rather than the actual NVMe access time.

GDS eliminates these software layers for the data path. The cuFile API registers a GPU memory region with the NVMe controller, establishing a direct DMA mapping. Subsequent reads transfer data from NVMe to GPU memory without traversing the kernel's file system or block layers. The CPU's

role is reduced to issuing the DMA command and checking for completion, a few microseconds of work compared to the 50+ microseconds of the traditional path.

The latency reduction from GDS matters most when the training pipeline is already optimized and the remaining bottleneck is the CPU's ability to mediate transfers. In a node with 8 GPUs, each running a data loader with 4 workers, the CPU must manage 32 concurrent I/O streams. At 120 µs per transfer, the CPU spends significant time in interrupt handling and buffer management. GDS offloads this work to hardware DMA engines, freeing CPU cores for data augmentation and other preprocessing tasks. Figure 4.6 contrasts the two data paths side by side, showing where GDS eliminates the CPU-mediated copies that dominate per-transfer overhead.

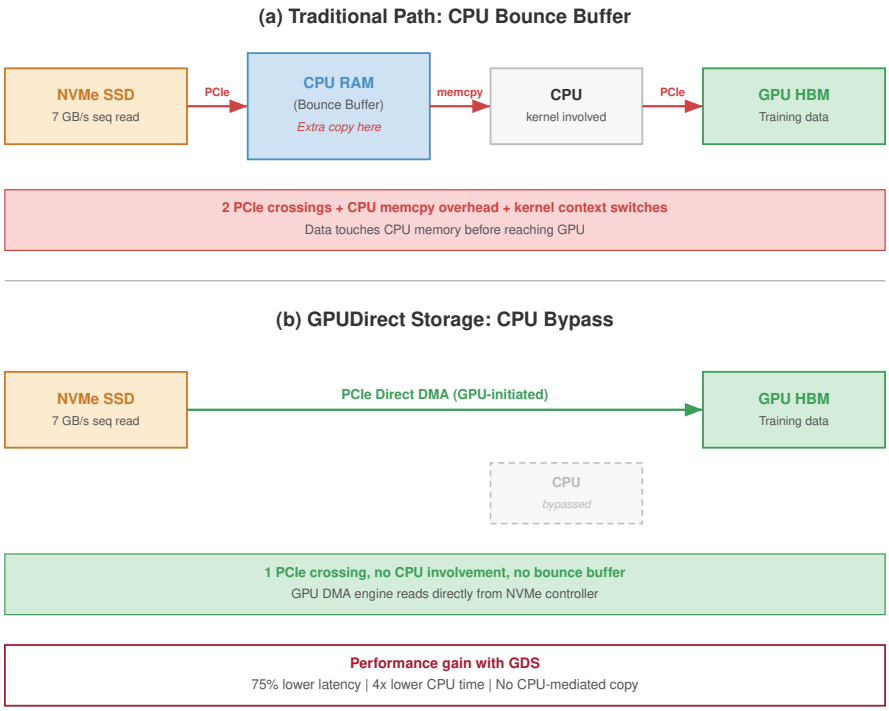


Figure 4.6: Traditional vs. GPUDirect Storage Path: The traditional path (top) requires two copies and CPU involvement. The GDS path (bottom) uses DMA to transfer data directly from NVMe to GPU memory, eliminating the CPU as a bottleneck.

As figure 4.6 illustrates, the throughput improvement from GDS is most pronounced for workloads that read many small objects (such as decoded image patches) where per-transfer overhead dominates. For large sequential reads (such as streaming a multi-gigabyte training shard), the throughput improvement is more modest because the transfer time dominates the setup overhead. The general principle is that GDS removes a constant overhead per transfer, so workloads with many transfers per second benefit most.

Napkin Math 4.6: The CPU bypass dividend

Problem: A training node with 8 GPUs loads 150 KB at 8,000 images/s per GPU (64,000 images/s total). Compare the CPU load under traditional I/O vs. GDS.

Traditional path: Each image requires a DMA from NVMe to DRAM, a host copy from kernel to user space, and a PCIe transfer to GPU. At 64,000 images/s with 120 µs of CPU time per

image, the CPU spends 7.68 seconds of CPU time per wall-clock second, consuming roughly 8 cores worth of processing just for data movement.

GDS path: Each image is DMA'd directly from NVMe to GPU. At 30 µs of CPU time per image (for initiating the DMA), the CPU spends 1.92 seconds of CPU time per wall-clock second, freeing roughly 6 cores for data augmentation.

Systems insight: GDS reduces latency and, more importantly, shifts CPU utilization from data copying (which adds no value) to data augmentation (which improves model quality).

GDS has practical limitations. Not all NVMe controllers support peer-to-peer PCIe transfers. The data must be in a format that the GPU can consume directly, which means it must already be decoded or GPU-decodable (raw pixel values, token IDs, or formats handled by GPU decoders, not arbitrary compressed JPEG or gzip streams). In practice, many pipelines use a hybrid approach: CPU-side decoding for compressed formats followed by a normal pinned-memory GPU transfer, while GDS is reserved for direct reads of GPU-consumable data from supported local or remote storage.

The architectural evolution from CPU-mediated I/O to direct storage access reflects a broader trend in ML systems: removing the CPU from the critical data path wherever possible. Section 3.4.1 described RDMA, which removes the CPU from network data transfers. GDS does the same for storage transfers. The combined effect of RDMA and GDS is that data can flow from remote storage, through the network fabric, and into GPU memory without a single CPU instruction on the critical path. The CPU is freed to perform the tasks where it adds unique value: data augmentation, pipeline orchestration, and error handling.

The GDS design principle also extends to checkpoint writes. In the traditional path, checkpoint data flows from GPU HBM through host DRAM (via PCIe) to NVMe (via CPU-mediated write). With GDS, the checkpoint can be written directly from GPU HBM to NVMe, eliminating the host DRAM copy. For a 1,750 GB checkpoint, this removes one full data copy and reduces the CPU's involvement in the write path, lowering T_{write} further.

4.5.1 The complete data path

Combining GDS with the multi-tier hierarchy, we can now trace the complete data path for our running example. At the beginning of the training job, the 6 TB serialized shard set, derived from the 3 TB compressed source corpus, is staged from object storage into the cluster's local NVMe caches, with each node receiving its assigned shard subset during setup. The data loader workers read tokenized shards from NVMe, assemble packed sequences in pinned host DRAM, and transfer those batches to the accelerator. With GDS enabled, token IDs that are already in a GPU-consumable layout can be loaded directly into GPU memory without CPU involvement; compressed formats still require CPU or GPU decompression before use.

During training, the pipeline operates in steady state. The CPU data loader workers continuously read from local NVMe, filling a prefetch queue in host DRAM. The GPU pulls batches from this queue via PCIe DMA. The compute phase (forward pass, backward pass, optimizer step) consumes the batch and updates the model weights in HBM. Every 10 minutes, the training framework initiates a checkpoint: model weights and optimizer state are written from GPU HBM to local NVMe (either through host DRAM or via GDS), and a background thread asynchronously copies the local checkpoint to the parallel file system.

At the end of the training job, the final model checkpoint is promoted from the parallel file system to object storage for long-term retention. The local NVMe copies are deleted during job cleanup. If the model is deployed for inference, it is loaded from object storage into the serving cluster's HBM, completing the lifecycle of data movement through the hierarchy.

The total number of data copies in this lifecycle is instructive. Each training sample traverses: object storage → NVMe (staging), NVMe → host DRAM (read), host DRAM → GPU HBM (transfer). Each checkpoint traverses: GPU HBM → NVMe (local save), NVMe → parallel file system (async copy), parallel file system → object storage (long-term retention). Every copy consumes bandwidth and contributes latency. The engineering goal is to minimize copies on the critical path (the training loop) and tolerate additional copies on noncritical paths (staging and archival).

The volume of data moved during a 30-day training run reveals a counterintuitive reality. A single staging copy of the 6 TB serialized dataset from object storage to local NVMe accounts for a modest transfer. The checkpointing process, however, generates a vastly larger data stream. With a roughly 1,750 GB checkpoint created every 10 minutes across 256 nodes for 30 days, the system produces approximately 4,320 checkpoints, totaling roughly 7.6 PB of state that must traverse the storage hierarchy. Checkpoint data movement dwarfs training data movement for large models: while the 6 TB training dataset might be read a handful of times (once per epoch), the 7.6 PB of checkpoint data is generated anew, making checkpoint I/O the dominant storage workload. This insight explains why checkpoint staging strategy (write locally, replicate asynchronously) can have a larger impact on overall storage design than training data pipeline optimization for large-model runs. The physical data path dictates where bytes move; the remaining question is what each movement costs.

4.6 Storage Economics

A 100 TB training dataset can reside on any tier of the hierarchy, but the cost differs by orders of magnitude depending on which tier is chosen and how the data is accessed. ML storage economics is not simply about choosing the cheapest tier; it is about minimizing **total cost of data delivery**, which includes storage costs, transfer costs, and the opportunity cost of idle accelerators.

Consider storing a 100 TB training dataset. The annual storage cost varies enormously by tier:

- **Object storage (S3 Standard)**: $24,000/year
- **Local NVMe (provisioned)**: $120,000/year
- **Archive (Glacier)**: $4,800/year
- **HBM equivalent** (hypothetically storing 100 TB in GPU memory at $15/GB): $1,500,000 in hardware amortization

The 5× cost difference between local NVMe and object storage (with the full HBM-to-archive span exceeding 3,000×) explains why the hierarchy exists: data must live at the cheapest tier possible, migrating upward only when needed and returning downward when done. This cost gradient reflects the underlying physics. Faster storage requires more expensive materials (HBM uses 3D-stacked silicon with through-silicon vias), more energy per bit accessed, and more physical proximity to the accelerator (which limits the amount that can be provisioned per node). Cheaper storage uses commodity components (standard hard drives for archive), consumes minimal energy when idle, and can be located anywhere with network connectivity.

Storage cost, however, is only part of the equation. **Data transfer costs** can dominate, especially in cloud environments. Under the representative cross-boundary egress assumption used here, reading the 100 TB dataset from object storage to compute instances incurs a transfer charge of $9,000. For multi-epoch training that reads the dataset 10 times, the egress cost alone exceeds $90,000, more than the annual storage cost. This inversion, where *reading* data costs more than *storing* it, drives the architecture decision to cache data on local NVMe rather than streaming from object storage each epoch.



Repeated egress, not storage, dominates cloud cost.

The economic analysis changes further when accounting for the opportunity cost of accelerator idle time. A 1,000 GPUs cluster at $2/GPU-hour costs $48,000 per day. If an undersized storage system reduces accelerator utilization from 90 percent to 70 percent, the organization loses $9,600 per day in wasted compute. Over a 30 days training run, this amounts to $288,000 of lost compute value, an amount that would easily fund a parallel file system upgrade or additional NVMe capacity. The economically rational approach treats storage investment not as an expense but as an enabler of compute utilization: every dollar spent improving storage throughput returns several dollars of increased accelerator productivity.

> Systems Perspective 4.2: The storage cost iceberg
>
> The visible cost of storage ($/GB/month) is the tip of the iceberg. Below the surface lie costs that often exceed the storage cost itself:
>
> - **Egress fees**: A representative cross-boundary egress charge of $0.09/GB makes a 100 TB dataset read once per epoch across 10 epochs cost $90,000 in egress alone.

- **IOPS charges**: Object storage charges per-request fees. S3 Standard GET requests cost about $0.0004 per 1,000 requests, so a dataset of 100M individual files, read once, costs about $40 in GET request fees before data transfer or retrieval charges, even though the storage cost is $2,300/month.
- **Idle accelerator cost**: If storage stalls reduce GPU utilization from 90 percent to 70 percent on a 1,000 GPUs cluster at $2/GPU-hour, the lost compute costs $9,600/day (20 percent of $48,000 daily spend), dwarfing any storage savings.
- **Retrieval fees**: Archive storage charges retrieval fees ($0.02/GB for Glacier) in addition to egress. Restoring a 100 TB dataset from Glacier costs $2,000 in retrieval fees plus $9,000 in egress.

The economically rational strategy often invests more in storage (local NVMe caching, parallel file system capacity) to reduce transfer costs and prevent accelerator stalls.

The cost iceberg reveals that raw storage capacity can account for less than half of the total expenditure; replication, metadata operations, API calls, and cross-region transfers can collectively dominate the bill at scale. Among those hidden line items, egress is the one that often flips the architecture decision. Unlike capacity charges, which scale with the volume of data stored, egress charges scale with the volume of data read, and ML training reads the full dataset on every epoch.

Napkin Math 4.7: The egress tax

Problem: A team trains a vision model on a 50 TB image dataset stored in S3. Training runs for 20 epochs. The decision is whether to stream from S3 each epoch or stage to local NVMe.

Option A: Stream from S3

- Storage: 50 TB $\times$ $0.023/GB/month $\times$ 12 months = $13,800/year
- Egress: 50 TB $\times$ 20 epochs $\times$ $0.09/GB = $90,000 per training run
- Total for 4 training runs/year: $13,800 + $360,000 = $373,800/year

Option B: Stage to local NVMe

- Storage (S3 source of truth): $13,800/year
- NVMe capacity (50 TB across cluster): 50 TB $\times$ $0.10/GB/month $\times$ 12 = $60,000/year
- Egress (stage once per training run): 50 TB $\times$ 4 runs $\times$ $0.09/GB = $18,000/year
- Total: $13,800 + $60,000 + $18,000 = $91,800/year

Systems insight: Local NVMe caching saves $282,000/year at 20 epochs per run, enough to fund 9 additional H100 GPUs at list price. Including the $60,000/year provisioned NVMe cost and the once-per-run staging egress, the break-even point is just over 4 epochs per run; by the fifth epoch, local caching is cheaper than streaming from S3 every epoch.

The notebook turns the iceberg into a decision rule: the cheapest storage tier is not always the cheapest delivery path. Object storage can remain the source of truth, but repeated reads make egress, request overhead, and idle-accelerator cost part of the storage design. Multi-epoch vision training may justify staging on local NVMe by the fifth epoch, while single-epoch language-model pretraining may leave the corpus in object storage and optimize the streaming path instead. The architecture is therefore chosen from access pattern, read count, and latency sensitivity rather than from $/GB/month alone.

The cost analysis extends to the parallel file system tier, which occupies a middle ground between NVMe and object storage in both performance and price. A parallel file system capable of delivering 1 TB/s aggregate bandwidth requires roughly 100 OSS nodes, each with multiple NVMe drives and high-bandwidth network connections. The capital and operational cost of such a system can exceed $10 million per year. This expense is justified only if the alternative, streaming from object storage and paying egress fees, is even more expensive, or if the latency requirements of checkpoint

writes cannot be met by object storage alone. The break-even analysis depends on the organization's workload mix: a team running a single long training job may find that staging data on local NVMe and bypassing the parallel file system entirely is the cheapest option, while a team running many concurrent short jobs benefits from the shared namespace that a parallel file system provides.

The decision to **build vs. buy** storage infrastructure hinges on the trade-off between the high capital expenditure of an on-premises parallel file system and the recurring operational expenditure of cloud object storage. A 1-petabyte Lustre deployment costs roughly $3–5 million in hardware, plus an additional $500,000 per year in operational overhead for power, cooling, and engineering support. Storing 1 PB in a cloud object store at $0.02/GB/month costs $240,000 per year for storage alone. However, egress fees for accessing the data dominate the total cost: reading that petabyte at 10 reads/year adds $900,000 in data transfer fees. Under these assumptions, annual cloud delivery costs $1,140,000; after subtracting $500,000 of on-premises operations, the net savings are $640,000 per year. The resulting hardware break-even is roughly 4.7–7.8 years, not 18 to 24 months. Reaching an 18- to 24-month payback requires a hotter workload with many more full-dataset reads per year, lower capital expense, or non-egress drivers such as checkpoint latency and shared-namespace requirements. For our running example, the 6 TB serialized corpus is much smaller than 1 PB, so the local-storage case is driven more by latency, repeated experiments, and checkpoint staging than by one-year egress savings alone.

Hardware degradation introduces the need for a **storage refresh cycle**. NVMe drives are rated for a specific write endurance, measured in Drive Writes Per Day (DWPD) over a typical five-year warranty period. Many large-scale ML training workloads are predominantly read-heavy once the initial dataset is ingested, so drives in a training cluster can last beyond their rated write lifespan. The primary driver for a refresh cycle may therefore be density rather than wear: newer drives often provide more capacity at a similar price point. Refreshing a cluster from 7.68 TB to 15.36 TB NVMe drives doubles the local cache capacity, enabling larger datasets to be staged directly on compute nodes and reducing dependence on the parallel file system for steady-state reads.

4.6.1 Tiering strategies

The cost structure described earlier turns tiering into a placement policy: keep data only as close to the accelerator as its access frequency and latency requirement justify. A **tiering strategy** encodes that policy across the hierarchy.

The fastest and most expensive layer is the **hot tier** (local NVMe + host DRAM), which holds data being actively consumed by the running training job. Datasets are staged from shared storage to local NVMe at job start. This tier is provisioned per-node and is not shared across the cluster, so data that multiple jobs or teams need simultaneously cannot live here alone.

That shared-access requirement is what justifies the **warm tier** (parallel file system), which holds datasets accessed by multiple jobs or teams, shared checkpoints for fault recovery, and model artifacts under active development. The parallel file system provides the shared namespace necessary for multi-tenant access with strong consistency.

Below the warm tier sits the **cold tier** (object storage), which serves as the canonical repository for all organizational training data. Datasets are authored and versioned in object storage, then promoted to warmer tiers when needed for training. Object storage also serves as the durable backup for checkpoints after they are staged from local NVMe through the parallel file system.

Data that must be retained for compliance or reproducibility but is not expected to be accessed during normal operations drops to the **archive tier** (Glacier or equivalent). Lifecycle policies automatically transition data from cold to archive after a configurable period, typically 90 to 365 days since last access.

The movement of data between tiers should be automated through lifecycle policies. A well-designed tiering system automatically promotes data from cold to warm when a training job requests it, and demotes data from warm to cold when no job has accessed it for a configurable period. Manual tiering introduces operational burden and inevitably leads to either over-provisioning (data left on expensive tiers) or under-provisioning (data unavailable when needed).

Return to our running example to see tiering in action. The 3 TB source corpus and 6 TB serialized training shards live permanently in object storage (cold tier). When a training job is scheduled, the orchestration system triggers a "data staging" phase that copies the tokenized shards to the parallel

file system (warm tier) before the first training node boots. Each node's data loader then copies its assigned shards to local NVMe (hot tier) during the first epoch. Subsequent epochs read entirely from local NVMe, incurring no network traffic. When the job completes, a cleanup process deletes the local NVMe copies and, after a configurable grace period, purges the parallel file system copy. The entire lifecycle, from staging through training to cleanup, is managed by policy rather than by the training engineer.

The economic benefit of automated tiering compounds over time. An organization running 50 concurrent training jobs, each using 10 TB of data, would need 500 TB of parallel file system capacity if all data were left in place. Automated demotion after job completion might reduce the steady-state parallel file system usage to 100 TB, saving $144,000 per year at the parallel file system's per-TB rate. These savings are invisible in any single experiment but substantial when accumulated across the fleet.

Data staging patterns

The staging decision determines where the job pays for movement: before training starts, during the first epoch, or on every epoch. Three common patterns address different trade-offs between startup latency and steady-state throughput:

- **Prestaging**: Copy the entire dataset from object storage to local NVMe before the first training iteration begins. This gives the best steady-state performance because all reads are local, but it creates the worst job start time; staging a 10 TB dataset across 256 nodes at 500 MB/s per node takes roughly 80 seconds per node, and shared parallel file system bandwidth can extend that delay to several minutes.
- **On-demand staging**: Copy shards to local NVMe only when the data loader first accesses them. This removes the startup delay but makes the first epoch pay the full network-read cost, which works best for multi-epoch training where later local reads amortize the initial staging cost.
- **Streaming**: Keep data off local storage and read from the parallel file system or object storage every epoch. This has zero startup delay and zero local storage requirement, but every epoch pays the full network-read cost, making it appropriate for single-epoch pretraining or datasets too large for local NVMe.

Tiering is a placement policy, so the next question is where the tier boundaries can move. Emerging devices shift the boundary between DRAM, NVMe, and archival storage, while inference workloads reuse the same hierarchy for a different objective: reducing cold-start latency and managing serving-time working sets rather than maximizing epoch throughput.

4.6.2 Storage technologies that move tiers

While the six-tier hierarchy represents a common production baseline, memory and storage technologies can fill the bandwidth gaps between existing tiers, reshaping effective storage architecture for large-scale ML. The widest unfilled gap in the hierarchy sits between host DRAM and local NVMe. Compute Express Link (CXL), an open standard interconnect that allows CPUs, accelerators, and memory devices to share memory with cache-coherent semantics, targets exactly this gap. CXL-attached memory provides bandwidth between that of local DRAM and NVMe (roughly 30–60 GB/s) at latencies that are also intermediate (200–500 ns). For ML workloads, CXL memory could serve as a "Tier 1.5" between host DRAM and local NVMe, creating a massive capacity pool for embedding tables and prefetch buffers without the latency penalty of NVMe. For our running example, CXL-attached memory could expand the effective host memory tier from 512 GB to over 4 TB per node, large enough to hold the compressed source corpus or a large local shard cache, though still short of the full 6 TB serialized training copy on one node.

A different bottleneck appears when CPU-side decoding, not storage bandwidth, throttles the pipeline. The **Computational Storage Drive (CSD)** closes that gap by embedding processing elements (FPGAs or simple CPU cores) directly on the SSD controller, enabling decompression, filtering, or format conversion to happen at the storage device rather than consuming host CPU cycles. For ML pipelines where CPU-side decoding is the bottleneck, computational storage could eliminate the CPU from the data path entirely, complementing GDS by offloading work that direct DMA cannot address.

Checkpoint writes remain throttled by SSD write latency, the T_{write} term the Young-Daly formula penalizes. **Persistent memory**, despite market shifts such as the discontinuation of Intel Optane, remains the architectural idea that attacks that term directly. Byte-addressable non-volatile memory that sits between DRAM and NVMe in both latency and capacity could transform checkpoint storage: the durability of an SSD with write latencies approaching DRAM would reduce T_{write} from milliseconds to microseconds, making frequent, low-overhead checkpointing more practical.

These technologies are not yet broadly deployed at the scale of the largest ML clusters, but they illustrate the direction of the storage hierarchy's evolution. The fundamental principle remains unchanged: faster, more expensive storage closer to the accelerator, with slower, cheaper storage at the periphery. What changes is the *granularity* of the tiers and the *size of the gaps* between them.

4.6.3 Storage for inference workloads

The storage requirements for inference differ fundamentally from training. Training reads datasets continuously and writes checkpoints in bursts. Inference reads model weights once at startup and performs no further storage I/O during normal operation. For inference serving, the dataset is replaced by a stream of incoming user requests, and the primary storage challenge shifts from sustained throughput for large datasets to latency for loading the model itself.

The most critical metric for inference storage is **model loading latency**, often called cold-start time: the duration required to load a model from storage into the accelerator's HBM. For our 175B parameter language model, the weights alone occupy 350 GB in FP16 format. Loading this sequentially from a single high-performance NVMe drive at 14 GB/s takes 25 seconds, an unacceptable delay for a user-facing application. Loading from a shared PFS at a per-node rate of 4 GB/s takes nearly 88 seconds. From object storage at 1 GB/s, the delay approaches six minutes.

To reduce cold-start time, the model is sharded and loaded in parallel. If the 350 GB model is striped across 8 NVMe drives, each loading a 43.75 GB shard at 7 GB/s, the total load time drops to roughly 6 seconds. When using tensor parallelism across 8 GPUs, each GPU loads only its 43.75 GB shard, which can be accomplished in under 4 seconds by reading from host DRAM. For the most latency-sensitive applications, organizations maintain **warm replicas**: one or more copies of the model weights kept preloaded in host DRAM, ready to be transferred to HBM in under two seconds. This trades DRAM capacity for near-instantaneous cold-start times.

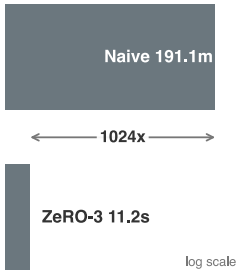
During autoregressive generation, the server keeps a per-request record of the attention keys and values produced by earlier tokens so it can generate the next token without recomputing the entire prefix. This record, called the **KV cache**, grows with sequence length and can consume significant HBM. For long sequences (128K tokens or more), KV cache offloading to host DRAM or NVMe extends the effective context window at the cost of increased latency per generated token, because cache blocks must be moved back into HBM before attention can use them. The storage lesson is that inference has its own working set, not just a model-loading problem.

While model serving dictates its own economic calculus, the training phase introduces an entirely different, burst-heavy I/O challenge: the preservation of model state through checkpoint storage.

4.7 Checkpoint Storage

After a thousand GPUs have been training for six hours, a power supply failure turns checkpoint storage into the difference between resuming quickly and repeating hours of work. The decisive storage questions are how quickly the most recent checkpoint was saved, where it resides, and whether it can be read back. Checkpoints are the most demanding write workload in the storage hierarchy. They are also among the most consequential: a lost checkpoint after a hardware failure means repeating hours or days of training. The storage architecture must therefore minimize T_{write}, the time the training pipeline pauses to save a checkpoint.

A 175B parameter model with Adam optimizer generates checkpoints of approximately 1,750 GB. The checkpoint includes 350 GB of FP16 model weights, 1,400 GB of FP32 optimizer state for momentum and variance, learning rate scheduler state, random number generator state, and the current data loader position. Every GPU in the cluster saves its shard of the checkpoint simultaneously, creating a checkpoint storm that the storage system must absorb without disrupting ongoing training reads.



Sharding turns replicated checkpoint storms into sharded writes.

Definition 4.5: Checkpoint storm

Checkpoint Storm is a burst of synchronized network and storage traffic that occurs when all nodes in an ML training fleet save model state simultaneously.

1. **Significance**: The storm magnitude scales as $T_{\text{write}} = R \times \text{per-replica state}/\text{BW}_{\text{fabric}}$, where R is the number of replicas writing at once. For a 70B-parameter model, each replica holds 140 GB FP16 weights, 140 GB gradients, and 840 GB optimizer state. With vanilla data parallelism across 1,024 replicas, a naive checkpoint of that full per-replica state generates 1146.9 TB of simultaneous writes. At 100 GB/s fabric bandwidth, that write takes roughly 11,468.8 seconds, or over 191.1 minutes of training stall per checkpoint event. ZeRO-3, an optimizer-state-sharding scheme in which each node stores only its $1/N$ slice of the state rather than a full replica, collapses the written state to roughly 1.12 TB total (140 GB weights + 140 GB gradients + 840 GB optimizer state, split across 1,024 replicas), bringing T_{write} down to roughly 11.2 seconds. This stall is the checkpoint T_{write} term and can dwarf the compute time between checkpoints when the sharding optimization is omitted.
2. **Distinction**: Unlike general I/O contention (which is stochastic and unpredictable), a Checkpoint Storm is synchronous and periodic: every node writes at the same moment because the training orchestrator triggers checkpoint after a fixed step count. Its predictability makes it both more damaging (all nodes compete simultaneously) and more tractable (it can be prevented by design through staggered scheduling or asynchronous serialization).
3. **Common pitfall**: A frequent misconception is that checkpointing every 100 steps is "low overhead." At 70B scale, this assumption is catastrophically wrong: if each training step takes 10 seconds and a checkpoint storm takes 11,468.8 seconds, checkpointing every 100 steps means spending most of the run on checkpoint I/O rather than useful training.

The severity of a checkpoint storm depends on two factors: the per-node shard size, determined by the model size and the sharding strategy, and the throughput of the first write tier, typically local NVMe. A concrete 175B example shows why the local-write phase, rather than the durable copy to the parallel file system, governs the exposed training pause.

Napkin Math 4.8: Tiered checkpoint staging

Problem: A 256-node cluster saves a 175B-parameter checkpoint every 10 minutes. Each checkpoint totals 1,750 GB. With ZeRO-3, each node saves roughly 6.8 GB.

1. **Per-node write to local NVMe** (4 drives at 7 GB/s each = 28 GB/s): 6.8 GB $\div$ 28 GB/s $\approx$ 0.24 seconds.
2. **Async copy to PFS**: 256 $\times$ 6.8 GB $\approx$ 1.8 TB total. If the PFS provides 1 TB/s aggregate, the storm completes in roughly 1.8 seconds.
3. **Per-node PFS bandwidth**: If all 256 write simultaneously, each gets 1000 GB/s $\div$ 256 = 3.9 GB/s.
4. **Training pause**: only about 0.24 seconds (the local NVMe write). The PFS copy overlaps with the next training iteration.
5. **Overhead**: 0.24 seconds pause every 600 seconds $\approx$ 0.04 percent training time lost to checkpointing.

Systems insight: Under these 1 TB/s aggregate-PFS assumptions, tiered staging reduces checkpoint overhead from about a 1.8 seconds PFS-direct write to a sub-second local write. The key insight is that T_{write} for the training pipeline is the *local* write time, not the *durable* write time.

The **tiered staging** strategy minimizes T_{write} by writing in two phases. In the first phase, each node writes its checkpoint shard to local NVMe at full bandwidth. With 4 NVMe drives providing 28 GB/s aggregate, a per-node shard of roughly 6.8 GB (a 256-node ZeRO-3 split of the 1,750 GB total checkpoint) completes in roughly 0.24 seconds. The training pipeline can resume immediately after the local write completes.

In the second phase, a background process asynchronously copies the local checkpoint to the parallel file system for durability. This copy can overlap with the next training iteration, so it does not block the pipeline. The risk is that if the node fails before the async copy completes, the local checkpoint is lost. The mitigation is to replicate checkpoints to at least two peer nodes' NVMe drives before declaring the save complete, providing durability even if one node fails.

The total storage consumed by checkpoints over the life of a training run is substantial. A 175B parameter model checkpointed every 10 minutes over a 30-day training run generates roughly 4,320 checkpoints, each 1,750 GB, for a total of approximately 7.6 PB of checkpoint data. Retaining all of them is neither necessary nor economical. The retention policy this run adopts keeps the three most recent checkpoints on the parallel file system for fast recovery, copies every 100th checkpoint to object storage for long-term auditability, and deletes the rest. This policy reduces the parallel file system checkpoint footprint from 7.6 PB to roughly 5.25 TB (three live full checkpoints) while preserving 43 historical snapshots in object storage for posttraining analysis.

Incremental checkpointing offers a further optimization for reducing T_{write}. Rather than saving the entire model state every checkpoint, an incremental checkpoint saves only the parameters that changed since the last full checkpoint. For models where a large fraction of parameters are frozen (such as fine-tuning scenarios where only the last few layers are updated), incremental checkpoints can be orders of magnitude smaller than full checkpoints. The trade-off is recovery complexity: restoring from an incremental checkpoint requires applying a chain of incremental updates to a base checkpoint, which extends recovery time. Most production systems use a hybrid approach, saving incremental checkpoints frequently and full checkpoints periodically (every 10th to 100th increment).

Checkpoint storage interacts directly with the system's failure model. From the storage perspective, the design goal is to minimize T_{write} (the time the training pipeline pauses) while ensuring that at least one durable copy of the checkpoint exists before the next failure window opens. Reducing T_{write} through tiered staging makes more frequent checkpoints affordable. Minimizing T_{write}, however, only matters if the checkpoint can actually be read back; the costlier failure is a checkpoint that was written but never verified as restorable.

War Story 4.1: The backup that had not been restored

Context: On January 31, 2017, around 23:30 UTC, a GitLab.com site reliability engineer working a database replication incident wiped what they believed was the secondary PostgreSQL data directory but had switched terminals to the production primary (GitLab 2017). The restored snapshot later returned the database to its state at about 17:20 UTC, which is why roughly six hours of writes were unrecoverable.

Failure mode: GitLab.com depended on five layered recovery paths: pg_dump backups to S3, LVM snapshots, Azure disk snapshots, PostgreSQL replication, and WAL archiving. All five failed or were absent. The S3 dumps were silently empty due to a PostgreSQL 9.2/9.6 version mismatch; Azure snapshots were not enabled for database servers; replication was the very procedure that had broken; LVM snapshots existed but were six hours stale; WAL archiving had never been implemented.

Consequence: Roughly six hours of recent commits and database state were unrecoverable. GitLab livestreamed the recovery on YouTube to about five thousand concurrent viewers and published a public Google document tracking progress, restoring service after about eighteen hours from a stale staging-environment snapshot.

Systems lesson: Storage reliability is a restore property, not a backup property. A checkpoint, replica, or dump only counts if the team regularly proves that it can be located, validated, and restored under incident conditions.

That incident turns checkpoint storage from a capacity question into a restore-design question.

Checkpoint 4.4: Checkpoint storage design

A training cluster of 128 nodes saves a 175B-parameter model checkpoint every 10 minutes. Each checkpoint is 1,750 GB total, distributed across all nodes.

- ☐ **Per-node state**: Calculate the per-node checkpoint size.
- ☐ **Local write**: Calculate T_{write} for the local write when each node writes to local NVMe at 28 GB/s.
- ☐ **Async copy**: Calculate the per-node write bandwidth and async-copy time when the parallel file system provides 1 TB/s aggregate and all 128 nodes write simultaneously.
- ☐ **Tiered staging**: Explain why tiered staging (local NVMe first, then async PFS copy) is faster than writing directly to PFS.

4.7.1 Distributed checkpoint coordination

In a sharded training setup with 128 nodes, no single node necessarily holds the complete training state. One node may hold a shard of the optimizer state, while other model-partitioning strategies split the weights themselves across devices. A complete checkpoint therefore requires every node to save its shard, and the checkpoint is only complete when all shards have been durably written. This creates a coordination problem: the system must confirm that all 128 nodes have finished writing before training can resume.

The simplest approach is a synchronous barrier: all nodes pause training, write their shards to local NVMe, then run a cluster-wide acknowledgement operation to confirm completion. Only when every node has acknowledged its write does training resume. This approach minimizes the risk of inconsistent checkpoints, where some shards are from step i and others from step $i+1$, but it maximizes the training pause because the slowest node determines the barrier time.

Asynchronous checkpointing reduces the training pause by writing in the background. Each node snapshots its shard to a pinned memory buffer (a fast copy within DRAM), resumes training immediately, and writes the buffer to NVMe in a background thread. The snapshot captures a consistent point-in-time view of the model state. The risk is that if a node fails during the asynchronous write, the local NVMe may contain an incomplete shard. The mitigation requires either waiting for the async write to complete before considering the checkpoint durable, or replicating the in-memory snapshot to a peer node before allowing the next training step to overwrite the snapshot buffer.

The choice between synchronous and asynchronous checkpointing depends on the ratio of T_{write} to $T_{\text{iteration}}$. If T_{write} is small relative to $T_{\text{iteration}}$ (for example, 1 second vs. 200 ms per iteration, meaning the checkpoint pause costs 5 iterations), synchronous checkpointing is acceptable. If T_{write} is large (for example, 30 seconds for a 500B+ parameter model), the training pause is prohibitive, and asynchronous checkpointing becomes essential.

Beyond the choice of synchronous or asynchronous methods, **checkpoint format optimization** reduces the per-node I/O volume. In the simplest data-parallel setup, every node holds an identical copy of the model weights, so a naive checkpoint would redundantly save the same data from every node. Sharded checkpoint protocols avoid that redundancy by saving shards instead of full replicas. DeepSpeed's **zero-redundancy checkpointing** uses its ZeRO optimizer-state partitioning: each of the H nodes saves only its $1/H$ shard of the optimizer state, and the system saves one consolidated copy of the weights. PyTorch's **Distributed Checkpoint (DCP)** protocol follows the same storage principle by writing each rank's sharded data to a distinct file. For our 175B model using ZeRO-3 across 256 nodes, the result is concrete: each node writes only its roughly 6.8 GB shard of the total 1,750 GB state, reducing the local write from minutes to around a second on one NVMe drive, or well below a second on a local stripe. The tiered-staging derivation in section 4.7 already threaded these per-node shard sizes and write times through the running example, and the same arithmetic governs distributed coordination: the exposed pipeline pause is the local NVMe write, while the durable copy to the parallel file system overlaps with subsequent training.

With the checkpoint accounting now derived in full, the running example's storage demands can be consolidated against the same hierarchy, so the training dataset, checkpoints, and archive lineage are visible side by side.

Systems Perspective 4.3: The 175B model's storage footprint

Table 4.5 assembles the complete storage picture for our running example, a 30-day training run of a 175B-parameter model on 256 nodes.

Table 4.5: 175B-parameter training storage footprint: Volume and primary storage tier for each data category in a 30-day, 256-node training run.

Category	Volume	Primary Tier
Training dataset (compressed source)	3 TB	Object Storage
Training dataset (tokenized, per epoch)	6 TB	Object Storage → NVMe cache → Host DRAM
Model weights (FP16)	350 GB	GPU HBM (distributed)
Optimizer state (FP32)	1,400 GB	GPU HBM (ZeRO-partitioned)
Single checkpoint (full)	1,750 GB	NVMe → PFS → Object Storage
All checkpoints (30 days, 10-min interval)	7.6 PB	NVMe (transient) → PFS (recent) → Object (archive)
Archive (retained checkpoints + dataset)	~84.2 TB	Glacier

The total data moved through the hierarchy is approximately 7.6 PB of checkpoint data plus 6 TB per 1 epoch of training data reads. For a single-epoch language model training run, checkpoint I/O dominates data loading I/O by a factor of over 1,260.

Feature stores and model registries, the operational systems that manage feature serving and model versioning, are also built on top of this storage hierarchy, depending on the same tiering, durability, and latency boundaries introduced here. Training streams its fuel sequentially and writes checkpoints in bursts; retrieval-augmented inference reverses that access pattern, replacing sequential streaming with random graph traversal, so the same hierarchy must now serve a workload it was not tuned for. Vector databases, specialized storage systems for approximate nearest-neighbor search in embedding spaces, serve the retrieval-augmented generation pipeline by making that reversed access pattern operational.

4.8 Retrieval Infrastructure: Vector Indexes

The storage hierarchy so far has optimized the training fuel line: large shards, sequential reads, and checkpoint bursts. Retrieval-augmented inference adds a storage workload that belongs in the same chapter because it reverses the access pattern again. A retrieval system first converts documents into vectors that represent semantic meaning, then serves an inference request by finding nearby vectors and returning their source documents as context. Instead of streaming sequential blobs via formats like TFRecord or Parquet, the storage system now traverses high-dimensional vector graphs.

Systems Perspective 4.4: The IOPS bottleneck in high-dimensional search

This architectural pattern shifts the primary storage bottleneck from **sequential read bandwidth** to **random-access IOPS**. Vector databases must traverse high-dimensional graph indexes to find approximate nearest neighbors across billions of embeddings. The performance is governed by a strict physical constraint: the **Recall vs. Latency vs. Capacity** trade-off. Maximizing recall requires traversing a larger portion of the index, driving up IOPS and latency. The index is

large because each embedded document chunk occupies substantial storage: a 1,536-dimension embedding in FP32 (4 bytes per value) consumes approximately 6.1 KB per vector, so a billion-document corpus requires roughly 6.1 TB of raw vector storage before the graph adjacency lists that enable fast navigation. If those adjacency lists add 50 to 100 bytes per node, they add roughly 50 GB to 100 GB, making raw vectors plus minimal graph metadata about 6.2 TB at the high end for a billion-entry corpus. Full production indexes can be larger once identifiers, graph levels, allocator overhead, deleted-entry slack, and replication are included, but those are implementation overheads rather than the raw vector-plus-adjacency footprint. At that scale, the index no longer fits in DRAM, which is the design pressure that motivates disk-backed alternatives. In-memory hierarchical navigable small world (HNSW) graph indexes make the same graph-navigation idea fast when the index fits in DRAM (Malkov and Yashunin 2020), while hybrid search combines vector similarity with lexical or metadata filters and therefore stresses both random vector traversal and document fetches. Disk-backed graph indexes such as DiskANN show how billion-scale search can trade DRAM footprint against SSD-backed graph traversal while targeting high recall and low query latency (Subramanya et al. 2019).

This makes retrieval infrastructure a useful counterexample to the training fuel line. The same hierarchy still applies, but the objective changes: training storage hides sequential bandwidth behind prefetching, whereas vector search spends storage budget on the index levels that keep random graph traversal inside the serving latency SLO. A disk-backed index is economical only if the extra SSD traversals do not erase the quality gain from searching a larger corpus. The retrieval workload thus reuses the same physical hierarchy while inverting the objective, which closes the access-pattern contrast that the fuel line opened.

4.9 The Synthetic Fuel Line

The storage architecture changes again when the fleet exhausts its supply of human-authored training data, because synthesized data must carry its own provenance. As high-quality human-generated text becomes harder to expand, the fleet encounters the **Data Wall**: the point where additional training samples must increasingly come from synthesis, simulation, curation, or expensive human collection (Villalobos et al. 2022). This shift from simple collection to managed data generation sends more than payload bytes through the hierarchy. A "synthetic fuel line" must also store the **provenance chain** of every sample: which model generated it, which filters accepted or rejected it, which judge or verifier models scored it, and which downstream run consumed it. Without that lineage, the system risks **model collapse**, where a model degrades by training on its own unverified errors (Shumailov et al. 2024).

Napkin Math 4.9: The synthetic tax

Problem: Calculate the storage amplification of a 1 TB synthetic dataset that requires cryptographic lineage and multi-model verification.

1. **Raw Payload**: 1 TB.
2. **Provenance Overhead**: 40 percent extra for lineage hashes, generation logs, and reward-model scores.
3. **Verification Factor**: To avoid "self-poisoning," each sample is verified by 3 independent "judge" models.
4. **Result**: Total footprint = 1 TB $\times$ 1.4 $\times$ 3 = 4.2 TB.

Systems insight: Synthetic data is *verified data*, and verification has a 4.2× storage tax. In the Machine Learning Fleet, storage moves from being a simple bit-bucket to a *provenance engine*. A system that cannot store the *why* and *who* behind a synthetic token risks poisoning the future of the fleet with its own past mistakes.

The synthetic fuel line closes the chapter's storage arc by making lineage part of the payload. Large-scale training, checkpointing, retrieval, and synthetic-data generation stress different tiers, but each failure mode comes from the same mistake: treating storage as passive capacity instead of an active system that controls throughput, reproducibility, and model quality.

4.10 Fallacies and Pitfalls

Storage design errors are among the most expensive mistakes in ML infrastructure because they are discovered late, when training begins, and remediation requires either expensive hardware upgrades or time-consuming data reformatting. The following fallacies and pitfalls capture the most common errors that experienced storage engineers make when first encountering ML workloads.

Fallacy: *NVMe is fast enough to feed GPUs directly without pipelining.*

A single NVMe drive delivers 7 GB/s, while an H100 consumes 3.35 TB/s from HBM. The gap is roughly 478.6×. Even four drives in RAID-0 only close this gap to about 120× before overhead. Without pipelining and prefetching to hide the latency of loading from NVMe to host DRAM to HBM, every batch transfer introduces a stall equal to the transfer time. NVMe speed is necessary but nowhere near sufficient; the entire pipeline architecture (multi-worker loading, prefetch buffers, async transfers) exists precisely because no single storage device can match accelerator bandwidth. The confusion arises because NVMe bandwidth is quoted in *absolute terms* (GB/s), which sounds impressive, but the relevant metric is the *ratio* of storage bandwidth to accelerator consumption rate, and this ratio is unfavorable by two to three orders of magnitude.

Pitfall: *Relying on prefetching to hide object storage latency.*

Prefetching hides *average* latency, not *tail* latency. Object storage P99 latency can spike to several hundred milliseconds during congestion or cross-region failover. A prefetch buffer of depth Q_{prefetch} absorbs variance up to $Q_{\text{prefetch}} \times T_{\text{compute}}$ milliseconds. If a tail-latency spike exceeds this window, the buffer drains and the accelerator stalls. For training jobs that run for weeks, even rare P99.9 spikes occur frequently enough (thousands of times per day at scale) to measurably reduce utilization. The mitigation is not to assume prefetching solves all problems, but to provision buffer depth based on measured tail latency and to stage data on local NVMe where possible.

Fallacy: *Millions of small training files are harmless at scale.*

Storing each training sample as an individual file (one JPEG per image, one JSON per text sample) is natural for data collection but catastrophic for training at scale. Each file requires metadata operations (`open()`, `stat()`, `close()`) that serialize on the metadata server. At 10,000 concurrent workers, metadata operations become the bottleneck long before data bandwidth saturates. The solution is to aggregate samples into large sequential shards (TFRecord, WebDataset tar, Parquet) during preprocessing. This reduces metadata operations by 10,000 to 100,000× and transforms random access patterns into sequential streaming.

Pitfall: *Ignoring egress costs when designing the storage architecture.*

Cloud egress costs are invisible during development but can dominate at scale. Under a representative $0.09/GB cross-boundary egress fee, a training job that reads a 100 TB dataset from object storage 10 times incurs $90,000 in egress fees, far exceeding the annual storage cost of $27,600 under the object-storage price assumption above. These fees apply when reads cross a region, provider, or internet boundary; same-region reads may be free or discounted, which is exactly why the cost stays hidden until a cross-region or multi-cloud topology makes it unavoidable. Teams that prototype with small datasets on object storage and then scale up discover that their storage architecture is economically unsustainable. The fix is to budget egress costs explicitly and to design the data pipeline to minimize cross-tier transfers, typically by staging data on local NVMe at job start rather than streaming from object storage each epoch.

Fallacy: *Average write bandwidth is enough for checkpoint design.*

Checkpoints are written by all nodes simultaneously, creating bursts that can exceed steady-state bandwidth by 10× or more. A parallel file system sized for average write load will bottleneck during checkpoint storms, extending T_{write} and reducing the effective training throughput. The parallel file system must be provisioned for peak burst bandwidth, not average, even though the burst capacity sits idle most of the time.

Pitfall: *Assuming faster accelerators automatically improve training throughput.*

When an organization upgrades from A100 to H100 GPUs, they expect training time to decrease proportionally to the compute improvement. If the storage pipeline was already marginal on A100s, delivering data just fast enough to avoid stalls, the faster H100 compute simply hits the I/O wall sooner. The accelerator upgrade reduces T_{compute} without reducing $T_{\text{I/O}}$, which means the data stall ratio increases. Every accelerator upgrade must be accompanied by a storage pipeline audit to ensure that the faster compute does not simply expose a previously hidden storage bottleneck.

Fallacy: *RAID configurations optimized for databases are suitable for ML training.*

Database-oriented RAID configurations like RAID 5 or RAID 6 prioritize redundancy over raw bandwidth, incurring a 15–25 percent write performance penalty due to parity calculations. This overhead is unnecessary for ML workloads where training data is immutable and already backed up in durable object storage. For local NVMe caches feeding accelerators, the correct choice is RAID 0 (striping), which maximizes sequential read throughput by combining the bandwidth of all drives without parity overhead. The data on local NVMe is a *cache*, not a *source of truth*; losing it to a drive failure costs a re-staging operation, not data loss.

Pitfall: *Creating full copies for each dataset snapshot.*

Naive dataset versioning is financially and operationally unsustainable at scale. For a 10 TB dataset, creating a full copy for each version quickly exhausts storage budgets and slows experimentation cycles. A 5 percent change between versions stores 9.5 TB of redundant data. The correct approach uses content-addressable storage or delta encoding, where only changed samples are stored. This reduces the storage cost of a new version from 10 TB to roughly 500 GB, a 20× reduction. Tools like DVC (Data Version Control) and lakeFS implement this pattern, tracking dataset lineage without duplicating unchanged content.

Fallacy: *Compression always helps because it reduces the amount of data to read.*

This is only true if the pipeline is I/O-bound. Compression reduces I/O volume but increases CPU load due to decompression. For a CPU-bound pipeline where complex data augmentation already saturates the host processor, adding decompression work makes the bottleneck *worse*, not *better*, leading to a net decrease in throughput to the accelerator. The correct approach is to profile the pipeline first: if the host CPU is the bottleneck, use uncompressed or lightly compressed formats (LZ4); if storage or network I/O is the bottleneck, use aggressive compression (zstd, gzip). The optimal compression level is a property of the pipeline's bottleneck, not a universal constant.

Pitfall: *Choosing a compression format before profiling the pipeline bottleneck.*

Teams often standardize on one compression format for operational simplicity, then apply it to every dataset regardless of access pattern. That shortcut hides the real question: whether the job is paying more for bytes moved or cycles spent decoding them. A storage pipeline should choose compression after measuring CPU utilization, storage bandwidth, network bandwidth, and accelerator idle time on representative batches; otherwise the format becomes another fixed assumption that moves the bottleneck rather than reducing it. These fallacies all return to the same accounting exercise: where each byte lives, how often it moves, and which tier pays the cost.

4.11 Summary

Storage in ML systems is not a passive repository; it is an active, multi-tiered pipeline whose sole purpose is to keep accelerator HBM populated with data. The hierarchy spanning HBM, host DRAM, local NVMe, parallel file systems, object storage, and cold archive exists because no single technology can simultaneously deliver the bandwidth, capacity, and cost profile that large-scale training demands. Roughly three orders of magnitude in aggregate bandwidth separate HBM from cold archive, and each intermediate tier serves as a staging buffer that absorbs the mismatch between the rate at which accelerators consume data and the rate at which persistent storage can supply it. The data pipeline throughput equation, $\text{BW}_{\text{required}} = N_{\text{GPU}} \times \eta_{\text{target}} \times D_{\text{vol,batch}} / T_{\text{iteration}}$, provides the quantitative foundation for sizing every tier: miss the required bandwidth at any level and expensive accelerators idle; over-provision and capital is wasted on storage capacity that sits underutilized.

What makes ML storage particularly challenging is that these workloads invert nearly every assumption baked into decades of storage system design. Databases optimize for random IOPS, cacheable working sets, and continuous small writes. ML training demands sequential streaming throughput over datasets that dwarf all cache levels, punctuated by massive checkpoint bursts that saturate bandwidth for seconds before returning to silence. The metadata overhead of small files,

harmless in traditional workloads, becomes the dominant bottleneck when millions of individual samples must be opened, stat'd, and closed per epoch. Aggregating samples into large shards, choosing sequential streaming formats like WebDataset or TFRecord, and designing prefetch pipelines sized for P99 tail latency rather than average latency are all direct engineering responses to these inverted access patterns. GPUDirect Storage pushes this optimization further by eliminating the CPU from the data path entirely, freeing host cores for the augmentation work that the training pipeline also requires.

The economics of the hierarchy are equally consequential. The orders-of-magnitude cost difference between HBM and archive storage mandates a tiering strategy, but the true cost of data delivery extends well beyond per-gigabyte storage prices. Egress fees for moving data out of object storage, IOPS charges for metadata-heavy access patterns, and the opportunity cost of idle accelerators waiting on slow checkpoints all factor into the total cost of ownership. Checkpoint staging, where models write first to fast local NVMe and replicate asynchronously to shared storage, exemplifies how careful pipeline design can decouple training pause time from the performance limitations of the underlying file system.

Engineers who internalize the storage hierarchy gain a systematic diagnostic framework for training performance. When a cluster reports low accelerator utilization, the instinct is often to suspect the compute configuration or the model itself. In practice, the root cause is frequently buried in the data path: a data loader reading individual files instead of shards, a prefetch buffer sized for average latency rather than tail latency, a checkpoint strategy that blocks training while writing to a slow parallel file system, or a NUMA-unaware memory allocation that halves effective DRAM bandwidth. Diagnosing these failures requires understanding which tier is the bottleneck and why, a question that the pipeline equation and the storage hierarchy framework make answerable.

This diagnostic perspective connects directly to how a training job is split across the fleet, because the splitting strategy dictates the demand patterns placed on storage. Data-parallel layouts replicate the model on every node, creating uniform read patterns but massive checkpoint redundancy as every node saves the same parameters. Pipeline-style layouts create sequential dependencies between stages, where a data loading stall in an early stage cascades down the pipeline and starves downstream work. Expert-style layouts create nonuniform access patterns where different accelerator groups require different subsets of the data at any given time. Each strategy imposes a unique storage signature, which is why storage cannot be designed independently from the computation it feeds.

Key Takeaways: Feed the accelerators or waste them

- **HBM is the destination**: Every lower tier exists to keep GPU HBM populated. The roughly 30× aggregate bandwidth gap between HBM and object storage (and the much larger per-client gap once a single instance pulls from a shared object endpoint) drives every design decision in the ML storage hierarchy.
- **ML workloads invert storage assumptions**: Traditional caching, IOPS optimization, and small-write durability patterns all produce the wrong answer for ML training. Throughput (GB/s) matters more than IOPS, and small files make metadata the first bottleneck; aggregated shards reduce metadata operations by orders of magnitude.
- **The pipeline equation governs design**: Required bandwidth scales linearly with GPU count and inversely with iteration time. Use $\text{BW}_{\text{required}} = N_{\text{GPU}} \times \eta_{\text{target}} \times D_{\text{vol,batch}}/T_{\text{iteration}}$ to size every tier.
- **Pipelining hides average latency, not tail latency**: Prefetch buffers must be sized for P99 I/O latency, not average, to prevent accelerator stalls at scale. Buffer depth of $\lceil T_{\text{I/O,p99}}/T_{\text{compute}} \rceil$ is the minimum.
- **GPUDirect Storage eliminates CPU bottlenecks**: GDS bypasses the CPU in the data path, reducing per-transfer latency by 4× and freeing CPU cores for augmentation.

- **Checkpoint staging minimizes T_{write}**: Write to local NVMe first, then replicate asynchronously to shared storage. This decouples checkpoint pause time from parallel file system performance.
- **Economics drive tiering**: The 5× cost difference between local NVMe and object storage (and the >3,000× span from HBM to archive) mandates a tiering strategy. Egress fees often exceed storage fees; design for total cost of data delivery, not storage cost alone.

Every other chapter in this volume optimizes something the accelerators do. This one optimizes what reaches them. The whole storage hierarchy exists to keep the most expensive resource in the building from going idle, and that reframes where performance is won: not in the compute, which is rarely the bottleneck, but in the supply chain that feeds it. At fleet scale the binding constraint is the rate at which data arrives, not the rate at which it is processed, which is why a cluster can sit at half utilization with every accelerator healthy. A system is only as fast as the tier that feeds it.

What's Next: From storage to splitting

Infrastructure is complete. We have built accelerators that compute at petaFLOP/s (Chapter 2), wired them with fabrics that move gradients at TB/s (Chapter 3), and now established the storage hierarchy that feeds them data. The next challenge is not a hardware problem but an algorithmic one: partitioning a single training job across these thousands of resources. Chapter 5 explores the parallelism strategies (data, tensor, pipeline, and expert parallelism) that split computation across the fleet.

Research Questions: For further inquiry

- How should a tiered storage pipeline be designed to keep accelerator HBM fed for a specific workload?
- How do ML workloads invert database-era storage assumptions, and what data format choices follow?
- When should data be prestaged to local NVMe rather than streamed from a parallel file system or object store?
- How should checkpoint storage balance pause time, durability, restore verification, and retention cost?

II

DISTRIBUTED ML

Part II

Distributed ML Principles

Coordinating the fleet requires solving problems that a single machine does not. If Part I built the physical substrate, Part II establishes the logic of distribution: the algorithms, protocols, and coordination strategies that transform a collection of independent accelerators into a singular, unified machine. The physics of distributed learning governs this transition, supplying the first-principles reasoning that explains why adding more GPUs does not always yield linear speedups, and why communication, not computation, is often the hidden bottleneck.

At this scale, the engineering challenge is a trade-off between parallelization and coordination. We can partition a model to reduce per-device memory pressure, but we necessarily increase the communication volume required to keep the weights synchronized. We can scale to thousands of nodes to reduce training time, but we simultaneously decrease the mean time between failures (MTBF), making fault tolerance a mandatory system component rather than an operational luxury. These principles decompose the communication tax introduced in Part I and add the reliability tax of scale.

The first pressure is the economics of scale itself: frontier capability demands compute, but each improvement gets progressively more expensive.

Principle 9: The Universal Scaling Law

Invariant: For frontier foundation models, loss ($\mathcal{L}$) improves as a power-law function of compute (C), dataset size (D), and parameters (P), with γ denoting the empirical scaling exponent. The compute-only simplification used to illustrate diminishing returns is:

$$\mathcal{L}(C) \propto C^{-\gamma}$$

Implication: At the frontier of model capability, scale is the binding requirement, not an optimization choice. Because loss improves sublinearly with compute, each equal-sized improvement requires disproportionately more compute; this diminishing return drives the transition from single-server training to warehouse-scale clusters.

That demand for scale turns each training step into a negotiation between faster local work and slower global coordination.

Principle 10: The Fleet Law (Distributed Step Time Law)

Invariant: The time to complete one training step across N workers or accelerators is the sum of parallelizable computation, communication, synchronization, and the overlapped portion that can be hidden, with $0 \leq T_{\text{overlap}} \leq T_{\text{comm}}(N) + T_{\text{sync}}(N)$.

$$T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}$$

Implication: Scaling is a race between parallelizable compute (which shrinks with N under ideal partitioning), communication overhead, and synchronization overhead. To scale efficiently, algorithms must reduce or amortize $T_{\text{comm}}(N)$ and $T_{\text{sync}}(N)$ (for example, through

Gradient Accumulation, which alters the effective batch and convergence regime, or through compression, which can introduce numerical error) and maximize T_{overlap} through **Communication Hiding** and pipelined execution.

The communication term in that negotiation is governed by the interconnect's latency and bandwidth, so message size determines which optimization helps.

Principle 11: The Bandwidth-Latency Trade-off (α-β Model)

Invariant: Communication time is a function of fixed latency (α), message size (n), and message-dependent bandwidth (β).

$$T(n) = \alpha + \frac{n}{\beta}$$

Implication: Small messages—for example, mixture-of-experts (MoE) routing metadata—are latency-bound; large messages (for example, gradients) are bandwidth-bound. Optimization strategies must match the regime: fuse small messages to amortize α, compress large messages to improve β.

Even when communication is tuned, a larger fleet fails often enough that progress itself must be protected.

Principle 12: The Young-Daly Checkpoint Law

Invariant: The optimal checkpoint interval (τ_{opt}) balances the cost of writing the checkpoint (T_{write}) against the expected cost of reworking lost progress due to system-level failures ($\text{MTBF}_{\text{system}}$).

$$\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$$

Implication: Checkpointing is not "free." As cluster size grows, MTBF drops, forcing more frequent checkpoints. This demands high-bandwidth storage (burst buffers) to prevent I/O from dominating training time.

Fault tolerance is one example of a broader rule: distributed systems rarely remove overhead; they move it.

Principle 13: The Displacement of Overhead

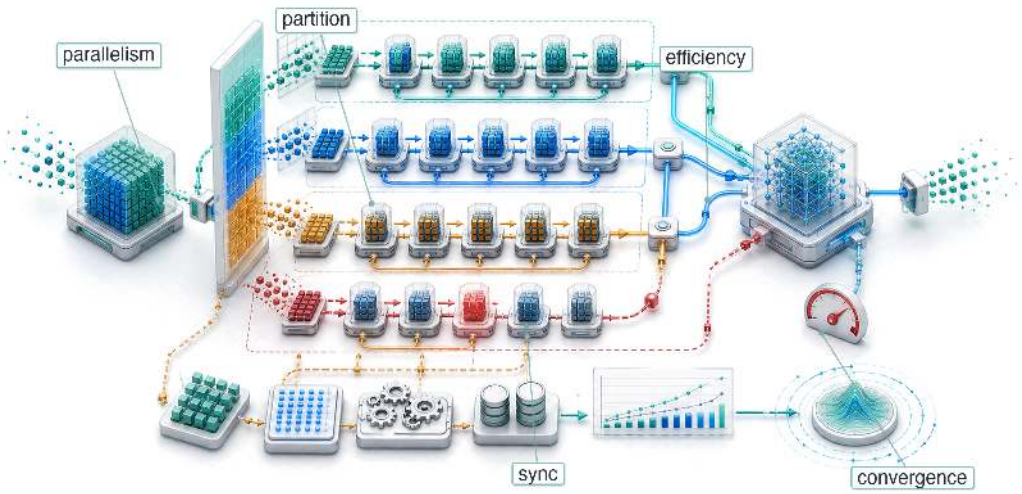
Invariant: Overhead in a distributed ML system cannot be eliminated, only relocated among Compute, Communication, and Coordination (the C^3 taxonomy). Reducing one necessarily increases at least one other.

Implication: Asynchronous training eliminates coordination barriers ($T_{\text{sync}}(N) \to 0$) but introduces gradient staleness that manifests as additional training iterations ($T_{\text{compute}} \uparrow$). Pipeline parallelism reduces communication volume but adds pipeline bubble time ($T_{\text{sync}}(N) \uparrow$). There is no free lunch in distributed systems; the C^3 taxonomy reveals where the system pays the bill.

With these invariants in place, Part II establishes how the fleet coordinates distributed training work: the parallelism strategies (data, tensor, pipeline, and expert) that partition the workload across thousands of accelerators, the collective operations (AllReduce, AllGather, AllToAll) that bind those accelerators into a coherent computer, the fault-tolerance mechanisms that absorb the inevitable failures of scale, and the orchestration layer (Slurm, Kubernetes, and their kin) that allocates the fleet to workloads. Together, these chapters develop the algorithmic and operational discipline that turns the physical fleet of Part I into a working distributed training system.

5

Distributed Training

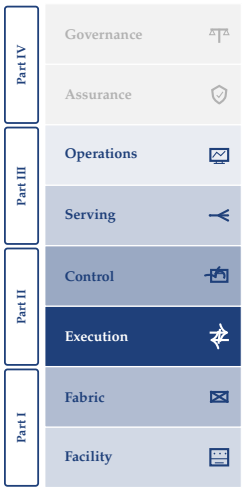


5.1 Why Distribution Is Necessary
5.2 The Distributed Training Step
5.3 Data Parallelism
5.4 Scaling Efficiency and Convergence
5.5 Model Parallelism
5.6 FP8 for distributed training
5.7 Hybrid Parallelism
5.8 Multi-Model Training: RLHF and Alignment
5.9 Parallelism Strategy Comparison
5.10 From Principles to Systems
5.11 Fallacies and Pitfalls
5.12 Summary

Purpose

Why does the linear logic of "more hardware = faster training" collapse at the bisection bandwidth wall?

Distributed training appears simple: split the work across machines and combine the results. As the machine learning fleet grows, however, a new physics emerges. Communication costs scale with the number of machines while computation per machine shrinks, until synchronization overhead dominates and adding hardware actively degrades performance. When a job trains on a single accelerator, the design optimizes for arithmetic intensity; when it trains on 10,000 accelerators, the design optimizes for communication intensity. The scaling ceiling is not a bug to be fixed but a fundamental property of the reliability gap and communication-computation ratio: coordinating independent machines requires moving terabytes of state across networks that are orders of magnitude slower than on-chip memory. The art of distributed training is managing this tension—partitioning work to minimize the coordination tax, overlapping communication with computation to hide latency, and choosing synchronization strategies that balance consistency against throughput. Without this understanding, organizations waste millions on hardware that sits idle waiting for gradients to arrive, or produce models that never converge because stale updates corrupted the optimization. Distributed training is the C^3 taxonomy in motion: every design choice trades parallelized compute against network-bound communication and synchronizing coordination.



Learning Objectives

- Apply the fleet law to diagnose compute-, memory-, communication-, or coordination-bound distributed training regimes
- Calculate scaling efficiency, critical batch limits, and synchronization overhead for data-parallel training jobs
- Design memory-sharding plans that trade accelerator capacity for additional communication
- Map tensor, pipeline, expert, and data parallelism onto hardware bandwidth tiers and model structure
- Construct microbatch pipeline schedules that minimize bubbles while preserving throughput and convergence stability
- Select synchronization and low-precision policies using staleness, straggler tolerance, numerical stability, and cluster heterogeneity
- Synthesize hybrid parallelism configurations for frontier, recommendation, and alignment training constraints

5.1 Why Distribution Is Necessary

The accelerator hierarchy, network fabric, and storage pipeline form the physical foundation of the fleet. The remaining challenge is algorithmic: partitioning a single training job across thousands of resources without losing the semantics of one coherent optimization process. The universal scaling law (principle 9) explains why this pressure keeps increasing: frontier quality improvements demand disproportionately more compute, data, and parameters, so the training problem eventually outgrows any single machine.

A single accelerator with 100 terabytes of memory and an exaflop of compute would make distributed training unnecessary. Real systems instead impose finite HBM capacity, finite interconnect bandwidth, and finite failure budgets, so training must be partitioned across many independent chips. In the fleet stack framework shown in figure 1.13, distributed training represents the distribution layer: the logic that partitions the *mathematical* workload across the *physical* fleet. The strategies defined here, including data, tensor, pipeline, and hybrid parallelism, create the traffic patterns that the interconnect must carry.

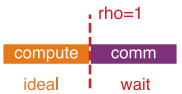
5.1.1 The physics of the cluster

Before optimizing algorithms, we must understand the physical constraints of the Machine Learning Fleet. The performance of any distributed training job is governed by the fleet law (principle 10), introduced in section 1.5.2, which decomposes the per-step time:

$$T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}$$

The critical term is the **communication-computation ratio**, $\rho = T_{\text{comm}}(N)/(T_{\text{compute}}/N)$. This ratio determines whether a cluster behaves as a supercomputer or a collection of idling heaters.

Two regimes follow from this ratio. A compute-bound cluster, where $T_{\text{compute}}/N \gg T_{\text{comm}}(N)$, spends most of its time multiplying matrices; this is the ideal state, typical for large batch sizes on dense models such as ResNet. A communication-bound cluster, where $T_{\text{comm}}(N) \approx T_{\text{compute}}/N$, spends significant time waiting for gradients or activations to arrive, the common state for large language models (LLMs) and DLRM-style recommendation models, where parameter synchronization saturates the network. Section B.5.2 works this ratio into a diagnostic framework that classifies a workload as compute-bound, memory-bound, or communication-bound from its measured bandwidth and arithmetic intensity, turning the qualitative regimes above into a repeatable test.



Past a communication-to-compute threshold, scaling stops being ideal.

5.1.2 Multi-machine training requirements

Three concrete signals indicate when distributed training becomes necessary rather than merely beneficial. The first signal is **memory exhaustion**: model parameters, optimizer states, and activation storage exceed single-device capacity. For full mixed-precision training with Adam, the parameter, gradient, and optimizer-state budget alone can exceed an 80 GB accelerator at roughly 5 billion parameters before activations; larger 10–20B models require sharding, offload, or other memory-saving techniques (Rajbhandari et al. 2020). Section G.1 records the 80 GB H100 and A100 capacity figures used in these budgets throughout the chapter, so the reader can trace every memory ceiling back to a single documented source.

The second signal is unacceptable **training duration**. Even when a model fits, single-device training may require weeks or months to converge, making wall-clock time itself a systems constraint. GPT-3's 175B-parameter training run used a cluster of V100 GPUs (Brown et al. 2020), illustrating why the calendar, not just HBM capacity, forces distribution at this scale.

The third signal is **dataset scale**. When training data reaches multiple terabytes, as occurs in large-scale vision or language modeling tasks, a single machine is no longer the natural unit of storage or input throughput. Distributed training then becomes a way to feed the model as much as a way to hold it.

5.1.3 Distributed training complexity trade-offs

Distribution changes the optimization problem by adding costs that do not exist inside one machine. Three complexity dimensions decide whether a parallelism strategy is viable, because any one of them can become the binding ceiling. Communication overhead is the cost of synchronizing gradients: for a model with P parameters distributed across N devices, all-reduce operations must transfer approximately $2P(N-1)/N$ gradient values per step, multiplied by the bytes per gradient element, and on commodity networks this can dominate computation time. Fault tolerance grows harder as the cluster grows, because the expected number of failures per unit time rises linearly with cluster size while the probability that the entire cluster survives an interval without any failure decays exponentially; if a 100-node cluster has 99.9 percent per-node hourly survival, the cluster-level failure probability is 9.5 percent per hour, corresponding to an MTBF of about 10 hours. Section E.1.2 derives this cascade from per-node survival to cluster MTBF and works the same calculation through a full example, so the reader can reproduce why MTBF collapses as the cluster grows. Algorithmic stability closes the set, because large batch sizes from data parallelism affect convergence behavior, requiring learning rate scaling and warmup strategies that single-machine training does not require (Goyal et al. 2017). Together, these costs explain why distributed training is a constraint satisfaction problem rather than a hardware multiplication exercise.

5.1.4 Single-machine to distributed transition

The systematic optimization methodology established for single-machine training extends to distributed environments with important adaptations. Profiling must capture inter-device communication patterns and synchronization overhead in addition to computation and memory metrics. The solution space expands to include data parallelism, model parallelism, pipeline parallelism, and hybrid approaches. Figure 5.1 visualizes this three-dimensional configuration space.

The key insight from figure 5.1 is that the total accelerator count is $N_{\text{total}} = d \times p \times t$: each axis is independent, and training systems select a specific coordinate in this cube based on the memory, compute, and bandwidth constraints of the target model and cluster.

5.1.5 Engineering trade-offs: Selecting a parallelism strategy

Choosing the right parallelism strategy is not a matter of preference; it is a constraint satisfaction problem governed by parameter count (P), batch size (B), and interconnect bandwidth. Table 5.1 quantifies the parallelism communication costs for each strategy, revealing which approaches are physically feasible for a given hardware topology.

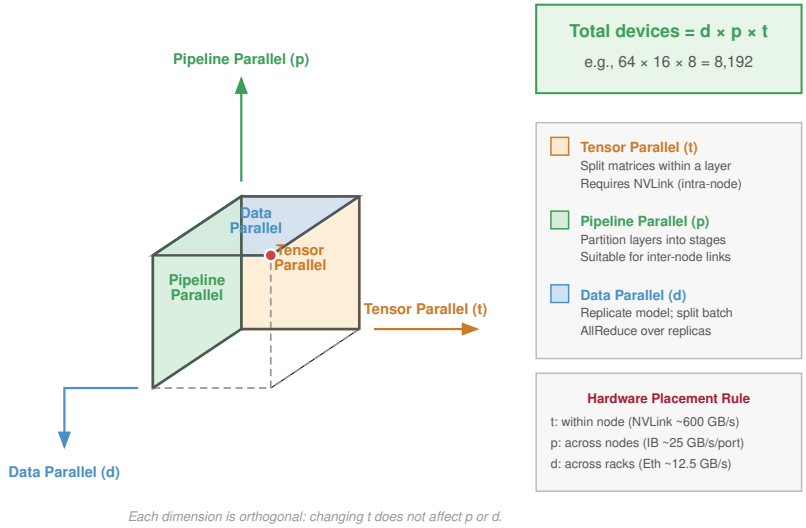


Figure 5.1: The 3D Parallelism Cube: A conceptual visualization of the three orthogonal scaling axes: Data Parallelism (replicating the model), Pipeline Parallelism (splitting depth), and Tensor Parallelism (splitting layers). Large-model training occupies a specific point (d, p, t) within this cube to balance memory usage, compute efficiency, and communication overhead.

Table 5.1: Parallelism Communication Costs: Tensor parallelism has the highest communication frequency (per layer), confining it to intra-node (NVLink) usage. Pipeline parallelism has the lowest communication volume (boundary activations only), making it suitable for inter-node (Ethernet/InfiniBand) scaling. Data parallelism sits in the middle but scales poorly as communicated gradient bytes M_{grad} grow, eventually requiring memory-efficient sharding.

Strategy	Communication Pattern	Comm. Volume	Hardware Constraint
Data Parallel (DP)	AllReduce Gradients	$\propto M_{\text{grad}}$ (gradient bytes)	Requires high bisection BW
Tensor Parallel (TP)	AllReduce Activations	$\propto B \times N_L$ (Layers)	Critical: Needs NVLink
Pipeline Parallel (PP)	Point-to-Point (P2P)	$\propto B \times S \times d_{\text{model}}$ (Activations)	Low BW (Ethernet is sufficient)

The bandwidth requirements impose a hard constraint on hardware placement: each parallelism strategy must be matched to the interconnect available between the GPUs that need to communicate, or the design fails before it leaves the whiteboard. A quick back-of-envelope bandwidth feasibility check catches that mismatch before any code is written.

> **Systems Perspective 5.1: Bandwidth feasibility check**
>
> Tensor parallelism across server racks connected by standard Ethernet will stall. The communication volume scales with batch size B and layer count N_L. It requires the 600 GB/s–900 GB/s throughput of NVLink. For cross-rack scaling, the design must switch to pipeline or data parallelism to respect the physics of the network.

Figure 5.2 formalizes this constraint satisfaction process as a decision tree, showing how model size and hardware topology determine the viable parallelism strategies.



The decision tree reveals that parallelism strategy selection is not a preference but a consequence of physical constraints. This is a constraint-level preview: the communication-volume formulas above name what each strategy moves, but their full meaning depends on the mechanics of tensor, pipeline, and hybrid parallelism developed over the rest of the chapter. Section 5.9 completes the filter once those mechanics are in place. The next question is *how* these constraints shape the mechanics of a distributed training step on a real cluster.

5.2 The Distributed Training Step

The selection constraints of the previous section assumed the job must be split; the question now is what a distributed step must guarantee. The central challenge is ensuring that 1,024 GPUs, operating completely independently, agree on a single, mathematically rigorous set of updated weights at the end of each training iteration.

Definition 5.1: Distributed training

Distributed Training is a training methodology that partitions the optimization loop across multiple compute nodes (distributing either data, model layers, or individual tensor operations) and coordinates their outputs through synchronized communication primitives to produce a single coherent model.

1. **Significance**: Distributed training becomes necessary when a model's memory requirement exceeds a single accelerator's capacity. GPT-3 (175B parameters) requires approximately 350 GB in BF16—more than 4× the 80 GB capacity of a single H100. Training it requires at least 5 H100s for model sharding alone, while large runs may use hundreds or thousands of accelerators to reach tractable wall-clock time.
2. **Distinction**: Unlike distributed systems for independent requests (web serving, database reads) where nodes share no mutable state, distributed training requires every node to maintain a consistent view of model parameters—making gradient synchronization a mandatory coordination step, not an optional optimization.
3. **Common pitfall**: A frequent misconception is that distributed training scales linearly with node count. In practice, communication overhead grows with cluster size and the serial fraction of each step (Amdahl's Law): with 30 percent of a step's time spent on synchronization, the theoretical scaling ceiling is $1/0.30 \approx 3\times$ regardless of how many accelerators are added.

A useful mental model frames these distributed strategies as loop transformations, the same conceptual toolkit that compilers use to optimize sequential code. If we view the training process as a massive loop over data and layers, distributed strategies are simply *loop transformations* applied by the cluster-level compiler. The logical training loop nests three iterators (epochs, batches, layers), and each parallelism strategy unrolls one of them across devices:

Data parallelism is the parallel for-loop, unrolling the outer loop (the batch dimension) across devices so that each device runs the same code body on different data indices. Tensor parallelism is vectorization, or single instruction, multiple data (SIMD), splitting the inner loops (matrix multiplication) across devices in a cluster-scale SIMD where NVLink acts as the vector register file. Pipeline parallelism is instruction pipelining, splitting the sequential operations (layers) across devices; just as a CPU pipeline stages fetch, decode, and execute, the cluster stages Layer 1, Layer 2, and Layer 3 to keep all ALUs busy.

Whichever loop a strategy unrolls, distributed training[1] spreads the workload across machines that must coordinate to train a single model. Coordination here means keeping every model shard or replica on a compatible training step. Basic barriers can keep a small research run ordered, but long-running training jobs also need timeout, checkpoint, and recovery mechanisms so one failed worker does not waste days of compute. Chapter 7 examines those reliability engineering challenges in depth.

Each rung of the scaling path inherits the previous one's challenges. Single-GPU training needs only local memory management and forward/backward passes. Scaling to multiple GPUs within

1 **Distributed Training**: Google's DistBelief (2012) was an early framework for training neural networks across thousands of machines, but its parameter server architecture created bandwidth bottlenecks at central nodes. This limitation drove the shift to decentralized AllReduce patterns in successors like Horovod and PyTorch DistributedDataParallel (DDP), where replicated data-parallel workers synchronize gradients at cost $2(N-1)/N$ per worker rather than concentrating traffic at a single server (Dean et al. 2012; Sergeev and Balso 2018; S. Li et al. 2020; Patarasuk and Yuan 2009).

a node adds high-bandwidth communication, handled through NVLink[2] or PCIe with NCCL[3] optimization while preserving single-machine fault tolerance and scheduling.

The leap to multi-node training adds network communication overhead, fault tolerance requirements, and cluster orchestration. Because each stage compounds the previous one's bottlenecks, single-GPU performance must be optimized before scaling out, so inefficiency does not multiply across the fleet. Although frameworks abstract away much of this through sharded data parallelism and communication libraries, implementing distributed training efficiently still demands careful network configuration (InfiniBand tuning, topology-aware routing), infrastructure management through cluster schedulers, and debugging of nonlocal issues such as synchronization hangs and communication bottlenecks.

Despite this complexity, the core workflow is mechanically straightforward; the engineering challenge is making it fast and reliable at scale. The recurring cost is the gradient synchronization that aggregates results across devices, an overhead that compounds as systems scale, as section 5.4 quantifies.

Four approaches address different constraint regimes. Data parallelism divides the training data across machines while each maintains a full model copy, making it the simplest approach for models that fit in single-device memory. Model parallelism splits the model itself across devices when parameters exceed single-device memory. Pipeline parallelism partitions models into sequential stages that process microbatches concurrently, improving utilization over naive model parallelism. Hybrid approaches integrate multiple strategies, enabling training at scales where any single approach would fail. Each strategy becomes necessary only after its predecessor reaches a physical ceiling.

5.3 Data Parallelism

The simplest approach gives each GPU a complete, identical copy of the model and assigns it a distinct slice of the data. Data parallelism is the natural starting point for distributed training because it requires minimal changes to the single-device training loop.

Definition 5.2: Data parallelism

Data Parallelism is a distributed training strategy in which each worker holds a complete replica of the model and processes an independent shard of the minibatch, then synchronizes gradient updates via AllReduce so all replicas apply identical parameter changes each step.

1. **Significance**: With N workers each processing batch size B, the effective global batch size is $N \times B$, scaling throughput linearly while keeping per-worker memory constant. For a 1B-parameter model at 2 GB in BF16, 1,024 workers achieve 1,024$\times$ the single-GPU throughput—until the gradient AllReduce (2 GB per step at ring-optimal $2(N-1)/N$ per worker) exceeds backward compute time and creates the communication bottleneck.
2. **Distinction**: Unlike model parallelism, where parameters are partitioned so no single worker holds the full model, data parallelism requires every worker to have sufficient memory capacity to store the complete model state—making it inapplicable for models larger than a single accelerator's memory without combining it with optimizer-state or parameter sharding.
3. **Common pitfall**: A frequent misconception is that data parallelism scales indefinitely with worker count. Scaling the effective batch size B beyond the workload-dependent critical batch size degrades statistical efficiency, requiring more training steps to reach target loss and eroding the throughput gains from adding more workers (Shallue et al. 2019).

Each device trains a complete copy of the model using its assigned subset of the data. When training an image classification model on 1 million images using 4 GPUs, each GPU processes 250,000 images while maintaining an identical copy of the model architecture.

Data parallelism is most effective when the dataset size is large but the model size remains manageable, since each device must store a full copy of the model in memory. This method is widely used in image classification and natural language processing, where the dataset can be processed in

[2] **NVLink**: NVIDIA's point-to-point GPU interconnect delivers 600 GB/s–900 GB/s bidirectional bandwidth, roughly 24×–36× InfiniBand HDR per port. This bandwidth gap is why tensor parallelism, which requires AllReduce on every layer, is confined to intra-node communication, while pipeline and data parallelism tolerate the slower inter-node fabric.

[3] **NCCL (NVIDIA Collective Communications Library)**: NCCL provides topology-aware inter-GPU collective communication primitives, including AllReduce, across PCIe, NVLink, InfiniBand, and IP networks. Current NCCL deployments expose ring, tree, and related algorithm families through runtime selection and tuning controls, so training frameworks can avoid naive mappings that route all traffic through the slowest inter-node path (Jeaugey 2017; NVIDIA 2026b).

parallel without dependencies between data samples. When training a ResNet model (He et al. 2016) on ImageNet, each GPU can independently process its portion of images because the classification of one image does not depend on the results of another.

The effectiveness of data parallelism stems from a property of stochastic gradient descent. Gradients computed on different minibatches can be averaged while preserving mathematical equivalence to single-device training. This property enables parallel computation across devices, with the mathematical foundation following directly from the linearity of expectation.

Consider a model with parameters θ training on a dataset D. The loss function for a single data point x_i is $\mathcal{L}(\theta, x_i)$. In standard SGD with batch size B, the gradient update for a minibatch is:

$$g = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta \mathcal{L}(\theta, x_i)$$

In data parallelism with N devices, each device k computes gradients on its own minibatch B_k:

$$g_k = \frac{1}{|B_k|} \sum_{x_i \in B_k} \nabla_\theta \mathcal{L}(\theta, x_i)$$

When all workers use the same local batch size, the global update averages these local gradients:

$$g_{\text{global}} = \frac{1}{N} \sum_{k=1}^{N} g_k$$

Under that equal-batch assumption, the averaging is mathematically equivalent to computing the gradient on the combined batch $B_{\text{total}} = \bigcup_{k=1}^{N} B_k$:

$$g_{\text{global}} = \frac{1}{|B_{\text{total}}|} \sum_{x_i \in B_{\text{total}}} \nabla_\theta \mathcal{L}(\theta, x_i)$$

For unequal local batch sizes, the combined-batch gradient is the weighted average $g_{\text{global}} = (1/|B_{\text{total}}|) \sum_k |B_k| g_k$. The equivalence shows *why* data parallelism maintains the statistical properties of SGD training: distributing distinct data subsets across devices, computing local gradients independently, and averaging them approximates the full-batch gradient. The averaging step itself is an AllReduce over the local gradient tensors; section D.2.1 derives the ring and tree AllReduce cost models that set the price of this synchronization, so the reader can predict when it overtakes the local compute saved by parallelism.

Checkpoint 5.1: Data parallelism mechanics

Verify your understanding of how data parallelism distributes work:

- ☐ **State placement**: Determine whether data parallelism shards or replicates the **model state** (weights) across GPUs.
- ☐ **Global batch size**: Calculate the **effective global batch size** for 8 GPUs with a per-GPU batch size of 32.
- ☐ **Single-device equivalence**: Explain why data parallelism is mathematically equivalent to single-device training with a larger batch size.
- ☐ **Binding constraint**: Identify the primary hardware resource that constrains data parallelism as more nodes are added: GPU TFLOP/s or network bandwidth.

The method parallels gradient accumulation, where a single device accumulates gradients over multiple forward passes before updating parameters. Both techniques use the additive properties of gradients to process large batches efficiently. However, moving the same idea into a cluster introduces operational challenges beyond this theoretical equivalence. Communication overhead, node failures, and cost constraints each impose second-order effects that the single-machine derivation does not capture.

5.3.1 Data parallelism implementation

The implementation details matter because the SGD equivalence only holds when each phase preserves disjoint data, complete local gradients, and a single synchronized update. The concrete workflow therefore traces the path from distributing data subsets to synchronizing the computed gradients. Consider figure 5.3: it traces the complete workflow from dataset splitting through gradient aggregation, showing how each GPU processes its assigned batch before synchronization brings all gradients together for parameter updates.

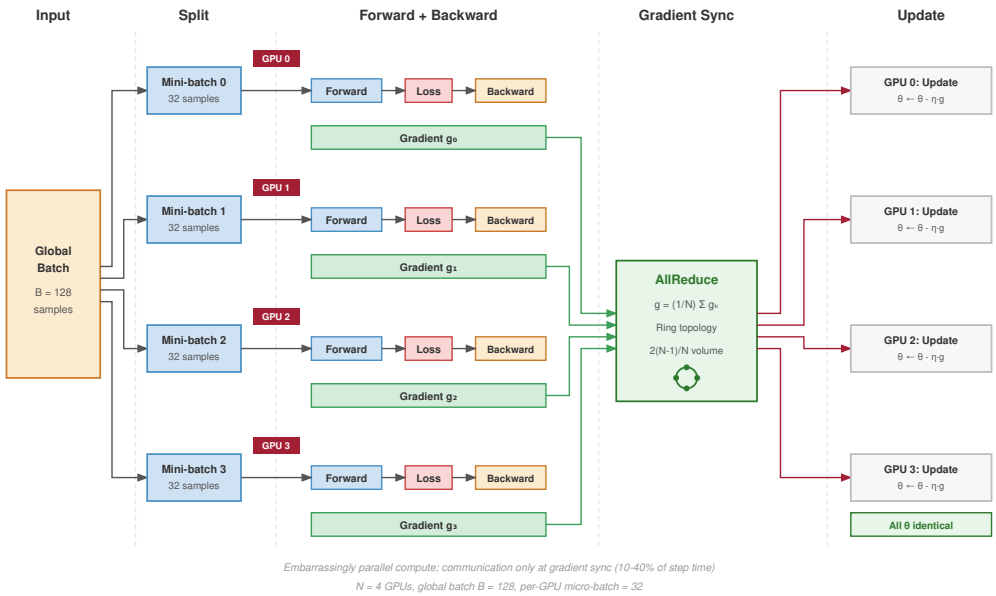


Figure 5.3: Data Parallelism Implementation Pipeline: The five-stage workflow for data parallel training: (1) split input data into nonoverlapping subsets, (2) assign batches to GPUs, (3) compute forward and backward passes independently, (4) synchronize gradients via AllReduce, and (5) update parameters uniformly across all devices. This approach contrasts with model parallelism, where the model itself is partitioned rather than replicated.

As figure 5.3 shows, the critical synchronization point is stage 4: AllReduce must complete before any GPU can update parameters, making gradient communication the dominant bottleneck as the device count grows.

5.3.1.1 Dataset splitting

Data splitting is the first place where the SGD equivalence can fail: each worker must see a unique, deterministic slice of the epoch. With a dataset of 100,000 training examples and 4 GPUs, each GPU receives 25,000 examples per epoch. The DistributedSampler must ensure no overlap between subsets to maintain gradient estimation validity: if two GPUs process the same example, the resulting gradient average would overweight that example, violating the unbiased gradient assumption that makes data parallelism mathematically equivalent to single-device training.

The sampler is therefore part of the training system, not only an input-loader convenience. Modern distributed training frameworks handle this distribution automatically through a distributed sampler that implements prefetching and caching mechanisms to keep accelerators fed without changing sample ownership. The sampler coordinates across workers using the process rank, the integer worker identifier assigned by the distributed runtime, to deterministically partition indices, ensuring reproducibility when the same random seed is used. For a 1.2 million example dataset distributed across 32 GPUs, each GPU processes approximately 37,500 examples per epoch, with the sampler padding the final batch to maintain consistent batch sizes across all workers.

5.3.1.2 Compute phase: Forward and backward passes

The defining feature of data parallelism is that the computation phase, both forward and backward, is *embarrassingly parallel*. Each GPU operates as an isolated island, executing an identical copy of the model on a unique micro-batch of data. Here, micro-batch means the per-GPU local slice used for activation accounting; pipeline microbatching uses the same word for a different mechanism, subdividing a global batch to keep pipeline stages occupied. For our 175B parameter reference model, this isolation is critical: during the forward pass, each GPU independently computes activations for its local batch (micro-batch size 4, sequence length 2048). Without optimization, storing these activations for backpropagation would consume roughly 1.1 TB of HBM, an order of magnitude beyond the capacity of even an H100 GPU. Activation checkpointing, which recomputes activations during the backward pass rather than storing them, becomes necessary in this scenario to suppress the footprint to ~19.3 GB.

The backward pass mirrors this independence but introduces the system's primary bottleneck. As the GPU traverses the computation graph in reverse, it computes gradients for the parameters held by that replica or shard. Under pure data parallelism, a full 175B FP16 replica would imply a 350 GB gradient tensor per worker, which exceeds single-accelerator memory budgets; large runs therefore combine data parallelism with sharding, tensor parallelism, or pipeline parallelism. The computation itself requires zero communication, yet the resulting gradients represent a fractured view of the true loss surface, valid only for the local micro-batch. Before the optimizer step can occur, the corresponding local gradients or gradient shards must be aggregated across data-parallel workers to form a valid global update. The transition from isolated, high-throughput compute to synchronization defines the rhythm of data parallel training: long periods of silent, intense arithmetic punctuated by bursts of heavy network traffic.

5.3.1.3 Gradient synchronization

Gradient synchronization is where independent SGD estimates become one update, so its cost determines whether data parallelism still behaves like a scaling strategy rather than a network benchmark. The immediate training requirement is simple: every data-parallel replica must apply the same averaged gradient before it moves to the next step. AllReduce is the primitive that performs this operation for replicated tensors: each worker contributes its local gradient tensor, the fleet sums the tensors, and every worker receives the same reduced result. In sharded variants, ReduceScatter and AllGather move corresponding pieces rather than the full tensor on every device, but the same synchronization cost remains. Small tensors are dominated by synchronization latency because each communication round has a startup cost; large tensors are dominated by bandwidth because the relevant gradient payload must cross links. Chapter 6 later derives the ring, tree, and hierarchical algorithms that implement this averaging operation on real fabrics.

When synchronization performance deviates from theoretical expectations, the fleet stack framework provides a structured approach to isolating the bottleneck.

Example 5.1: Debugging slow gradient synchronization

Scenario: An AllReduce of a 3 GB gradient tensor across 128 nodes (1,024 GPUs) takes 100 ms. This chapter treats AllReduce as the synchronization primitive created by data parallelism; Chapter 6 derives the ring, tree, and hierarchical algorithms in detail. For now, the useful debugging move is to compare the observation against two coarse bounds: a pessimistic flat communication path and a topology-aware path that separates fast intra-node traffic from slower inter-node traffic.

Context:

- **Topology**: 128 nodes, 8 GPUs per node
- **Fast local path**: NVLink at 300 GB/s bidirectional between GPUs
- **Slower network path**: InfiniBand HDR at 200 Gb/s (25 GB/s) per port

- **Diagnostic bounds**: A flat all-GPU path over the network would land near 239.8 ms; a topology-aware local-then-network path predicts roughly 48.5 ms because each node sends only its share across the slower link

Analysis:

- **Diagnosis**: The collective library is using a hierarchical path that treats the node boundary as the expensive communication boundary
- **Expected behavior**: The inter-node phase should scale with each node's reduced share rather than with the full tensor from every GPU
- **Diagnosis check**: A flat single ring over all 1,024 GPUs would have measured close to 239.8 ms, so the algorithm is already hierarchical

Observation:

- **Observed latency**: 100 ms (well below the 239.8 ms flat-ring upper bound, but about 51.5 ms above the hierarchical ideal)
- **Bandwidth utilization**: Switch counters during the inter-node phase show only 60 percent of theoretical InfiniBand throughput on the affected uplinks (end-to-end efficiency is lower still, since the intra-node phases add time)
- **Network counters**: Show congestion on specific switch uplinks

Diagnosis: NCCL is already using a hierarchical algorithm (100 ms vs. the 239.8 ms flat-ring prediction confirms this). The remaining gap between observed and modeled latency can come from switch congestion on a handful of inter-node uplinks, effective-bandwidth loss, or implementation overhead, not from a fully flat algorithm.

Solution: Monitor InfiniBand switch port utilization to identify hot spots. Consider the rail-optimized topology in section 3.5.3 or further tuning of the hierarchical hand-off so it explicitly partitions intra-node (NVLink) from inter-node (InfiniBand) communication. The residual gap likely represents achievable optimization through better network provisioning and bandwidth utilization rather than a wholesale algorithm change.

Systems lesson: The analysis demonstrates how the fleet stack layers interact: Physical constraints (bandwidth) bound Operational choices (algorithm), which manifest in Service metrics (latency). Debugging requires examining all three layers, not just tuning one in isolation.

Stepping back from that specific cluster to the design space, figure 5.4 contrasts three high-level synchronization topologies: the bandwidth-optimal Ring AllReduce, the centralized Parameter Server, and the fully connected All-to-All mesh. In dense synchronous data-parallel settings, ring AllReduce can avoid a single reducer bottleneck by distributing traffic evenly across participating links, as Baidu's implementation illustrates (Gibiansky 2017).

The trade-off visible in figure 5.4 is between *bandwidth* and *latency*: a ring spreads bandwidth evenly but adds a latency step per hop, while an all-to-all mesh collapses the latency path to constant rounds at the cost of a link count that grows quadratically with the node count. High-performance libraries such as NCCL select among these topologies automatically based on message size and cluster topology. Chapter 6 formalizes the latency complexity of each topology and derives when a ring beats the alternatives.

5.3.1.4 Synchronization models

Distributed training systems operate under explicit synchronization models that govern when workers observe each other's updates. The choice of model determines whether the system guarantees mathematical equivalence to single-device training or trades consistency for throughput. The baseline model, Bulk Synchronous Parallel (BSP)[4] (Valiant 1990), requires all workers to complete their local computation in forward and backward passes, synchronize gradients through a barrier with AllReduce, and then simultaneously update parameters.

4 **Bulk Synchronous Parallel (BSP)**: Introduced by Valiant (1990) as a "bridging model" between hardware and software for parallel computation. BSP divides work into supersteps (compute, communicate, barrier), guaranteeing mathematical equivalence to sequential execution. The cost: iteration time equals the slowest worker's time, and at 1,000 GPUs with 1 percent straggler probability per device, roughly 10 GPUs straggle every step, making the barrier increasingly expensive.

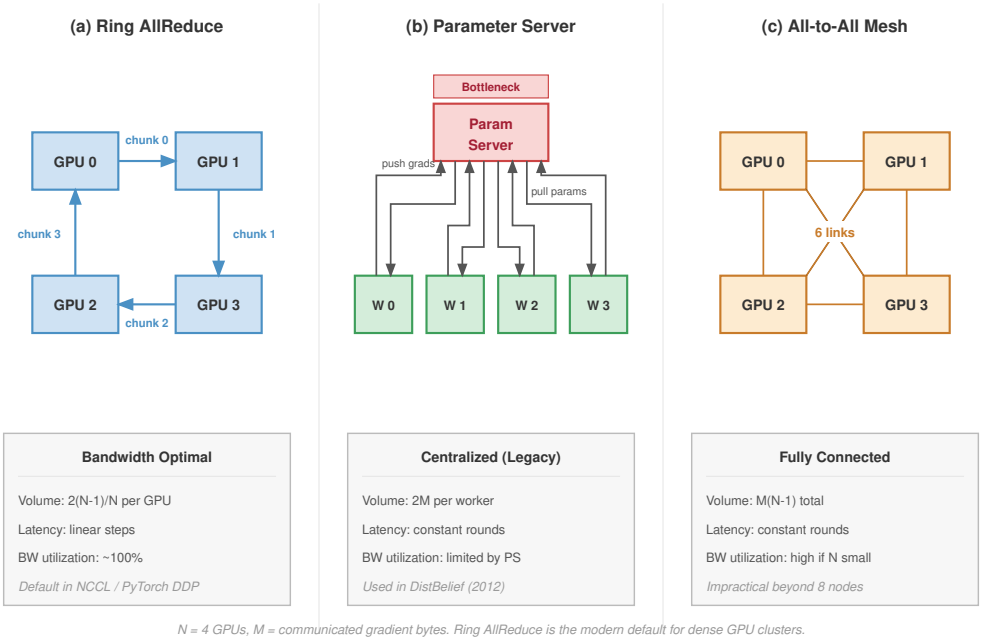


Figure 5.4: Gradient Synchronization Topologies: Visual comparison of communication patterns. (A) Ring AllReduce distributes bandwidth evenly across all links but has linear latency scaling. (B) Parameter Server uses a central node, creating bandwidth bottlenecks at the server. (C) All-to-All mesh connects every node to every other node, achieving constant communication rounds but with a link count that grows quadratically, making it impractical beyond roughly eight nodes.

BSP provides strong guarantees where every worker sees identical parameter values at each step, ensuring mathematical equivalence to single-device training. The cost is that the slowest worker determines iteration time, creating the straggler problem.

Stale Synchronous Parallel (SSP) relaxes this constraint by allowing workers to proceed up to s iterations ahead of the slowest worker before blocking. This bounds staleness while reducing synchronization delays. SSP requires careful learning rate tuning since workers compute gradients on slightly different parameter versions. The bounded staleness guarantee provides a middle ground between BSP's strong consistency and fully asynchronous approaches (Ho et al. 2013).

Asynchronous SGD eliminates synchronization barriers entirely as workers update parameters independently. This maximizes hardware utilization but introduces gradient staleness that can degrade convergence. The operational guarantee determines how much convergence risk the system accepts; section 5.4.4 develops the convergence rates, the staleness penalty, and the compensation techniques each model requires.

The key trade-offs across synchronization models are summarized in table 5.2, and figure 5.5 illustrates how each strategy schedules work across workers over time.

Table 5.2: Synchronization Model Trade-offs: BSP, SSP, and asynchronous SGD trade consistency for throughput along a spectrum. BSP guarantees reproducibility but pays for the slowest worker; SSP unlocks throughput under bounded staleness; async maximizes throughput at the cost of convergence guarantees.

Model	Consistency	Throughput	Convergence	Use Case
BSP	Strong	Bounded by slowest worker	Equivalent to single-GPU	Final training runs, reproducibility
SSP	Bounded staleness	Higher than BSP	Near-equivalent with tuning	Hyperparameter search
Async	Weak	Maximum	Degraded, requires compensation	Large heterogeneous clusters

The same trade-off becomes clearer when the schedules are placed on a timeline.

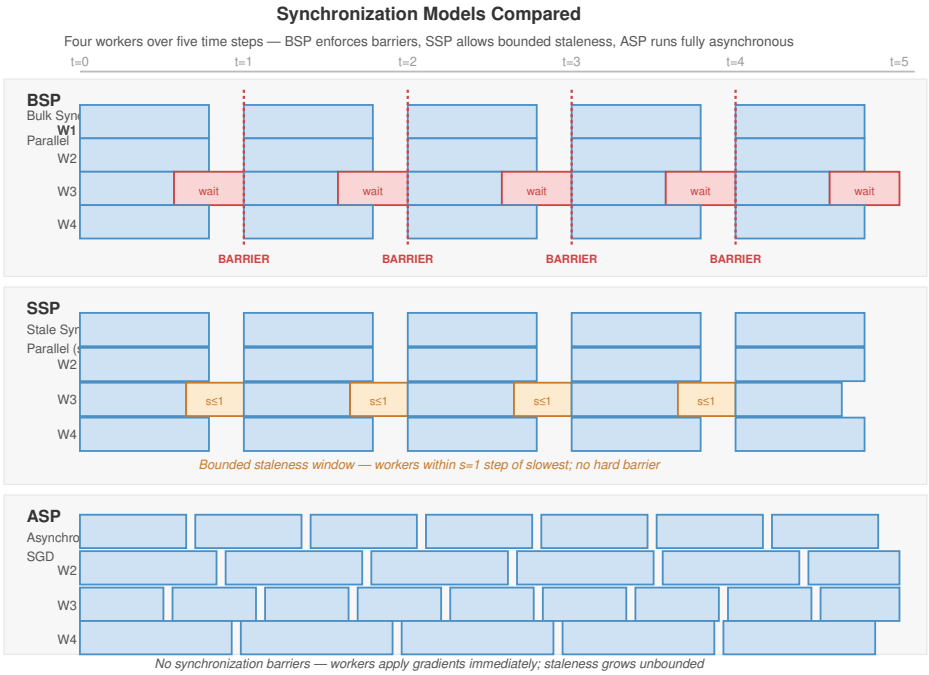


Figure 5.5: Distributed Synchronization Models: Timeline comparison of three synchronization strategies. (A) Bulk Synchronous Parallel (BSP) forces all workers to wait at a global barrier every step. (B) Stale Synchronous Parallel (SSP) allows workers to proceed up to s steps ahead of the slowest worker. (C) Asynchronous SGD eliminates barriers entirely, allowing maximum throughput but introducing gradient staleness.

The choice of synchronization model directly affects both system throughput and model convergence. Training teams often use BSP for final runs to preserve reproducibility, while exploring SSP or async approaches during hyperparameter search where exact reproducibility is less critical.

5.3.1.5 Barrier semantics and failure modes

AllReduce operations implement implicit barriers where no worker can proceed until all workers have contributed their gradients. This coupling creates failure modes absent from single-device training.

Worker failures during AllReduce cause all other workers to block indefinitely while waiting for the missing contribution. Without timeout mechanisms, the entire training job hangs rather than failing cleanly. Deployed systems often implement watchdog timers on the order of minutes to detect and terminate stuck jobs.



One missing worker stalls the entire AllReduce barrier.

Gradient mismatches occur when workers disagree on which tensors to synchronize due to conditional computation paths or dynamic batching. AllReduce operations may block waiting for tensors that some workers never send. This commonly occurs with variable-length sequences in NLP models, dynamic computation graphs, and mixture-of-experts with different routing decisions.

Straggler-induced delays arise because iteration time equals the slowest worker's time plus synchronization overhead. A single slow worker, whether due to thermal throttling, network congestion, or OS jitter, delays all workers and reduces cluster utilization. At 1000 GPUs with 1 percent probability of straggler per GPU per step, approximately 10 GPUs straggle every iteration.

Deployed systems address these issues through timeouts, heartbeat monitoring, and elastic training mechanisms. Chapter 7 provides comprehensive coverage of failure detection, checkpointing strategies, and recovery mechanisms that enable training jobs to complete despite inevitable hardware failures.

5.3.1.6 Parameter updating

Parameter updating closes the data-parallel invariant: after aggregation, every device must apply the same optimizer update from the same gradient values. Each device independently updates model parameters using the chosen optimization algorithm such as SGD with momentum or Adaptive Moment Estimation (Adam). This decentralized update strategy avoids a central coordination server because synchronization has already made the local gradients identical.

In a system with 8 GPUs training a ResNet model, each GPU computes local gradients based on its data subset. After gradient averaging via ring all-reduce (Patarasuk and Yuan 2009), every GPU has the same global gradient values. Each device then independently applies these gradients using the optimizer's update rule. With SGD and learning rate 0.1, the update becomes `weights = weights - 0.1 * gradients`. The example shows why the update can remain decentralized without sacrificing mathematical equivalence to single-device training.

The cycle of splitting data, computing gradients, synchronizing results, and updating parameters repeats for each batch. Frameworks automate this cycle, but they cannot remove the ordering constraint that makes the replicas coherent: every worker must update only after the synchronized gradient is complete.

5.3.2 Trade-offs: The communication wall

Data parallelism is a common starting strategy for a reason: it scales throughput linearly with device count, provided the model fits in memory and communication is not the bottleneck. However, it hits a hard ceiling defined by the communication-computation ratio: once gradient exchange dominates useful computation, more workers mainly add synchronization work (Ben-Nun and Hoefler 2019).

Data parallelism offers three principal advantages. Throughput scales linearly for compute-bound models: scaling ResNet-50 on ImageNet from 1 to 256 GPUs yields near-linear speedup because the gradient exchange is small relative to the compute time. The model architecture also remains unchanged; the framework wraps the model in a data-parallel container that intercepts backward-pass hooks to trigger gradient synchronization automatically. Utilization stays high because, unlike model parallelism, there are no pipeline bubbles: all GPUs work on the forward and backward pass simultaneously.

Three hard ceilings limit these advantages. The memory wall requires every GPU to hold a full copy of the model parameters, gradients, and optimizer states; for a 175B parameter model, this demands more than 1 TB of memory per GPU, exceeding per-device HBM budgets without ZeRO sharding. The bandwidth wall emerges as N grows: the AllReduce cost $\frac{2(N-1)}{N} \times \frac{M}{\text{BW}_{\text{net}}}$ eventually dominates, and for large language models gradient synchronization can consume more than 50 percent of the step time, collapsing efficiency. The batch size trap compounds the problem because scaling to thousands of GPUs requires increasing the global batch size ($B_{\text{global}} = N \times B_{\text{local}}$), and eventually the critical batch size is reached, where adding more data per step yields diminishing returns in convergence.

Napkin Math 5.1: GPT-2 data parallel scaling: single node

A GPT-2 scaling scenario makes the efficiency loss concrete by holding the model fixed and changing only the communication domain. The first case keeps all eight GPUs inside a single NVLink node.

Single GPU Baseline

- Batch size: 16 (with gradient checkpointing, fits in 32 GB)
- Time per step: 1.8 s
- Time to 50K steps: 25 hours

8 GPUs: Single Node with NVLink

- Per-GPU batch: 16, global batch: 128
- Gradient synchronization: 5.2 GB @ 450 GB/s (NVLink, per direction) $\approx$ 11.7 ms

Performance results:

- Compute: 1800 ms per step
- Communication: 11.7 ms per step
- Total: 1811.7 ms per step
- Speedup (throughput): 8×
- Parallel efficiency: 99.4 percent

Training time: 25 hours ÷ 8 = 3.1 hours

Inside the node, NVLink keeps gradient exchange small relative to compute, so efficiency stays near 99.4 percent. The picture inverts once the same model must synchronize across nodes over a commodity network.

Napkin Math 5.2: GPT-2 data parallel scaling: commodity scale-out

The second case scales the same GPT-2 run across multiple nodes, replacing the intra-node NVLink hop with inter-node Ethernet for the bulk of the AllReduce.

Commodity network configuration: 32 GPUs across 4 nodes

- Per-GPU batch: 16, global batch: 512
- Intra-node communication: 11.7 ms (NVLink)
- Inter-node communication: 5.8 GB @ 1.25 GB/s (10GbE) ≈ 4650 ms

Performance results:

- Compute: 1800 ms (27.9 percent of time)
- Communication: 4661.7 ms (72.1 percent of time), so communication dominates and becomes the bottleneck.
- Total: 6461.7 ms per step
- Speedup (throughput): 8.9× faster → 2.8 hours
- Parallel efficiency: 27.9 percent

Gradient accumulation offers a direct remedy by keeping all communication within a single node's NVLink domain while still training on an equivalently large effective batch.

Napkin Math 5.3: Gradient accumulation speedup

Problem: A GPT-2 run on a commodity 10G network is communication-bound at 32 GPUs, costing $3,021 for a fixed number of samples. Can a single 8-GPU node achieve the same effective batch size more efficiently using gradient accumulation?

Math:

1. **Effective batch size**: 8 GPUs × batch 16 × 4 accumulation steps = 512.
2. **Communication overhead**: With 4-step accumulation, we AllReduce once every 4 steps.
   - Overhead = 5.8 ms / (4 × 1800 ms) ≈ 0.081%.
3. **Training duration**: Total time is 3.1 hours.
4. **Total cost**: 3.1 hours × $128/hr = $400.

Systems insight: Gradient accumulation saves $2,621 (86.7 percent) by concentrating computation where bandwidth is abundant (NVLink within the node) and minimizing the frequency of synchronization. When the network is slow, do not scale out—scale the batch size locally.

The calculation changes the scaling decision: when inter-node bandwidth is the binding constraint, gradient accumulation on a bandwidth-rich node can beat naive scale-out even if the wall-clock

run becomes modestly longer. Four insights emerge. NVLink enables efficient scaling within single nodes (99.4 percent efficiency), while inter-node communication kills efficiency (dropping to 27.9 percent). Gradient accumulation beats naive scale-out for communication-bound runs when the scale-out network is slow, so the sweet spot for this GPT-2 scenario is 8 GPUs per node with gradient accumulation, not naive scaling to 32+ GPUs. OpenAI's GPT-2 paper reports training on 32 V100s across 4 nodes using optimized communication (likely gradient accumulation combined with pipeline parallelism), not pure data parallelism.

5.3.3 Memory-efficient data parallelism: ZeRO and FSDP

The memory constraints of data parallelism motivate a family of techniques that shard memory state across workers while preserving the simplicity of data parallel training. ZeRO (Zero Redundancy Optimizer)[5] (Rajbhandari et al. 2020) and its PyTorch implementation FSDP (Fully Sharded Data Parallel) (Zhao et al. 2023) enable training models that would otherwise require model parallelism.

[5] **ZeRO (Zero Redundancy Optimizer)**: Published by Microsoft Research in 2019, ZeRO partitions optimizer states, gradients, and optionally parameters across workers instead of replicating them. At ZeRO Stage 3 with 64 GPUs, per-device memory drops from 16 bytes/parameter (full replication) to 0.25 bytes/parameter, converting a 112 GB memory footprint into 1.75 GB. The trade-off: FSDP (PyTorch's ZeRO-3 implementation) adds AllGather and ReduceScatter on every forward and backward layer, introducing 10–25 percent communication overhead that only pays off when memory pressure justifies it.

Definition 5.3: Sharded data parallelism

Sharded Data Parallelism is the data-parallelism variant (implemented as the ZeRO stages and FSDP) that partitions optimizer state, gradients, and at the deepest stage the parameters themselves across the data-parallel workers, reconstructing each shard on demand through collectives so that per-worker memory falls toward $1/N$ of the full training state while every worker still processes its own minibatch shard.

1. **Significance**: Mixed-precision Adam training carries 16 bytes of state per parameter, so a 7B-parameter model requires 112 GB of training state, out of memory on any 80 GB accelerator even though the model fits comfortably for inference. ZeRO Stage 3 across 64 workers cuts the per-device state to 1.75 GB (0.25 bytes per parameter), converting the memory wall into a communication cost: the AllGather and ReduceScatter traffic that reassembles shards on demand adds 10–25 percent overhead per step.
2. **Distinction**: Unlike vanilla data parallelism (which replicates the complete training state on every worker) and unlike model parallelism (which partitions the computation itself), sharded data parallelism partitions only the storage: every worker still executes the full forward and backward pass, gathering each layer's parameters just in time and discarding them immediately after use.
3. **Common pitfall**: A frequent misconception is that sharding is free memory. The on-demand AllGathers place parameter traffic on the critical path of every layer in every step; on slower interconnects or at small per-worker batch sizes, the exposed communication erodes throughput faster than the memory savings help. Capacity is purchased with bandwidth.

To understand the scale of memory savings ZeRO provides, consider the concrete memory budget for a large language model.

Napkin Math 5.4: ZeRO memory savings

Problem: Training a 7B parameter Llama-2 model in mixed precision requires 16 bytes per parameter for weights, gradients, and optimizer state. Does the full training state fit on a single A100-80 GB, and how does ZeRO Stage 3 with 64 GPUs change the per-device memory requirement?

Baseline: Standard DDP (Replicated State) Per-Parameter Memory Cost:

- **Weights (FP16)**: 2 bytes
- **Gradients (FP16)**: 2 bytes
- **Optimizer state (FP32)**: 12 bytes (4 master weight + 4 momentum + 4 variance)
- **Total**: 16 bytes/parameter

Total Memory for 7B Model:

$$C_{\text{state,total}} = 7 \times 10^9 \times 16 \text{ bytes} \approx 112 \text{ GB}$$

Baseline outcome: out-of-memory (OOM) on A100-80 GB.
Optimization: ZeRO-3 (Fully Sharded) With $N = 64$ GPUs, state is partitioned:

- **Weights**: $2/N$ bytes
- **Gradients**: $2/N$ bytes
- **Optimizer**: $12/N$ bytes
- **Total**: $16/N = 0.25$ bytes/parameter effective storage!

Per-GPU Memory:

$$C_{\text{state,ZeRO3}} = \frac{112 \text{ GB}}{64} \approx 1.75 \text{ GB}$$

Result: Fits easily, leaving ~78 GB for activations (batch size).

ZeRO addresses this redundancy through progressive sharding, as figure 5.6 illustrates and table 5.3 summarizes.

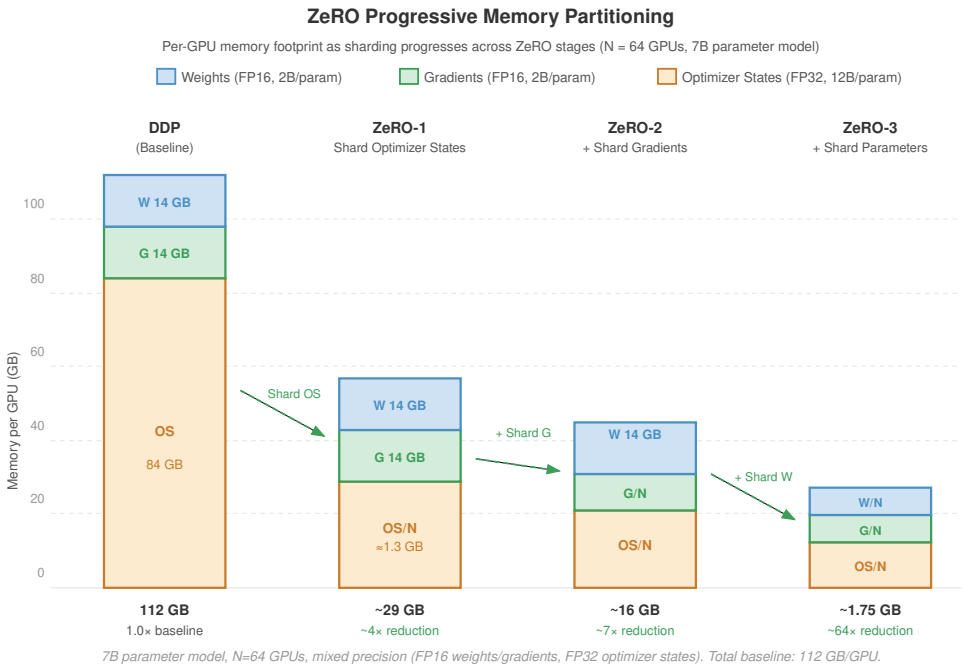


Figure 5.6: ZeRO Memory Reduction: Standard data parallelism (DDP) replicates all model states across every GPU. ZeRO progressively partitions these states: ZeRO-1 shards optimizer states, ZeRO-2 adds gradient sharding, and ZeRO-3 shards the parameters themselves. ZeRO-3 achieves linear memory scaling, enabling models with 100B+ parameters to fit on accelerators that could not hold the replicated state.

Table 5.3: ZeRO Sharding Stages: Each stage shards a different layer of training state and trades a different communication primitive. ZeRO-1 is free relative to DDP; ZeRO-2 swaps AllReduce for ReduceScatter; ZeRO-3 (FSDP) adds an AllGather before every layer but unlocks near-linear memory scaling.

Stage	What is Sharded	Memory Reduction	Communication Overhead
ZeRO-1	Optimizer states only	~4×	None (same as DDP)
ZeRO-2	+ Gradients	~8×	ReduceScatter replaces AllReduce
ZeRO-3/FSDP	+ Parameters	~N (linear in workers)	AllGather before each layer

ZeRO-1 shards optimizer states across GPUs. Each GPU stores only $1/N$ of the Adam optimizer-related state. After gradient AllReduce, each GPU updates only its shard of parameters, then broadcasts updates to other GPUs. Under the 12-byte convention that counts FP32 master weights, momentum, and variance, memory savings reduce optimizer state from $12N$ bytes/param to 12 bytes/param total across the cluster.

ZeRO-2 additionally shards gradients. Instead of AllReduce, which leaves full gradients on each GPU, ZeRO-2 uses ReduceScatter so each GPU receives $1/N$ of the reduced gradients. Under the FP16-gradient convention used here, memory savings reduce gradients from $2N$ bytes/param replicated across N workers to 2 bytes/param total, or $2/N$ bytes/param per GPU.

ZeRO-3 and FSDP shard parameters themselves. Each GPU stores only $1/N$ of the model. Before each layer's forward pass, parameters are gathered via AllGather; after backward pass, gradients are reduced via ReduceScatter, then parameters are discarded. This achieves maximum memory efficiency at the cost of additional communication that FSDP introduces relative to standard DDP.

This sharding places communication on the critical path that DDP avoids. The forward pass needs an AllGather to reconstruct each layer's parameters (M_{layer} bytes); the backward pass needs a second AllGather to reconstruct them when parameters are resharded after the forward pass (M_{layer} bytes), followed by a ReduceScatter for gradients (M_{layer} bytes). For a model with N_L layers, full-shard FSDP with resharding therefore performs about $3N_L$ collective operations per training step against the single AllReduce that DDP needs, raising total communication volume to roughly $3M_{\text{state}}$ bytes versus $2M_{\text{state}}$ for DDP. The collectives are spread across more operations with overlap opportunities, however: while layer i computes, layer $i+1$ can prefetch its parameters.

The choice between FSDP and DDP depends on model size and memory constraints. When the model fits in GPU memory with room for activations, DDP usually wins because it avoids the repeated AllGather work. As memory pressure rises, ZeRO-2 becomes attractive because it shards gradients and optimizer state while leaving parameters replicated; once parameters themselves exceed single-GPU memory, ZeRO-3/FSDP becomes necessary even though it puts AllGather on the critical path. For training 70B+ models on 80 GB, FSDP typically has to combine with tensor parallelism rather than replace it.

Memory-efficient data parallelism requires careful tuning of sharding strategy (by layer, by transformer block, or flat) and mixed precision settings. The sharding granularity determines the trade-off: finer sharding reduces per-GPU memory but increases communication frequency as more AllGather and ReduceScatter operations must execute per training step.

Eliminating memory as the bottleneck through ZeRO and FSDP makes it tempting to scale data parallelism to hundreds of GPUs. Doing so, however, changes the optimization landscape in ways the communication analysis alone does not predict. Large global batch sizes alter gradient noise statistics, and learning-rate schedules tuned for eight-GPU runs can diverge catastrophically at 256 GPUs. A landmark large-scale demonstration of this failure mode, and an engineering response that became widely influential, came from a single experiment.

War Story 5.1: The one-hour ImageNet run (2017)

Context: Facebook AI Research set out to train ResNet-50 on ImageNet in one hour using 256 GPUs, pushing data parallelism far beyond the batch sizes that earlier practice treated as safe (Goyal et al. 2017).

Failure mode: Naively increasing the global batch size destabilized optimization. The communication system could supply more throughput, but the optimizer no longer behaved like the single-machine run.

Consequence: The team recovered accuracy by pairing the linear scaling rule with gradual learning-rate warmup, stabilizing a global batch size of 8,192 while preserving convergence.

Systems lesson: Distributed training is not just parallel hardware. Scaling changes the optimization regime, so the cluster, communication schedule, batch size, and learning-rate schedule must be tuned as one system.

5.4 Scaling Efficiency and Convergence

When doubling the number of GPUs yields only 1.5× speedup, communication overhead and synchronization barriers have consumed the missing 25 percent of compute budget. Data parallelism revealed the practical mechanics of gradient synchronization and memory sharding, but understanding *why* scaling efficiency degrades and *how* convergence changes with parallelism requires a quantitative framework. The metrics and convergence theory in this section apply to all parallelism strategies—data, model, pipeline, and hybrid—governing the fundamental trade-offs between throughput, communication cost, and optimization quality.

5.4.1 The mathematics of scaling efficiency

Before examining specific parallelism strategies, we must understand the metric that determines whether scaling from one device to many is worthwhile: scaling efficiency. If a model trains in time T_1 on one device, ideal (linear) scaling would train it in time T_1/N on N devices. In practice, communication overhead, pipeline bubbles, and load imbalances reduce the speedup. Scaling efficiency is defined by equation 5.1:

$$\eta_{\text{scaling}} = \frac{T_1}{N \times T_N} \tag{5.1}$$

where T_N is the training time on N devices. An efficiency of 1.0 means perfect linear scaling; an efficiency of 0.5 means we achieve only half the expected speedup.

Definition 5.4: Scaling efficiency

Scaling Efficiency $(\eta_{scaling})$ is the ratio of actual training throughput to ideal linear throughput when increasing the number of ML compute devices (N).

1. **Significance**: It is the most important metric for cluster productivity ($\eta_{\text{scaling}} = \frac{T_1}{N \times T_N}$). A scaling efficiency of 0.50 means that a 10,000-GPU cluster is delivering only the same useful work as a 5,000-GPU cluster, wasting 50 percent of the hardware investment.
2. **Distinction**: Unlike single-node efficiency (which captures local bottlenecks like BW), scaling efficiency captures the cluster-level overhead of communication time $(T_{\text{comm}}(N))$ and synchronization.
3. **Common pitfall**: A frequent misconception is that scaling efficiency is constant. In reality, it is a function of problem size: as N increases, the communication-to-compute ratio typically worsens (Amdahl's Law), making it harder to maintain high efficiency for small models.

For data-parallel training of our 175B model, the communication cost per step is dominated by the AllReduce of 350 GB of gradients.

Using ring-AllReduce over InfiniBand at 50 GB/s effective bandwidth, the raw communication time is approximately $2 \times (N-1)/N \times 350/50$, which for large N approaches $2 \times 350/50$ seconds. With 75 percent overlap between gradient communication and the backward pass, the effective exposed communication time drops to $T_{\text{comm}}(N) \approx 3.5$ seconds. Under the same assumptions, the compute term is $T_{\text{compute}}/N \approx 2.1$ seconds, so the exposed step time is about 5.6 s and naive

data-parallel scaling efficiency is 2.1 s/5.6 s ≈ 37.5 percent. Well-configured systems can recover much of this loss by combining tensor parallelism, pipeline parallelism, topology-aware placement, and more effective communication overlap.

The scaling efficiency depends critically on the ratio of computation to communication. Three factors govern this ratio. Larger models have more computation per training step (more FLOPs per weight update), so the same communication overhead represents a smaller fraction of total step time; this *model-communication ratio* is why scaling efficiency improves as models grow larger. Larger batches push in the same direction, raising the computation per step without proportionally raising communication, because gradients are the same size regardless of batch size; the catch is that large batches can harm convergence and so require learning rate tuning and warmup schedules. Network bandwidth enters most directly, since doubling the InfiniBand bandwidth halves the communication time and improves scaling efficiency in proportion. The network fabric is therefore a first-order determinant of cluster productivity, not a secondary concern; its cost (10–15 percent of total system cost) is easily justified if it lifts scaling efficiency by even a few percentage points, because poor scaling efficiency wastes the other 85–90 percent of the investment.

Paradoxically, larger models are easier to scale than smaller ones.

These three factors interact in important ways. Larger models with larger batch sizes achieve better scaling efficiency, which means that the most expensive training workloads are also the ones that benefit most from scale. This creates a virtuous cycle for large-scale infrastructure: the workloads that justify building thousand-GPU clusters are also the workloads that use them most efficiently. Conversely, small models and small batch sizes scale poorly, which is why teams training 1B-parameter models on 64 GPUs often achieve only 40–60 percent scaling efficiency.

However, even for large models, scaling does not continue indefinitely. There exists a **scaling cliff** beyond which adding more GPUs actually reduces cost-efficiency. Under the assumptions of our 175B model, the cost-efficient region is approximately 1,024–4,096 GPUs, where the communication-to-compute ratio remains favorable and scaling efficiency stays above 70 percent. Beyond 8,192 GPUs, the AllReduce communication time begins to dominate the backward pass computation time, and the efficiency drops below 50 percent. While the wall-clock training time may still decrease slightly with more GPUs, the *cost per useful FLOP* increases because the organization is paying for 8,000 GPUs to do the work of 4,000. The nonlinear relationship dictates that the economic viability of training large models is bounded by the physics of interconnect latency, not merely by hardware availability. The cluster must be sized to operate in the linear regime of the scaling curve, and the model architecture (batch size, sequence length, parallelism dimensions) must be co-designed with the cluster size to maintain this balance.



Napkin Math 5.5: Scaling efficiency for a 175B model

Setup: Training a 175B model on a DGX H100 cluster with 400 Gb/s InfiniBand per GPU.

- **Compute per step** (assuming batch size 2M tokens, 6 FLOPs per parameter per token): $O_{\text{step}} = 6 \times 175 \times 10^9 \times 2 \times 10^6 \approx 2.1 \times 10^{18}$ FLOPs
- **Per-GPU compute time** on 1,024 GPUs, each at 1979 TFLOP/s FP8 peak (50 percent utilization): $T_{\text{compute}}/N \approx 2.1$ seconds
- **AllReduce time** for 350 GB of gradients using ring-AllReduce with overlap: $T_{\text{comm}}(N) \approx 3.5$ seconds (raw transfer is about 14 seconds; this example assumes 75 percent overlap with the backward pass)
- **Scaling efficiency**: $\eta_{\text{scaling}} \approx \frac{T_{\text{compute}}/N}{T_{\text{compute}}/N + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}} = 2.1/5.6 \approx 0.375$ in this simplified example, where synchronization is included in the AllReduce term and 75 percent communication overlap has already been applied.

The low efficiency (37.5 percent) shows why naive data parallelism at this scale is insufficient. Hierarchy-aware systems recover much of that lost efficiency by combining data parallelism with tensor parallelism, which communicates over NVLink, and pipeline parallelism, which overlaps computation with communication.

5.4.1.1 Parallelism-infrastructure interaction

The scaling efficiency analysis reveals a deeper insight: the effective parallelism strategy is not determined by the *model architecture alone* but by the *interaction* between the model's communication requirements and the infrastructure's bandwidth hierarchy. Each combination of parallelism strategy and infrastructure topology produces a different scaling efficiency curve, and selecting the wrong combination can waste a significant fraction of the cluster's capacity. The napkin math above already showed pure data parallelism stalling near 37.5 percent efficiency at this scale; recovering that lost capacity means mapping each parallelism dimension onto the bandwidth tier that can carry its traffic. Section 5.7 works that combination through three concrete configurations and formalizes it as hierarchy-aware parallelism, once tensor and pipeline parallelism have been developed.

Selecting and combining parallelism strategies therefore leads directly to the traffic they create. Chapter 3 examined how topology is co-designed with the parallelism mapping to maximize scaling efficiency; here the next question is how the collective operations themselves shape the step time. AllReduce operations can consume 10–40 percent of total training time in data parallel systems, and this overhead grows with cluster size. BERT-Large on 128 GPUs can experience communication overhead reaching a large fraction of total runtime, while GPT-3-scale models require tensor, pipeline, and data parallelism plus communication overlap to avoid data-parallel gradient synchronization dominating the step.

AllReduce complexity depends on two components: latency (α) and bandwidth (β). Ring AllReduce achieves bandwidth-efficient communication with $(N-1)/N$ utilization, while tree-based approaches offer lower latency at $\mathcal{O}(\log N)$ steps. The choice depends on message size: tree algorithms win for latency-dominated small messages, ring algorithms win for bandwidth-dominated large gradients. High-performance implementations such as NCCL use hierarchical algorithms that combine tree latency within nodes and ring bandwidth between nodes. Chapter 6 provides detailed algorithm analysis, including complexity formulas, hierarchical variants, and topology-aware optimizations for large-scale collective operations.

Interconnect selection determines whether large-scale deployments remain compute-bound or collapse into communication-bound regimes, and the bandwidth requirements for efficient distributed training are substantial, particularly for transformer models. Efficient systems often require 100–400 GB/s aggregate bandwidth per node for transformer architectures. BERT-Base (110M parameters) requires approximately 440 MB of gradient synchronization per iteration in FP32, while BERT-Large (340M parameters) requires approximately 1.4 GB. Across 64 GPUs, these synchronization demands require 100–200 GB/s sustained bandwidth for sub-50 ms synchronization latency. For 175B-parameter language models, exact bandwidth requirements depend on the 3D-parallel configuration, gradient accumulation, overlap, and interconnect topology rather than a single universal number.

Synchronization frequency presents a trade-off between communication efficiency and convergence behavior. Accumulating gradients for 4 microsteps reduces synchronization frequency by 75 percent, but the realized step-time reduction depends on the compute/communication mix and how much communication can overlap with backpropagation. In standard implementations, gradient accumulation reuses one resident gradient buffer and accumulates in place; memory pressure comes from keeping that buffer resident and from any larger microbatch or activation choices, not from storing 4 independent gradient tensors. Asynchronous methods eliminate synchronization costs entirely but introduce staleness that degrades convergence by 15–30 percent for large learning rates.

5.4.2 The physics of scaling: Amdahl's Law with communication

Just as the Iron Law of Processor Performance governs single-thread execution, distributed training is governed by an extended version of Amdahl's Law that explicitly accounts for communication overhead. The time to complete one training step on N devices is not simply T_{single}/N, but is constrained by the sequential nature of synchronization. Section C.3.2 derives the speedup ceiling at fleet scale and works it through a concrete example; here we establish the idea that a fixed synchronization fraction caps speedup no matter how many accelerators are added.

Figure 5.7 visualizes this divergence from ideal linear scaling as the **Scaling Tax**—a direct consequence of the scaling efficiency bound (principle 8). It shows how communication overhead (r)

acts as a drag on performance, creating a communication wall where adding more GPUs yields diminishing returns.

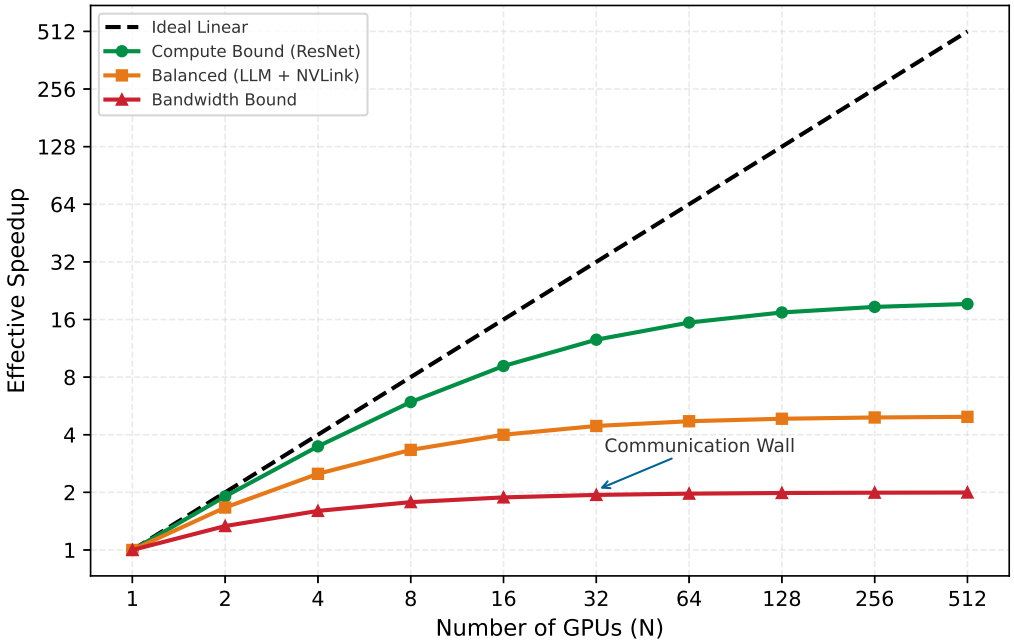


Figure 5.7: The Scaling Tax: Effective speedup vs. number of GPUs. While ideal scaling (dashed black) is linear, real-world systems pay a 'tax' for communication. Compute-bound models like ResNet (green) scale well because they have high arithmetic intensity. Bandwidth-bound models like GPT-3 (red) hit a communication wall where adding more GPUs yields diminishing returns (efficiency < 50 percent).

The fleet law (principle 10) maps directly onto the iron law's variables, separating compute, bandwidth, and coordination into additive terms:

$$T_{\text{step}}(N) = \underbrace{\frac{T_{\text{compute}}}{N}}_{\text{Compute-Time Term}} + \underbrace{T_{\text{comm}}(N)}_{\text{Bandwidth Term}} + \underbrace{T_{\text{sync}}(N)}_{\text{Coordination Term}} - T_{\text{overlap}}$$

The fleet-law terms separate step time into four distinct costs:

- **Compute-Time Term** (T_{compute}/N): The total computation required for the batch after ideal partitioning across N devices.
- **Bandwidth Term** $(T_{\text{comm}}(N))$: The time spent moving data. This is governed by the iron law's data-movement term, D_{vol}/BW. For Ring AllReduce, this term is $\frac{2(N-1)}{N} \times \frac{M}{\text{BW}_{\text{net}}}$, where M is the communicated gradient or model-state bytes and BW_{net} is network bandwidth.
- **Coordination Term** $(T_{\text{sync}}(N))$: The nonoverlapped cost of barriers, ordering, and straggler waiting.
- **Overlap** (T_{overlap}): The portion of communication hidden behind computation.

The fleet law leads to the scaling efficiency metric for fixed global work, where T_{compute} is the single-device compute time for that work:

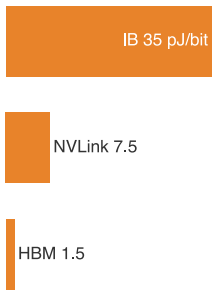
$$\eta_{\text{scaling}} = \frac{T_{\text{compute}}}{N \times T_{\text{step}}(N)} = \frac{1}{1 + \frac{N(T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}})}{T_{\text{compute}}}}$$

This is the scaling efficiency bound (principle 8): perfect linear scaling $(\eta_{\text{scaling}} = 1.0)$ is a *theoretical limit*, not a *practical target*. Well-configured systems can achieve $\eta_{\text{scaling}} = 0.85$–$0.95$ at moderate scale

and degrade further as N grows. The gap between $\eta_{\text{scaling}} = 1.0$ and the achieved efficiency is the communication tax and coordination tax: the price of distributed execution.

The equation reveals the **scaling wall**: as N increases, the compute term (T_{compute}/N) shrinks, but the communication and synchronization terms can remain constant or grow. Eventually, the denominator is dominated by overhead, driving efficiency toward zero. Beyond wall-clock time, this communication overhead imposes an energy tax that scales with physical distance between devices.

> **Systems Perspective 5.2: The energy tax of scale**
>
> Distributed training is a race against energy as much as against time. In a single GPU, moving a byte from HBM to the cores costs roughly 1–2 pJ/bit. Moving that same byte across an NVLink interconnect costs 5–10 pJ/bit. Moving it across an InfiniBand network through switches costs 20–50 pJ/bit.
>
> At the scale of 10,000 GPUs, the multiplication by aggregate bandwidth is what changes the engineering problem. A full H100 NVLink envelope is roughly 9 PB/s across the cluster; at 7.5 pJ/bit, that movement represents about 540 kW before cooling and switch overhead. The aggregate NDR InfiniBand envelope is smaller (500 TB/s), but at 35 pJ/bit it still represents about 140 kW. Communication-computation overlap is therefore necessary for wall-clock efficiency, but avoiding unnecessary movement is the direct way to reduce the power term itself.



Communication energy climbs from HBM out to the network.

Wall-clock efficiency and the energy tax degrade together as N grows, following predictable patterns across GPU counts. As a representative rule of thumb, systems in the linear scaling regime of 2–32 GPUs often achieve 85–95 percent parallel efficiency because communication overhead remains small. The communication-bound regime emerges at 64–256 GPUs, where efficiency can drop to 60–80 percent even with well-matched interconnects. Beyond 512 GPUs, coordination overhead can become dominant and limit efficiency to 40–60 percent due to collective operation latency.

Hardware selection critically impacts these scaling characteristics. NVIDIA DGX A100 systems provide 600 GB/s of bidirectional NVLink bandwidth per GPU, with aggregate NVSwitch bandwidth at the system level, enabling high parallel efficiency inside an 8-GPU node. Multi-node scaling requires a network fabric with enough bisection bandwidth; EDR-class 100 Gbps links can support smaller multi-node jobs, while HDR-class 200 Gbps links can support larger clusters when topology, placement, and overlap are well matched.

The efficiency metrics directly influence the choice of parallelism strategy. Data parallelism works well in the linear scaling regime but becomes communication bound at scale. Model parallelism addresses memory constraints but introduces sequential dependencies that limit efficiency. Pipeline parallelism reduces device idle time but introduces complexity in managing microbatches. The effective strategy depends on which constraint—memory, bandwidth, or synchronization—dominates the target workload.

Hardware efficiency metrics govern throughput, but convergence theory determines whether distributed training reaches the same solution quality as single-device training. Parallelism affects optimization convergence in three ways: convergence rate changes with batch size, adding workers yields diminishing returns beyond the critical batch size, and learning rates must scale with batch size.

5.4.3 Convergence rate for synchronous data parallel SGD

The fundamental convergence result for distributed SGD explains why adding workers can reduce iteration count without changing the optimizer's meaning. The theorem uses two standard regularity assumptions: the loss is smooth enough that gradients cannot change arbitrarily fast, and each stochastic gradient has bounded variance around the true gradient. Written formally, for a loss function $\mathcal{L}(\theta)$ with L_s-Lipschitz gradients and variance-bounded stochastic gradients $\mathbb{E}[\|g_i - \nabla\mathcal{L}(\theta)\|^2] \leq \sigma^2$, synchronous data parallel SGD with N workers achieves the following convergence rate.

Theorem 5.1: Convergence rate for distributed SGD

For synchronous data parallel SGD with N workers, each computing gradients on a local batch of size b, after K total iterations the expected optimization error satisfies:

$$\mathbb{E}[\mathcal{L}(\theta_K)] - \mathcal{L}^\star \leq \underbrace{\frac{L_s \|\theta_0 - \theta^\star\|^2}{2K}}_{\text{optimization error}} + \underbrace{\frac{\eta L_s \sigma^2}{2Nb}}_{\text{variance floor}}$$

where η is the learning rate, L_s is the smoothness constant, $\mathcal{L}^\star$ is the optimal loss value, $\theta^\star$ is an optimal parameter vector, and σ^2 is the gradient variance. The effective convergence rate is $\mathcal{O}(1/\sqrt{NbK})$ when the learning rate is tuned optimally as $\eta = \mathcal{O}(\sqrt{Nb/K})$.

The theorem reveals several important insights. First, the variance floor decreases linearly with the number of workers N, explaining why distributed training can achieve the same final loss with fewer iterations in the linear scaling regime. With equal local batch size b, synchronous data-parallel SGD computes the same minibatch gradient as a single worker using batch size Nb for that step; convergence speed still depends on learning-rate scaling and critical-batch-size effects. Second, the convergence rate $\mathcal{O}(1/\sqrt{NbK})$ shows that workers, local batch size, and iteration count jointly determine statistical progress, assuming infinite bandwidth. This is the **statistical efficiency** of distributed training, distinct from hardware efficiency.

However, the theorem assumes perfect synchronization (BSP). When workers proceed at different rates or use stale gradients, convergence guarantees degrade, as we examine next.

5.4.4 Staleness impact: BSP vs. SSP vs. ASP

The operational guarantees of BSP, SSP, and ASP established in section 5.3.1.4 have a convergence cost, which this section sketches with stylized smooth-objective bounds. The staleness parameter τ_{stale} quantifies how many iterations behind a gradient may be when applied to parameters, and it is the variable that ties throughput to solution quality (Ho et al. 2013; Dutta et al. 2018).

Definition 5.5: Gradient staleness

Gradient Staleness (τ_{stale}) is the number of parameter updates that occur between the time a gradient is computed and the time it is applied to the global model state.

1. **Significance**: It represents the synchronization error in distributed optimization. Increasing τ_{stale} can improve throughput by reducing barrier waits $(T_{\text{sync}}(N))$, but it typically degrades the rate of convergence, requiring more operations (O) to reach the same accuracy.
2. **Distinction**: Unlike network latency, which is a physical delay, staleness is an algorithmic offset that arises from the choice of synchronization protocol (e.g., ASP, SSP).
3. **Common pitfall**: A frequent misconception is that staleness is "always bad." In reality, it is a throughput-convergence trade-off: for some large-scale workloads, allowing bounded staleness is one practical way to keep thousands of GPUs in use.

The convergence behavior differs across these models in ways that directly affect training cost and solution quality. In Bulk Synchronous Parallel (BSP, $\tau_{\text{stale}} = 0$), all workers compute gradients on the same parameter version, then synchronize via barrier before updating. This guarantees mathematical equivalence to single-device training with larger batch size, an optimal convergence rate of $\mathcal{O}(1/\sqrt{NbK})$, and no hyperparameter adjustment beyond batch size scaling.

Stale Synchronous Parallel (SSP, $\tau_{\text{stale}} \leq s$) relaxes the barrier by allowing workers to proceed up to s iterations ahead of the slowest worker. In a stylized bound, the convergence rate gains an additive delay term:

$$\mathbb{E}[\mathcal{L}(\theta_K)] - \mathcal{L}^\star \leq \mathcal{O}\left(\frac{1}{\sqrt{NbK}}\right) + \mathcal{O}\left(\frac{s^2\eta^2 L_s^2}{Nb}\right)$$

The second term represents the **staleness penalty**. For bounded staleness s, this penalty is controlled by keeping delay small and retuning the learning rate. Deployments that choose bounded staleness accept convergence degradation for throughput improvement on heterogeneous clusters.

Asynchronous SGD (ASP, $\tau_{\text{stale}} = \infty$) eliminates waiting entirely: workers update parameters immediately. While this maximizes throughput, the same teaching model shows a larger delay-dependent term:

$$\mathbb{E}[\mathcal{L}(\theta_K)] - \mathcal{L}^\star \leq \mathcal{O}\left(\frac{1}{\sqrt{K}}\right) + \mathcal{O}\left(\bar{\tau}_{\text{stale}}^2 \eta^2 L_s^2\right)$$

where $\bar{\tau}_{\text{stale}}$ is the average staleness. The staleness penalty now scales with the square of average delay, and critically, the variance reduction from N workers disappears in the dominant term. Several techniques compensate for this. Learning rate decay ($\eta' = \eta/\sqrt{1+\bar{\tau}_{\text{stale}}}$) reduces the staleness penalty but slows convergence. Momentum correction adjusts the momentum term to account for delayed updates. Gradient clipping prevents stale gradients with large magnitudes from destabilizing training.

Table 5.4 summarizes the convergence properties of each synchronization model.

Table 5.4: Convergence Properties by Synchronization Model: BSP provides optimal convergence guarantees at the cost of synchronization overhead. SSP offers a tunable trade-off between throughput and convergence. ASP maximizes throughput but loses the variance reduction benefit of parallelism.

Model	Staleness	Convergence Rate	Variance Reduction	Best For
BSP	$\tau_{\text{stale}} = 0$	$\mathcal{O}(1/\sqrt{NbK})$	Full ($1/N$)	Final training, reproducibility
SSP	$\tau_{\text{stale}} \leq s$	$\mathcal{O}(1/\sqrt{NbK}) + \mathcal{O}(s^2\eta^2 L_s^2/Nb)$	Partial	Heterogeneous clusters
ASP	$\tau_{\text{stale}} = \infty$	$\mathcal{O}(1/\sqrt{K}) + \mathcal{O}(\bar{\tau}_{\text{stale}}^2 \eta^2 L_s^2)$	None	Maximum throughput, early exploration

5.4.5 Learning rate scaling rules

When increasing the effective batch size through data parallelism, the learning rate must be adjusted to maintain convergence quality. Two primary scaling rules have emerged from both theory and practice.

The linear scaling rule (Goyal et al. 2017) states that when the batch size is multiplied by k, the learning rate should also be multiplied by k:

$$\eta_{\text{large}} = k \cdot \eta_{\text{base}}$$

This rule is justified by approximating one large-batch step at learning rate $k\eta$ as k consecutive small-batch steps each at learning rate η. With batch size kB, the gradient variance decreases by factor k, so the larger step is well-conditioned; the approximation holds when parameters do not move much during those k small-batch steps. The rule is empirical rather than universal: Goyal et al. validated it for ResNet-50/ImageNet at global batch size 8,192 with gradual warmup, while LARS and LAMB extend large-batch recipes through layer-wise adaptation for ImageNet and BERT-style pretraining (Goyal et al. 2017; You et al. 2017; You et al. 2020). A **warmup period** that increases the learning rate linearly from η_{base} to $k \cdot \eta_{\text{base}}$ over the first W iterations lets the model reach a region of the loss landscape where large learning rates are stable:

$$\eta_t = \eta_{\text{base}} + \frac{t}{W}(k \cdot \eta_{\text{base}} - \eta_{\text{base}}) \quad \text{for } t < W$$

The square root scaling rule applies when batch sizes grow so large that linear scaling fails:

$$\eta_{\text{large}} = \sqrt{k} \cdot \eta_{\text{base}}$$

The more conservative square root rule is motivated by the observation that gradient noise (not just magnitude) affects optimization dynamics. The square root rule better preserves the signal-to-noise ratio of gradient updates. Empirically, square root scaling becomes necessary when batch sizes exceed the critical batch-size regime.

For extreme batch sizes (32K–1M), layer-wise adaptive learning rate scaling (LARS) (You et al. 2017) and its Adam variant LAMB (You et al. 2020) automatically adjust learning rates per layer based on the ratio of weight norm to gradient norm:

$$\eta_\ell = \eta_{\text{global}} \cdot \frac{\|w_\ell\|}{\|g_\ell\| + \lambda_{\text{wd}} \|w_\ell\|}$$

Here λ_{wd} is the optimizer's weight-decay coefficient in the LARS/LAMB update, avoiding collision with the failure-rate parameter used in reliability analysis. Layer-wise scaling prevents layers with small weights from receiving disproportionately large updates. LAMB enabled BERT training with batch sizes up to 64K while maintaining convergence quality.

5.4.6 Critical batch size and diminishing returns

A fundamental property of distributed training is that adding more workers stops helping past a threshold. The critical batch size B^* marks the transition point beyond which increasing batch size yields diminishing returns in convergence per sample seen.

Definition 5.6: Critical batch size

Critical Batch Size (B^*) is the distributed-training batch size at which the gradient noise scale is large enough that further batch growth yields diminishing returns in sample efficiency (McCandlish et al. 2018; Shallue et al. 2019).

1. **Significance**: It marks the transition point for parallel scaling efficiency. Below B^*, increasing the batch size linearly improves the convergence per step. Above B^*, larger batches yield diminishing returns, requiring proportionally more samples (D) to reach the same loss.
2. **Distinction**: Unlike the memory-limited batch size (determined by memory capacity and activation footprint), the critical batch size is an algorithmic property of the model and dataset.
3. **Common pitfall**: A frequent misconception is that training can be sped up indefinitely by adding GPUs. In reality, B^* defines the physical ceiling for data parallelism: adding workers beyond this point wastes energy and compute (O) without reducing total training time (T).

The gradient-noise-scale proxy for the critical batch size can be estimated as:

$$B^* \approx \frac{\text{tr}(\Sigma)}{\|\nabla \mathcal{L}(\theta)\|^2}$$

where $\text{tr}(\Sigma)$ is the trace of the gradient covariance matrix (total gradient variance) and $\|\nabla \mathcal{L}(\theta)\|^2$ is the squared gradient norm (signal strength). Intuitively, B^* is the batch size at which averaging reduces gradient variance to the level of the true gradient magnitude.

Published large-batch regimes illustrate the scale rather than providing fixed thresholds. On ImageNet, Goyal et al. stabilized ResNet-50 at global batch 8,192 with warmup, and LARS extends ImageNet large-batch training further in some settings (Goyal et al. 2017; You et al. 2017). For BERT-Large pretraining, LAMB reports training at 32K–64K batch sizes with layer-wise adaptive updates (You et al. 2020). Across other domains, McCandlish et al. show useful batch-size limits ranging from tens of thousands in ImageNet-like settings to millions in reinforcement-learning workloads, so new frontier-language-model thresholds should be measured rather than copied (McCandlish et al. 2018).

The scaling law regime exhibits three distinct behaviors. Below the critical batch size $(B < B^*)$, linear scaling holds: doubling the batch size halves the iterations needed to reach the target loss,

and hardware efficiency determines throughput. At the critical point ($B \approx B^*$), samples-per-second efficiency is maximized, the optimal trade-off. Above it ($B > B^*$), returns diminish, because doubling the batch size requires more than double the total samples, so additional workers add throughput but not sample efficiency.

Figure 5.8 illustrates this relationship between batch size and training efficiency.

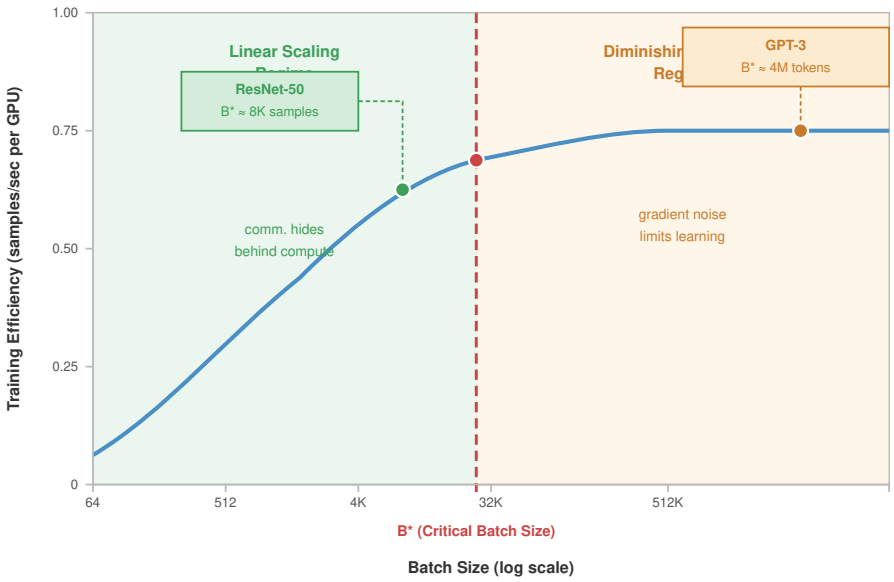


Figure 5.8: Critical Batch Size and Scaling Regimes: Below the critical batch size B^*, larger batches reduce noise and improve sample efficiency (linear scaling regime). Above B^*, larger batches provide diminishing returns: while throughput increases, total samples required also increases, reducing sample efficiency. The optimal operating point balances hardware utilization against convergence efficiency.

The critical batch size has important implications for distributed training system design. Adding workers beyond B^*/b (where b is the per-worker batch size) improves throughput but not sample efficiency, though it may still be worthwhile for cost if the marginal cost of additional workers is low. The learning rate schedule matters too, because above B^* aggressive warmup becomes essential, since the loss landscape near initialization may not support the large updates that linear scaling would produce. Communication trade-offs shift as well, because above B^* the reduced benefit of larger batches makes communication overhead relatively more costly, strengthening the case for gradient compression or asynchronous methods.

Checkpoint 5.2: Scaling decisions

Given a 7B parameter model distributed across a cluster of 64 A100 GPUs (80 GB HBM2e each), determine the maximum useful batch size. Use a pilot trace with $\text{tr}(\Sigma) = 1.6 \times 10^6$, $\|\mu\|^2 = 100$, per-GPU batch $b = 4$, and a proposed gradient-accumulation depth of 32 steps, giving $B_{\text{global}} = 64 \times 4 \times 32 = 8192$ samples.

- ☐ **Critical batch size**: Calculate the critical batch size (B_{crit}), where the gradient noise scale equals the batch size.
- ☐ **Proposed batch**: Use the Gradient Noise Scale metric $\mathcal{B} \approx \frac{\text{tr}(\Sigma)}{\|\mu\|^2}$ to evaluate the proposed batch size.

- ☐ **Scaling efficiency**: Classify scaling efficiency as meeting the 80 percent threshold or as marginal convergence gain from additional compute.

5.4.7 Worked example: Convergence comparison for 8 vs. 64 workers

To illustrate these concepts concretely, consider scaling from 8 to 64 workers when training a transformer language model with baseline batch size $B = 32$ per worker.

Napkin Math 5.6: Scaling from 8 to 64 workers

Setup: Transformer model with 1.3B parameters, target perplexity 15, baseline training: 100K iterations on single GPU with $b = 32$.

8 Workers (BSP)

- Effective batch size: $B = 8 \times 32 = 256$
- Learning rate: $\eta = 8 \times \eta_{\text{base}}$ (linear scaling with warmup)
- Expected iterations: 100K / 8 = 12.5K iterations
- Convergence: Reaches target perplexity in 12.8K iterations (98 percent efficiency)
- Communication overhead: 15 percent (NVLink intra-node)
- Wall-clock speedup: 100K $\times$ 1 / (12.8K $\times$ 1.15×) = 6.8×

64 Workers (BSP)

- Effective batch size: $B = 64 \times 32 = 2{,}048$
- Learning rate ($N = 64$ workers): $\eta = N \times \eta_{\text{base}}$ (if $B < B^*$) or $\eta = \sqrt{N} \times \eta_{\text{base}}$ (if $B > B^*$)
- Assuming $B^* \approx 4{,}000$ (below critical): Linear scaling applies
- Expected iterations: 100K / 64 = 1.56K iterations
- Convergence: Reaches target perplexity in 1.72K iterations (91 percent efficiency)
- Communication overhead: 45 percent (InfiniBand inter-node, 8 nodes)
- Wall-clock speedup: 100K $\times$ 1 / (1.72K $\times$ 1.45×) = 40.1×

SSP workers: 64 workers ($s = 4$)

- Same effective batch size: $B = 2{,}048$
- Learning rate: $\eta' = \eta_{\text{BSP}}/\sqrt{1+s}$ with $s = 4$, giving $\eta' \approx 0.45 \times \eta_{\text{BSP}}$
- Expected iterations: Higher due to staleness penalty
- Convergence: Reaches target perplexity in 2.1K iterations (74 percent efficiency)
- Communication overhead: 25 percent (reduced synchronization)
- Wall-clock speedup: 100K $\times$ 1 / (2.1K $\times$ 1.25×) = 38.1×

Analysis (table 5.5)

Table 5.5: Scaling SGD across cluster sizes: Iteration count, communication overhead, wall-clock speedup, and sample efficiency for the same training job at 1, 8, and 64 GPUs.

Configuration	Iterations	Comm. Overhead	Wall-clock Speedup	Sample Efficiency
1 GPU (baseline)	100,000	0%	1×	100%
8 GPU BSP	12,800	15%	6.8×	98%
64 GPU BSP	1,720	45%	40.1×	91%
64 GPU SSP	2,100	25%	38.1×	74%

The 64-GPU BSP configuration achieves 40× speedup despite only 91 percent sample efficiency because the communication overhead (45 percent) is offset by the massive parallelism. SSP provides comparable wall-clock time with lower communication overhead but requires more total samples.

Cost Analysis (assuming \$3/GPU-hour):

- 8: 12.8K iters × 0.4 s/iter × 8/3600 × \$3/GPU-hour = \$34
- 64 BSP: 1.72K iters × 0.58 s/iter × 64/3600 × \$3/GPU-hour = \$53
- 64 SSP: 2.1K iters × 0.50 s/iter × 64/3600 × \$3/GPU-hour = \$56

Despite higher parallelism, 64-GPU training costs more per run due to communication overhead and reduced sample efficiency. The 8-GPU configuration is more cost-efficient but takes 6× longer wall-clock time. The choice depends on whether minimizing cost or minimizing time-to-result is the priority.

5.4.8 Trade-off: Communication cost vs. convergence speed

The fundamental trade-off in distributed training is between communication efficiency and convergence quality. Figure 5.9 visualizes this trade-off space.

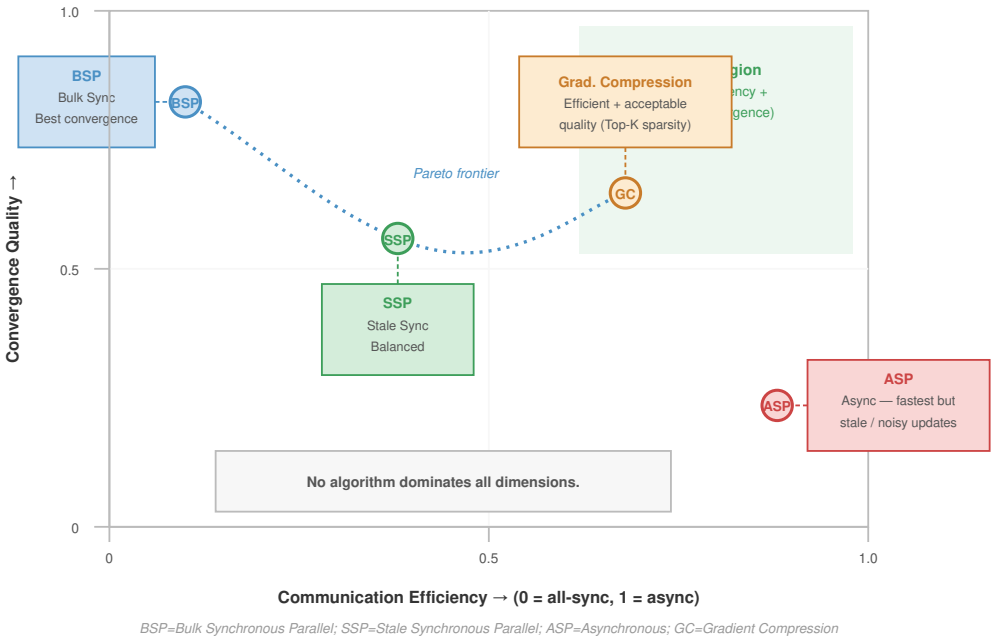


Figure 5.9: Communication-Convergence Trade-Off Space: Each point represents a different distributed training configuration. The Pareto frontier (dashed line) shows optimal configurations where improving one metric requires sacrificing the other. BSP sits at high convergence quality but lower throughput; ASP provides maximum throughput at convergence cost. Gradient compression and SSP occupy intermediate positions.

Several techniques occupy different positions on this trade-off curve. Gradient compression reduces communication volume by transmitting compressed gradient information through quantization, sparsification, or low-rank approximation. Examples include sign-based updates (Bernstein et al. 2018), Deep Gradient Compression (Lin et al. 2018), PowerSGD (Vogels et al. 2019), QSGD quantization with convergence analysis (Alistarh et al. 2017), and empirical sparse gradient dropping for neural machine translation (Aji and Heafield 2017).

Local SGD takes a different approach: workers perform H_{local} local updates before synchronizing, reducing communication frequency by factor H_{local}. Convergence analysis shows that for smooth, strongly convex objectives, Local SGD achieves the same asymptotic rate as synchronous SGD with appropriately tuned learning rates (Stich 2019).

Decentralized SGD restricts workers to communicating only with neighbors in a communication graph rather than performing global AllReduce. This reduces bandwidth requirements at the cost of slower mixing, making it suitable for geo-distributed training where global synchronization is expensive.

The choice among these methods depends on the specific bottleneck. When network bandwidth limits throughput, gradient compression provides the best trade-off. When synchronization latency dominates, Local SGD or SSP are preferred. When network topology constraints exist, decentralized approaches may be necessary.

5.5 Model Parallelism

When model state, activations, optimizer state, or an individual tensor operation exceeds the memory or communication budget of one accelerator group, data parallelism entirely collapses. This is the memory capacity gap (principle 3) in operational form: model parameter growth outpaces device memory growth, forcing the model's computation and state to be partitioned. Data-parallel memory optimizations extend replication's reach, but eventually the model itself must be partitioned.

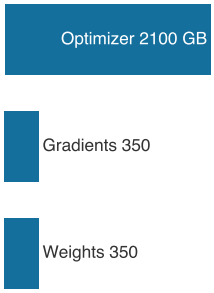
Definition 5.7: Model parallelism

Model Parallelism is a distributed training strategy that partitions a single neural network's parameters or operations across multiple devices so each device computes a distinct portion of the model.

1. **Significance**: If a model with parameter state S_{model} is split across N_{mp} devices, the ideal per-device parameter footprint falls toward $S_{\text{model}}/N_{\text{mp}}$, enabling training of models that exceed single-device memory. The cost is extra communication for activations and gradients plus idle time from sequential dependencies, so the memory gain must exceed the transfer and pipeline-bubble overhead introduced by the partition.
2. **Distinction**: Unlike data parallelism, which replicates the full model on every worker and shards the minibatch, model parallelism shards the model itself. Pipeline parallelism and tensor parallelism are implementation forms of model parallelism: one partitions layers into stages, while the other partitions individual tensor operations.
3. **Common pitfall**: A frequent misconception is that model parallelism automatically speeds up training. Its primary benefit is capacity, not throughput; without careful scheduling and overlap, downstream devices wait for upstream activations and utilization can fall below a single-device baseline.

Even with ZeRO-3 fully deployed, sharding optimizer states, gradients, and parameters across workers, some architectures remain intractable. Tensor parallelism, illustrated in figure 5.10, addresses this by partitioning individual weight matrices across devices; pipeline parallelism partitions layers across stages and is introduced later in section 5.5.2.2. For a 175B parameter model, weights alone occupy 350 GB, or about 5.5 GB per GPU across 64 GPUs. Full mixed-precision Adam training state is much larger: 2,800 GB globally, or roughly 43.8 GB per GPU before activations. This distinction matters because optimizer sharding reduces static state, but it does not eliminate the activation and per-layer capacity constraints that force model parallelism.

For long-context transformers where activation memory dominates, a 2048-token sequence through 175B parameters generates on the order of a terabyte of intermediate activations at even a small micro-batch (roughly 1.1 TB at micro-batch 4, as computed earlier), and no amount of optimizer sharding addresses this constraint. Model parallelism addresses these limitations by splitting the model architecture itself across devices, rather than replicating it with sharded state.



Optimizer state, not weights, dominates the per-replica training budget.

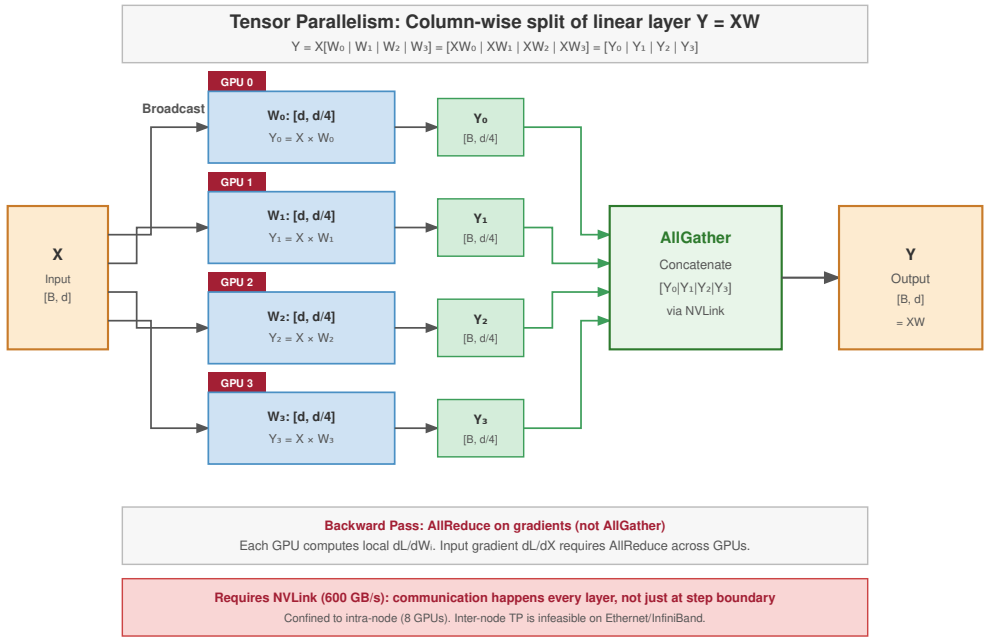


Figure 5.10: Tensor Parallelism Data Flow: Tensor parallelism column-splits a single weight matrix ($W_0 \dots W_3$) across four GPUs, with each GPU computing a partial output that is recombined through an AllGather. The companion strategy, pipeline parallelism, partitions layers vertically across stages and is shown separately in the layer-block figure. The two strategies are often combined in hybrid 3D parallelism for large models.

Napkin Math 5.7: The memory wall of scale

Problem: Training a 175B parameter model (like GPT-3) on NVIDIA A100s (80 GB). Can data parallelism with ZeRO-3 handle this?

Math:

1. **Parameter storage**: 175B params $\times$ 2 bytes (FP16) = 350 GB.
2. **Gradients and optimizer state**: gradients add 175B params $\times$ 2 bytes = 350 GB, while Adam FP32 master weights, momentum, and variance add 175B params $\times$ 12 bytes = 2,100 GB.
3. **Total static memory**: 2,800 GB.
4. **ZeRO-3 sharding**: With 64 GPUs, per-GPU static memory = 2,800 GB/64 GPUs $\approx$ 43.8 GB.
5. **Activation memory**: For sequence length 2048 and batch size 1, a 96-layer transformer generates $\approx$ 50 GB of activations per GPU.

Systems insight: 43.8 GB (static) + 50 GB (dynamic) = 93.8 GB. This exceeds the 80 GB capacity of the A100. Even with full ZeRO-3 sharding, pure data parallelism fails. The only recourse is tensor parallelism to split the layers themselves.

Model parallelism addresses this limitation by distributing the model architecture itself across devices, but the placement decision is where to cut the model. Layer-based splitting assigns devices to sequential layer groups, such as layers 1–4 on one device and layers 5–8 on the next. Channel-based splitting divides the channels within a layer, such as 512 channels on one device and the remaining channels on another. Transformer architectures add attention-head splitting, where separate devices own different heads. Each cut changes a different cost: layer cuts save memory

but serialize the pipeline, while channel and head cuts expose more parallel work but require faster intra-layer communication.

These cuts enable the large-scale cases that exceed a single device for different reasons. GPT-3, with 175B parameters, relies on model parallelism for training. Vision transformers processing high-resolution $16k \times 16k$ pixel images use model parallelism to manage activation and memory constraints. Mixture-of-Experts architectures use this approach to distribute their conditional computation paths across hardware (Shazeer et al. 2017; Lepikhin et al. 2021; Fedus et al. 2022).

Device coordination proceeds in three phases. In the forward pass, data flows sequentially through model segments on different devices. The backward pass propagates gradients in reverse order through these segments. During parameter updates, each device modifies only its assigned portion of the model. The coordination ensures mathematical equivalence to training on a single device while enabling models that exceed individual device memory capacities. The cost is sequential dependency between stages: while one device computes its forward pass, downstream devices sit idle, creating the pipeline bubble that dominates utilization analysis for model-parallel systems.

Checkpoint 5.3: Model parallelism foundations

Verify your understanding of model sharding:

- ☐ **Memory relief**: Determine whether model parallelism reduces the **memory footprint** per GPU for a given model.
- ☐ **Sequential dependency**: Identify whether sequential dependencies between model shards arise in the forward pass, the backward pass, or both.
- ☐ **Pipeline bubble**: Explain why pure model parallelism often leads to lower GPU utilization than data parallelism through the "pipeline bubble."
- ☐ **Layer vs. tensor split**: Distinguish layer-based and tensor-based ways to split a Transformer layer.

5.5.1 Model parallelism implementation

Once the model itself is the object being partitioned, the implementation problem is placement: every cut must save enough memory to justify the activation transfer and idle time it introduces. Figure 5.11 captures this bidirectional data flow: input data propagates forward through sequentially assigned model partitions while gradients flow backward to update parameters, with intermediate results transferring across device boundaries at each stage.

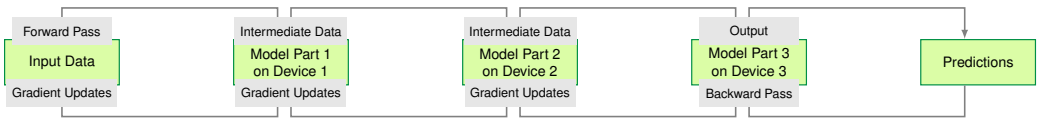


Figure 5.11: Model Parallelism Data Flow: Sequential distribution of model partitions across three devices: input data flows forward through each partition in order (top path), while gradients propagate backward during the update phase (bottom path). Each device handles a distinct portion of the model, with intermediate activations and gradients transferring at partition boundaries. This approach enables training of models exceeding single-device memory at the cost of sequential dependencies that reduce hardware utilization.

Consider our running example: the 175B parameter model requires 350 GB of memory in FP16, exceeding the 80 GB capacity of a single A100 by a factor of four. Model parallelism addresses this **capacity wall** by partitioning model weights across multiple devices, effectively stitching them into a single super-accelerator. Unlike data parallelism, where every GPU holds a full replica of the model and processes a unique fraction of the global batch, model parallelism requires each GPU to hold a *unique* fraction of the model and process the *same* data stream sequentially. With 8-way partitioning on A100s, BF16 weights alone occupy approximately 44 GB per GPU. That is only the weights budget: full training still needs activations, gradients, and optimizer state, so pure 8-way model parallelism is not sufficient by itself. Training requires additional sharding or offload for

optimizer and gradient state, plus pipeline or hybrid parallelism to keep activation memory within capacity.

In a typical pipeline parallel implementation, the training loop operates as a relay race. The forward pass initiates on GPU 1, which computes the initial transformer blocks and transmits the resulting intermediate activation tensor across the interconnect to GPU 2. For our 175B model with a hidden dimension of 12,288 and a micro-batch size of 4 sequences at 2,048 tokens each, this handoff involves moving approximately 200 MB of data per stage boundary per step. GPU 2 must wait for this payload before it can begin its computation, creating a strict dependency chain that propagates through all stages. The backward pass mirrors this path in reverse, propagating error gradients from the final layer back to the input, with each device computing gradients only for its local parameters.

The architecture fundamentally changes the optimization dynamics compared to data parallelism. Instead of a global AllReduce to average gradients across replicas, each GPU performs a local optimizer step (Adam (Kingma and Ba 2014), AdaFactor, or similar) on its specific slice of parameters. A device holding transformer layers 1–12 updates only those layers' weights and biases, with no cross-device synchronization required during the optimization step. While this eliminates the bandwidth-heavy gradient synchronization of data parallelism, it trades one bottleneck for another: pipeline bubbles. If the layers assigned to GPU 1 are computationally heavier than those on GPU 2 (common when attention layers have different head counts or when embedding layers are unevenly sized), valuable compute cycles are lost to waiting. The primary engineering challenge thus shifts from maximizing arithmetic intensity to minimizing serialization latency and ensuring balanced load across the partitioned fleet (Rasley et al. 2020).

5.5.2 Parallelism variations

Partitioning strategy determines which cost the model pays after memory no longer fits on one device. Layer-wise partitioning spends idle time to reduce per-device memory; pipeline parallelism spends scheduling complexity to hide that idle time; tensor parallelism spends NVLink bandwidth to split a single layer. The approaches below separate these choices by the architecture and interconnect that make each viable.

5.5.2.1 Layer-wise partitioning

Layer-wise partitioning is the least invasive model cut: it saves memory by assigning consecutive layers to separate devices, but it preserves the model's sequential dependence. In transformer architectures, this translates to specific devices managing defined sets of attention and feed-forward blocks. Figure 5.12 demonstrates this partitioning for a 16-layer transformer: four consecutive blocks reside on each of four devices, with forward activations flowing left-to-right and backward gradients propagating right-to-left across the device boundaries.

Sequential processing introduces device idle time, as each device must wait for the previous device to complete its computation before beginning work. While device 1 processes the initial blocks, devices 2, 3, and 4 remain inactive. Similarly, when device 2 begins its computation, device 1 sits idle. This pattern of waiting and idle time reduces hardware utilization efficiency compared to other parallelization strategies.

5.5.2.2 Pipeline parallelism

Pipeline parallelism extends layer-wise partitioning by introducing microbatching to minimize device idle time. Instead of waiting for an entire batch to sequentially pass through all devices, the computation is divided into smaller segments called microbatches, with overlapping execution across pipeline stages. Figure 5.13 shows how this overlapping works: while device 1 processes microbatch $i+1$, device 2 computes microbatch i, device 3 handles $i-1$, and device 4 executes $i-2$, creating a continuous flow that keeps all devices active simultaneously.

As figure 5.13 shows, each device, as represented by the rows in the drawing, processes its assigned model layers for different microbatches simultaneously. The forward pass involves devices passing activations to the next stage, such as $F_{0,0}$ to $F_{1,0}$. The backward pass transfers gradients back through the pipeline, such as $B_{3,3}$ to $B_{2,3}$. This overlapping computation reduces idle time and increases throughput while maintaining the logical sequence of operations across devices, giving the strategy its formal shape.

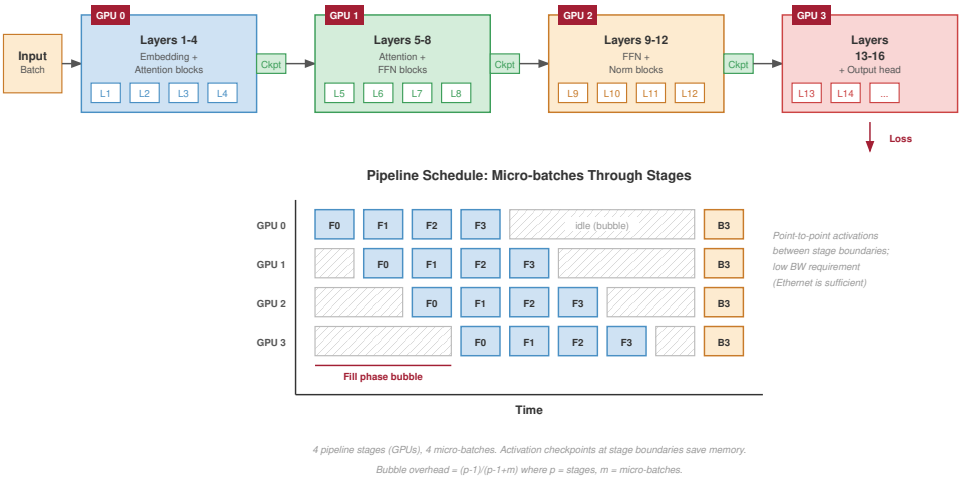
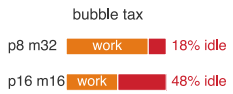


Figure 5.12: Layer-Wise Model Parallelism: A 16-layer transformer distributed across four GPUs, with four consecutive transformer blocks assigned to each device (L1–L4, L5–L8, L9–L12, L13–L16). Forward activations (black arrows) flow left-to-right through device boundaries, while backward gradients (red arrows) propagate right-to-left during parameter updates. This partitioning reduces per-GPU memory from the full model size to $1/4$, but introduces sequential dependencies where downstream devices wait for upstream computation to complete.

Definition 5.8: Pipeline parallelism

Pipeline Parallelism is a model parallelism technique that partitions a neural network's layers into sequential stages assigned to different devices, passing activations forward and gradients backward between stages while overlapping computation across stages using micro-batches to maintain throughput.

1. **Significance**: Inter-stage communication transmits only the activation tensor at each stage boundary, sized as $B_\mu \times S \times d_{\text{model}} \times 2$ bytes at BF16, where B_μ is the micro-batch size. For a hidden dimension of 8,192 with micro-batch size 1 and a 2,048-token sequence, this is approximately $8{,}192 \times 2{,}048 \times 2 \approx 32$ MB per boundary, compared to the gigabytes required for gradient AllReduce in data parallelism. This low communication volume makes pipeline parallelism the primary technique for scaling model depth across nodes connected by 50 GB/s InfiniBand. The pipeline bubble wastes approximately $(p-1)/(m+p-1)$ of total compute, where p is the number of stages and m is the number of micro-batches—with $p = 8$ stages and $m = 32$ micro-batches, bubble overhead is about 18 percent.
2. **Distinction**: Unlike tensor parallelism, which shards individual matrix multiplications within a single layer and requires AllReduce on every layer's output (demanding NVLink-class bandwidth within each stage), pipeline parallelism shards at layer boundaries and requires only point-to-point activation transfers between stages—tolerating InfiniBand bandwidth between nodes.
3. **Common pitfall**: A frequent misconception is that adding more pipeline stages always improves throughput. The pipeline bubble fraction grows as $(p-1)/(m+p-1)$: with $p = 16$ stages and $m = 16$ micro-batches, 48 percent of compute is wasted idle—requiring large micro-batch counts ($m \gg p$) to keep the bubble below 10 percent, which in turn increases peak activation memory and the memory pressure on each stage.



More stages require enough microbatches or the bubble dominates utilization.

The same bubble term becomes concrete in the worked example below, where an 8-stage pipeline and 32 microbatches still leave a measurable idle-time tax.

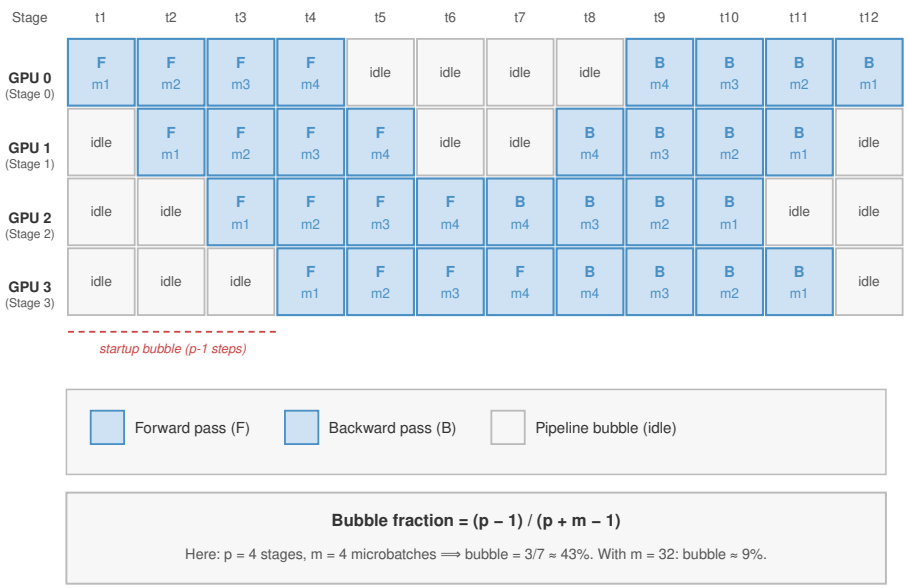


Figure 5.13: Pipeline Parallelism Schedule: A 4-stage pipeline processing 4 microbatches, showing forward passes ($F_{i,j}$) and backward passes ($B_{i,j}$) across time. Rows represent pipeline stages (GPUs), columns represent time steps. The staggered execution keeps all devices active: while stage 0 computes $F_{0,1}$, stage 1 processes $F_{1,0}$ from the previous microbatch. After all forward passes complete, backward passes propagate in reverse order. The "Update" column shows synchronized parameter updates after gradient accumulation across all microbatches.

Napkin Math 5.8: The cost of the pipeline bubble

Problem: A large-model training run uses pipeline parallelism across 8 stages. To hide the sequential delay, the batch is split into 32 microbatches. What is the "bubble tax"—the fraction of GPU cycles lost to idle waiting?

Math: In a synchronous pipeline (1F1B), the bubble fraction is determined by the ratio of stages to microbatches.

1. **Wait time**: At the start and end of each batch, GPUs sit idle for $p-1$ steps.
2. **Productive time**: GPUs compute for m steps.
3. **Bubble fraction**: $(p-1)/(p-1+m) = 7 \,/\, (7+32) \approx 17.9$ percent.

Systems insight: Pipeline parallelism is a *utilization-memory trade-off*. Reducing the bubble from 18 percent to 5 percent requires more microbatches (m), which consume additional activation memory on each GPU. In the machine learning fleet, depth does not scale for free; the cluster pays a 17.9 percent capacity tax just to keep the stages coordinated. This is why techniques like interleaved pipelining are essential: they chop the bubble into smaller pieces to recover that lost 18 percent of fleet capacity.

GPipe[6] (Huang et al. 2019) introduced synchronous pipeline parallelism with micro-batch accumulation, while PipeDream (Narayanan et al. 2019) developed asynchronous approaches with weight stashing. Modern systems employ **1F1B (One-Forward-One-Backward)**[7] scheduling to reclaim activation memory earlier; that lower memory pressure makes larger microbatch counts practical, and the larger m is what shrinks the bubble term.

6 **GPipe**: Published by Google in 2019, GPipe introduced synchronous micro-batch pipelining that trained a 557M-parameter AmoebaNet across 8 Tensor Processing Units (TPUs) with near-linear scaling. The key trade-off GPipe exposed: the pipeline bubble fraction $(p-1)/(m+p-1)$ means that with $p=4$ stages and $m=4$ micro-batches, 43 percent of compute is wasted in idle time – driving the field toward schedules that reduce activation residency and make larger micro-batch counts feasible.

At the schedule level, 1F1B is a warmup, steady-state, and drain loop. Algorithm 5.1 makes the contract explicit: each stage alternates forward work for newer microbatches with backward work for older microbatches once gradients arrive, so activations can be released before the whole batch has completed.

Algorithm 5.1 One-forward-one-backward pipeline schedule

Require: p ordered pipeline stages; m microbatches; activation links $j \to j{+}1$ and gradient links $j{+}1 \to j$
Ensure: one optimizer update after gradients accumulate across all m microbatches
1: split the global batch into m microbatches; assign consecutive layers to the p stages
2: warm-up: launch forwards from stage 0, forwarding each activation downstream as produced
3: **for** each stage in steady state **do**
4: alternate one backward (oldest ready microbatch) with one forward (next microbatch)
5: release a microbatch's activation as soon as that stage's backward consumes it
6: **end for**
7: drain: finish the remaining backward passes in reverse stage order
8: apply the optimizer update once every stage has accumulated gradients for all m microbatches

The warm-up and drain phases still leave a bubble of roughly $(p-1)/(m+p-1)$, but interleaving backward work in the steady state frees each activation as soon as its backward consumes it, so peak activation memory grows with the stage count rather than the microbatch count. That is the gain over GPipe-style all-forward-then-all-backward scheduling, and it lets the system raise m until activation memory, recomputation, or the global-batch constraint becomes the next limit. In a transformer model distributed across four devices, device 1 would process blocks 1-6 for microbatch $i+1$ while device 2 computes blocks 7-12 for microbatch i. Simultaneously, device 3 executes blocks 13-18 for microbatch $i-1$, and device 4 processes blocks 19-24 for microbatch $i-2$. Each device maintains its assigned transformer blocks but operates on a different microbatch, creating a continuous flow of computation.

The transfer of hidden states between devices occurs continuously rather than in distinct phases. When device 1 completes processing a microbatch, it immediately transfers the output tensor of shape $B_\mu \times S \times d_{\text{model}}$ to device 2 and begins processing the next microbatch. This overlapping computation pattern maintains full hardware utilization while preserving the model's mathematical properties.

One important variant targets the remaining bubble directly. GPUs at the beginning of the pipeline are idle while waiting for gradients to flow back from the end, and vice versa. These bubbles represent wasted compute. The 1F1B schedule in algorithm 5.1 keeps SMs busy during the steady state and reduces activation residency, but the fill and drain bubble remains unless the system increases m or schedules useful work into those otherwise idle slots.

Zero-bubble pipeline schedules further reduce idle time by overlapping weight gradient computation with activation gradient communication. In a standard backward pass, the GPU computes $\partial\mathcal{L}/\partial W$ (weight gradient) and $\partial\mathcal{L}/\partial X$ (activation gradient, sent to the previous stage) together. Zero-bubble scheduling splits these into separate kernels: a B kernel that computes only the activation gradient $\partial\mathcal{L}/\partial X$ and sends it to the previous stage, and a W kernel that computes the weight gradient $\partial\mathcal{L}/\partial W$ locally. The B kernel must execute promptly (it is on the critical path), but the W kernel can be scheduled opportunistically to fill bubbles.

The scheduling freedom provided by this B/W split is substantial. In a 4-stage pipeline with 8 microbatches, the standard 1F1B schedule has a bubble fraction of approximately $(p-1)/(m+p-1)$ where p is the number of stages and m is the number of microbatches. For $p=4, m=8$, this is $3/11 \approx 27\%$ idle time. Zero-bubble scheduling can reduce this to near zero by filling the startup and teardown bubbles with W computations.

The trade-off is memory: zero-bubble scheduling requires storing intermediate activations for longer (because the W computation is deferred), increasing peak memory usage. Some implementations address this by combining zero-bubble scheduling with activation checkpointing, selectively recomputing certain activations rather than storing them. The interaction between these techniques

7 **1F1B Scheduling**: "One-Forward-One-Backward." Unlike GPipe (which processes all forward passes before any backward passes), 1F1B interleaves them. This reduces the peak activation memory footprint from $\mathcal{O}(m \times p)$ to $\mathcal{O}(p)$, where m is the number of micro-batches and p is the number of pipeline stages. This memory reclamation is what enables the massive micro-batch counts ($m \gg p$) required to keep the pipeline bubble small.

creates a three-way trade-off among pipeline bubble size, memory consumption, and recomputation overhead, an example of the displacement of overhead (principle 13).

5.5.2.3 Tensor parallelism

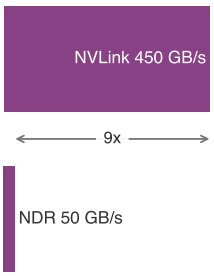
Pipeline parallelism, examined above, addresses device idle time by overlapping microbatch processing across stages. Each device holds complete layers and processes them sequentially, with communication only at stage boundaries when activations transfer between devices. This approach tolerates moderate interconnect bandwidth because communication occurs infrequently, once per layer boundary per microbatch. However, pipeline parallelism cannot help when individual layers themselves exceed device memory, or when the communication pattern within layers benefits from a different granularity than layer boundaries.

Tensor parallelism takes a fundamentally different approach: instead of assigning complete layers to devices, it splits the weight matrices within each layer. This operator-level parallelism (also called intra-layer parallelism) enables finer-grained distribution but requires high-bandwidth interconnects for the frequent intra-layer communication it introduces.

Definition 5.9: Tensor parallelism

Tensor Parallelism is a model parallelism technique that partitions individual tensor operations—primarily matrix multiplications—across multiple devices using column-parallel or row-parallel weight splits, typically requiring two AllReduce operations per transformer layer in Megatron-style transformer blocks to sum partial results from all participating devices.

1. **Significance**: Megatron-LM style tensor parallelism places two AllReduce operations per transformer block—one after attention and one after the MLP—with each collective reducing an activation tensor of size $B_{\text{batch}} \times S \times d_{\text{model}} \times 2$ bytes in BF16. A ring AllReduce moves roughly $2(t-1)/t$ times that payload per GPU. At degree $t = 8$ on NVLink with 900 GB/s bandwidth, each AllReduce takes approximately 0.1–0.5 ms. The same operation over InfiniBand (50 GB/s) takes 2–10 ms per layer—with 96 transformer blocks, this adds 200–960 ms of pure synchronization overhead per step, collapsing MFU to single digits.
2. **Distinction**: Unlike pipeline parallelism, which communicates only at layer boundaries and transfers small activation slices, tensor parallelism synchronizes within every layer via AllReduce, making it sensitive to the per-operation latency of the interconnect and restricting it to within-node NVLink fabrics in practice.
3. **Common pitfall**: A frequent misconception is that tensor parallelism can simply be scaled across nodes over InfiniBand. NVLink delivers approximately 900 GB/s bidirectional, or 450 GB/s per direction; InfiniBand NDR delivers 50 GB/s per port—a 9× per-direction gap. Running tensor parallelism across InfiniBand typically collapses MFU to single-digit percentages, making the communication overhead larger than the compute benefit of distributing the matrix multiplication.



Tensor parallelism needs NVLink bandwidth; pipeline parallelism tolerates slower fabric.

The interconnect topology dictates which form of model parallelism is viable at each level of the cluster hierarchy. Tensor parallelism's per-layer synchronization demands NVLink-class bandwidth; pipeline parallelism's boundary-only communication tolerates InfiniBand. This bandwidth pattern creates the design pressure for a hybrid: use tensor parallelism for bandwidth-intensive intra-layer splits and pipeline parallelism for coarser inter-layer splits.

Megatron-style tensor parallelism[8] (Shoeybi et al. 2019) partitions matrix multiplications in two ways. Examine figure 5.14: column-parallel splitting divides weight matrices along columns for QKV projections, allowing independent computation across GPUs, while row-parallel splitting divides along rows for output layers, requiring AllReduce to combine partial sums at the end of each block.

Column-parallel linear layers split weights along columns. For input X and weight matrix $W = [W_1|W_2]$ split across 2 GPUs:

$$Y = XW = X[W_1|W_2] = [XW_1|XW_2]$$

8 **Megatron-LM**: NVIDIA's 2019 framework that trained an 8.3B-parameter transformer – 24× BERT and 5.6× GPT-2 at the time – by strategically placing only two AllReduce operations per transformer block (one after attention, one after the multilayer perceptron (MLP)). This column-then-row partitioning eliminates inter-GPU communication between the two linear layers within each block, achieving 76 percent scaling efficiency across 512 GPUs and establishing tensor parallelism patterns that remain influential in large-scale transformer training.

Each GPU computes its partition independently, and the outputs are concatenated without communication when the next operation is row-parallel. The row-parallel half of the pattern splits weights along rows, $W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}$:

$$Y = XW = X_1W_1 + X_2W_2$$

Each GPU computes a partial sum, and the system pays one AllReduce to combine the partial outputs.

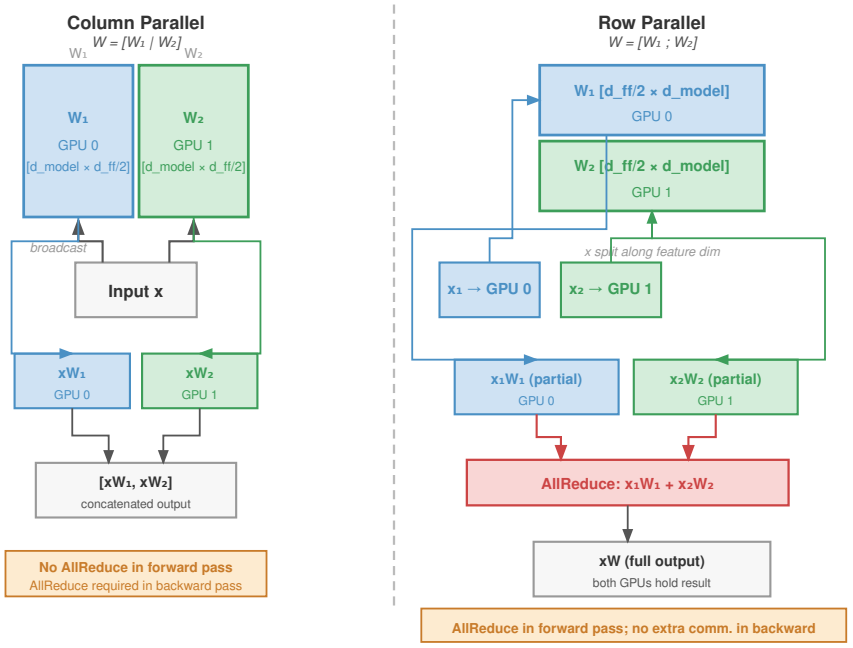


Figure 5.14: Tensor Parallelism - Matrix Partitioning: Illustration of Megatron-LM style tensor parallelism. The first linear layer (e.g., QKV) is split column-wise $[W_1|W_2]$. The second layer (e.g., Output) is split row-wise $[W_1;W_2]$. This arrangement allows the output of the first layer to flow directly into the second without synchronization, requiring only one AllReduce after the second layer to sum the partial results.

The column-then-row arrangement shown in figure 5.14 is the key design insight. By pairing a column-parallel layer with a row-parallel layer, the intermediate activations flow directly between them without communication, confining synchronization to the end of the pair. Megatron applies that pattern twice in each transformer block: the QKV projection is column-parallel and its attention output projection is row-parallel, then the first feed-forward layer is column-parallel and the second feed-forward layer is row-parallel. The design places AllReduce operations strategically, one after attention and one after the feed-forward network, for two AllReduce operations per transformer layer. Communication volume per transformer layer depends on sequence length S, hidden dimension d_{model}, tensor-parallel degree t, and batch size B. The activation payload is:

$$M_{\text{act}} = B \times S \times d_{\text{model}} \times \text{sizeof(dtype)}$$

A ring AllReduce transfers approximately $2(t-1)/t \times M_{\text{act}}$ bytes per GPU, which is close to $2M_{\text{act}}$ for large t. With $S = 2048$, $d_{\text{model}} = 4096$, $B = 4$, and FP16, the activation payload is $4 \times 2048 \times 4096 \times 2 \approx 67$ MB. Applying the ring factor gives roughly 134 MB per AllReduce. For a 96-layer model with two AllReduce operations per layer, this totals approximately 25.7 GB per forward pass; including the symmetric backward AllReduces brings per-training-step communication to roughly 51 GB, requiring NVLink bandwidth to avoid becoming the bottleneck.

Tensor parallelism scaling degrades rapidly beyond 8-way parallelism because the same split that reduces memory also reduces the computation available to hide communication. As the tensor-parallel degree grows, per-GPU work falls while collective latency and aggregate traffic rise; per-GPU ring AllReduce bytes approach $2\times$ the activation tensor size rather than growing linearly without bound, but the smaller local matrix multiply leaves less useful work to cover that exchange. Once NVLink bandwidth saturates, adding another tensor-parallel shard mainly exposes more synchronization rather than more throughput. Published systems such as Llama 3 405B use TP=8 within NVLink-connected H100 nodes (Dubey et al. 2024), and similar node-local tensor parallelism is a common design pattern for large LLM training (Z. Jiang et al. 2024).

The same partitioning idea can apply to sequence length as well as matrix dimensions. While standard tensor parallelism tiles computation across the HBM-NVLink boundary within a node, **Ring Attention** partitions the attention state across GPUs rather than forcing every device to hold the full prefix (Liu et al. 2023). This becomes relevant for sequences that exceed a single GPU's memory, such as million-token context windows. Each GPU owns a block of queries Q and circulates the key/value (K/V) blocks around a ring, computing attention against the block currently resident in memory and then forwarding that block to its neighbor.

The algorithm proceeds in $N-1$ communication rounds (where N is the number of GPUs). In each round, each GPU computes attention between its local Q block and the currently resident K/V block, sends that K/V block to its ring neighbor, and receives the next block from its other neighbor. The implementation overlaps this exchange with computation: while GPU i computes attention using K_j/V_j, it simultaneously receives K_{j+1}/V_{j+1} from the ring.

If the compute time for one tile exceeds the communication time for transferring one tile over NVLink, the communication is fully hidden. On an H100 with 3.35 TB/s HBM bandwidth and 900 GB/s NVLink bandwidth, this overlap is achievable for typical tile sizes.

The practical impact of Ring Attention is measured in context length. Without it, a single GPU's attention computation is limited by HBM capacity: the key/value state for a sequence of length S must fit entirely in one GPU's memory. With Ring Attention across N GPUs, each GPU holds S/N tokens of that state, enabling context lengths of $N \times S_{\text{single}}$. Chapter 9 later connects this sequence-level distribution to FlashAttention's tile-level HBM reuse; here, the distributed-training lesson is that sequence length can become a partitioning dimension just like layers, tensors, or data.

5.5.3 Parameter servers and embedding sharding

While AllReduce dominates dense model training, the **Parameter Server (PS)** architecture, formalized by Li et al. (2014), remains common for recommendation systems and other sparse workloads. A Parameter Server architecture separates workers (who compute gradients) from servers (who store parameters and apply updates).

For dense models (like ResNet or BERT), the PS architecture creates a bottleneck: with N workers, the server's inbound bandwidth must absorb N gradient streams simultaneously, saturating the server's network bandwidth and making it the communication chokepoint. This "incast" problem drove the adoption of Ring AllReduce, where each worker sends and receives at its own link rate, achieving N-fold higher aggregate bandwidth by distributing the load across all nodes. For dense models beyond 4–8 GPUs, decentralized AllReduce wins decisively.

However, for **Recommendation Systems**—specifically DLRM—the model parameters are dominated by massive Embedding Tables (often 10 TB+) that cannot fit on any single GPU. Furthermore, the updates are sparse: a batch of users interacts with only a tiny fraction (e.g., 0.001 percent) of the items.

In this sparse regime, the parameter server avoids dense synchronization by moving only the rows that a batch touches:

1. **Embedding Sharding**: The massive tables are partitioned across the PS fleet (often CPU nodes with massive DRAM).
2. **Sparse Lookups**: Workers send a list of IDs to the PS.
3. **Sparse Pull**: The PS returns only the requested embedding vectors, not the full table.
4. **Sparse Push**: Workers send gradients only for the touched embedding rows.

The **Sparse Pull/Sparse Push** pattern avoids the bandwidth bottleneck of dense AllReduce. Implementations such as TorchRec or Meta's hierarchical sharding place "hot" embeddings on GPUs and "cold" embeddings on CPU PS nodes, creating a tiered memory hierarchy for model parameters.

5.5.4 Expert parallelism (mixture of experts)

While tensor parallelism splits dense layers across devices, expert parallelism enables scaling model capacity (parameters) without increasing compute cost (FLOPs) by using conditional computation. In a Mixture-of-Experts (MoE) architecture (Shazeer et al. 2017), the feed-forward network (FFN) of each transformer block is replaced by a set of E "experts" (independent FFNs). For each token, a gating network selects a small subset (typically top-1 or top-2) of experts to process it.

In a distributed setting, experts are partitioned across workers. If we have 8 GPUs and 8 experts, each GPU hosts one expert. The training process introduces a distinct communication pattern, as figure 5.15 illustrates:

1. **Gating**: Each token determines its destination expert.
2. **All-to-All Dispatch**: Tokens are routed across the network to the device hosting their selected expert.
3. **Computation**: Experts process their assigned tokens.
4. **All-to-All Combine**: Processed tokens are routed back to their original device to resume the sequence.

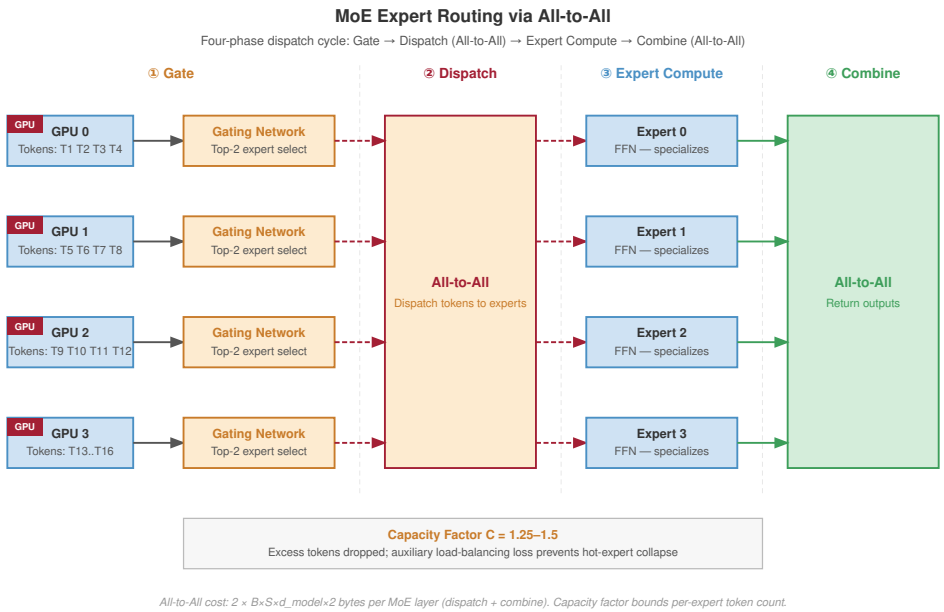


Figure 5.15: Mixture of Experts (MoE) All-to-All Routing: Expert parallelism requires a unique 'All-to-All' communication pattern. Tokens from a local batch are routed by a gating network to experts sharded across the cluster. This involves a global shuffle where every GPU sends tokens to and receives tokens from potentially every other GPU, stressing the cluster's bisection bandwidth.

The primary advantage is decoupling model size from compute budget. A trillion-parameter MoE model might use only 10B parameters per token, enabling training on feasible hardware budgets. The constraint is the **All-to-All** communication, which is bandwidth-intensive and sensitive to load imbalance.

Definition 5.10: Expert parallelism

Expert Parallelism is the distribution strategy for mixture-of-experts models that places different experts on different devices and routes each token's activations to the devices hosting its selected experts through All-to-All collectives, scaling total parameter count across the cluster without scaling the compute each token consumes.

1. **Significance**: It decouples model capacity from compute budget: a trillion-parameter MoE model may activate on the order of 10B parameters per token, so per-token FLOPs remain near those of a 10B dense model while capacity grows roughly 100×. The price is communication structure: every MoE layer requires two All-to-All shuffles per pass (dispatch and combine) whose volume scales with tokens moved times hidden dimension, stressing the cluster's bisection bandwidth rather than any single link.
2. **Distinction**: Unlike tensor parallelism, which splits dense operations so that every device computes a fraction of every token, expert parallelism is conditional computation: each device computes only the tokens routed to its experts, making both the communication pattern and the load distribution data-dependent rather than fixed by the architecture.
3. **Common pitfall**: A frequent misconception is that experts share work evenly. Natural token distributions are skewed: a hot expert can receive 3–5× its fair share of traffic, overflowing its device's activation buffer while sibling devices idle, which is why production MoE systems cap per-expert token budgets and penalize routing skew during training rather than trusting the router alone.

At the heart of expert parallelism lies the All-to-All communication primitive, which shuffles tokens across the cluster based on dynamic routing decisions. Consider a configuration with $E = 64$ experts distributed across 64 GPUs, processing a batch of $B = 4$ sequences at length $S = 2048$ with hidden dimension $d_{\text{model}} = 4096$. For every MoE layer, the system must dispatch $B \times S$ tokens to their assigned experts. In FP16, this moves $B \cdot S \cdot d_{\text{model}} \cdot 2$ bytes—approximately 67 MB—in a single direction. Since the processed embeddings must return to their original device for the residual connection, the total network overhead is roughly 134 MB per transformer block. While manageable in isolation, this latency accumulates rapidly in deep, sparse architectures like the Switch Transformer (Fedus et al. 2022) (up to 2,048 experts) or GShard (Lepikhin et al. 2021).

Network efficiency relies on the assumption of uniform token distribution, but natural language is inherently skewed: specific experts handling common syntax or connector words may receive 3–5× their fair share of traffic. A hot expert is both a memory problem and a scheduling problem. If one expert receives too many tokens, its GPU runs out of activation buffer while other experts sit underused.

MoE systems therefore enforce a hard limit defined by the **capacity factor** C, typically set between 1.25 and 1.5. This parameter caps the maximum number of tokens an expert processes at roughly $C \cdot (B \times S)/E$ for the routing group. If the routing gate assigns more tokens than this buffer allows, the excess tokens are dropped, passing through the layer unprocessed via the residual connection. To reduce that data loss, training objectives include an **auxiliary load balancing loss**, weighted at 0.01–0.1 relative to the main cross-entropy loss, that penalizes the router for favoring specific experts. Models such as Mixtral 8x7B use top-2 routing across 8 experts, achieving a favorable balance between capacity scaling and routing stability.

The sparse communication pattern distinguishes recommendation and MoE workloads (Archetype B (DLRM at Scale)) from dense LLM training (Archetype A (GPT-4/Llama-3)) (section 1.6.1).

Lighthouse 5.1: Archetype B (DLRM at Scale): DLRM vs. LLM scaling

Each scales through a different mechanism, bound by a different resource:

- **LLMs (Dense)**: Scale via Tensor/Pipeline Parallelism. Constraint: compute and interconnect bandwidth (NVLink).

- **DLRM-style models (Sparse)**: Scale via Embedding Sharding (Parameter Servers). Constraint: memory capacity and interconnect latency (Random Access).

The distinction dictates fundamentally different cluster designs: dense GPU pods for LLMs vs. memory-rich CPU/GPU hybrids for RecSys.

5.5.5 Trade-offs: The bubble vs. bandwidth dilemma

Model parallelism breaks the memory wall but introduces sequential dependencies that reduce hardware utilization. The engineering challenge is balancing pipeline bubbles (idle time) against all-to-all bandwidth (communication time).

Model parallelism offers three principal advantages. Memory scaling enables training of models that exceed single-device capacity: with 8-way tensor parallelism, the FP16 weight slice of a 175B model fits within A100 HBM before optimizer state and activation budgets are counted. Splitting the model also allows larger global batch sizes without out-of-memory errors, since each GPU processes a smaller parameter slice. The approach maps naturally to the physical structure of transformers, where attention heads split via tensor parallelism and layers split via pipeline parallelism.

The advantages come at the cost of three fundamental limitations. Pipeline bubbles cause GPUs to sit idle while filling and draining the pipeline; the bubble fraction is approximately $(p-1)/(m+p-1)$, where p is pipeline stages and m is microbatches, and achieving more than 90 percent efficiency requires $m \gg p$, which increases activation memory. Communication intensity in tensor parallelism is equally constraining: two AllReduce operations execute per layer on the critical path, demanding extremely high-bandwidth, low-latency interconnects (NVLink) and typically preventing scaling beyond a single node (8 GPUs) before hitting the bandwidth wall. Implementation complexity rounds out the trade-off, requiring invasive changes to the model definition—replacing standard linear layers with column-parallel and row-parallel variants—unlike data parallelism, which wraps the model externally without modifying internals.

5.6 FP8 for distributed training

Numerical precision is itself a distribution lever: halving the bytes per value halves the gradient and activation payloads that the AllReduce and tensor-parallel collectives move, shrinking the $T_{\text{comm}}(N)$ term that dominates step time at scale. The 8-bit floating point (FP8) format exploits this lever by reducing the numerical precision of computation while trying to preserve enough range for stable optimization.

FP8 matters to distributed training when its narrower formats reduce communication without destabilizing optimization. Traditional mixed-precision training uses FP32 master weights with FP16 forward and backward passes. FP8-capable accelerators such as the NVIDIA H100 add support for FP8, offering two formats optimized for different phases of training (Micikevicius et al. 2022).

The choice between the two FP8 formats is therefore not an instruction-set detail; it decides which tensors can be narrowed without breaking the optimizer. E4M3 (4-bit exponent, 3-bit mantissa) provides a range of approximately ± 448 with moderate precision, making it suitable for *weights and activations* in the forward pass where values cluster in predictable distributions. E5M2 (5-bit exponent, 2-bit mantissa) provides a much larger range of approximately ± 57344 but coarser precision, which is why it is used for *gradients* that can span many orders of magnitude during backpropagation. Using E4M3 for gradients would cause frequent overflow and underflow, while E5M2 captures the full gradient distribution at the cost of slightly noisier updates. Read table 5.6 as a distribution constraint map: each row changes the balance among range, precision, memory traffic, and trainability.

Table 5.6: Training Precision Formats: Each row represents a floating-point format used in training. FP8 formats (E4M3, E5M2) occupy the communication-efficient middle ground between FP16/BF16 training stability and the byte reduction needed to shrink distributed payloads.

Format	Exponent	Mantissa	Range	Precision	Use Case
FP32	8 bits	23 bits	$\pm 3.4 \times 10^{38}$	Very high	Master weights
FP16	5 bits	10 bits	± 65504	High	Mixed-precision
BF16	8 bits	7 bits	$\pm 3.4 \times 10^{38}$	Moderate	Training
E4M3	4 bits	3 bits	± 448	Low	FP8 forward pass
E5M2	5 bits	2 bits	± 57344	Very low	FP8 gradients

As table 5.6 summarizes, the critical engineering challenge in FP8 training is *dynamic scaling*. FP8's narrow dynamic range means that a fixed scale factor will cause either overflow or underflow. Per-tensor scaling multiplies each tensor by a scale factor before casting to FP8, then divides by that factor after the FP8 computation. The scale factor is adjusted dynamically, typically by tracking the running maximum absolute value of each tensor and choosing a scale that maps this maximum to near the FP8 maximum representable value.

The three-precision approach (FP32 master weights, FP8 general matrix multiply (GEMM) operations, FP16 accumulation) can achieve near-FP16 training quality while improving effective throughput on FP8-capable hardware for suitable workloads. The systems implication is concrete but hardware-dependent: FP8 halves the bytes moved relative to FP16/BF16 tensors, while operation throughput and energy gains depend on the accelerator implementation. The throughput gain is therefore a performance expression of narrower movement and hardware support, not an additional multiplicative factor.

For distributed training, FP8 earns its place only when smaller payloads reduce communication while dynamic scaling protects convergence. It changes the payload size, not the parallelization axis. If precision reduction is still insufficient, the system must decide how data, tensor, and pipeline parallelism map onto the hardware hierarchy.

5.7 Hybrid Parallelism

Training a large model when data parallelism runs out of memory and model parallelism runs out of network bandwidth requires orchestrating both strategies simultaneously across three dimensions. The preceding sections revealed a fundamental tension: data parallelism scales throughput but demands massive memory, while model parallelism enables large models but starves the compute.

Hybrid parallelism resolves this tension by applying both strategies orthogonally: model parallelism splits the architecture to fit available memory, while data parallelism scales throughput across multiple model replicas. Training a 175B parameter language model on a dataset of 300 billion tokens demonstrates this approach in practice. The neural network layers distribute across multiple GPUs through model parallelism, while data parallelism enables different GPU groups to process separate batches. This dual strategy addresses both memory constraints from model size and computational demands from dataset scale simultaneously, and it is precisely this combination that defines Archetype A training at large-model scale.



Lighthouse 5.2: Archetype A (GPT-4/Llama-3): physics of 3D parallelism

Archetype A (GPT-4/Llama-3) is the primary driver for hybrid parallelism. Because the model parameters (P) exceed the memory of any single accelerator $(C_{\text{mem,device}})$, and the training dataset (D) requires massive throughput, we must split the problem along three orthogonal axes:

1. **Tensor parallelism**: Splits individual layers to fit P within a node's memory.
2. **Pipeline parallelism**: Splits layers across nodes when the parameter footprint exceeds a single node.
3. **Data parallelism**: Replicates the entire split-model pipeline to scale throughput on D.

Only by combining all three can we train Archetype A systems efficiently.

To see why the combination matters, consider three parallelism configurations for training the 175B model on a 1,024-GPU cluster organized as 128 nodes of 8 GPUs each. The progression from pure data parallelism through tensor and pipeline parallelism makes the efficiency gain concrete.

Configuration A: Pure Data Parallelism (DP-1024). All 1,024 GPUs replicate the full model (using ZeRO to shard optimizer states), and each GPU processes a different data shard. The gradient AllReduce exchanges 350 GB across the full InfiniBand fabric. As the scaling-efficiency analysis showed, this achieves approximately 37.5 percent efficiency because the inter-node communication dominates.

Configuration B: TP-8, DP-128. Within each node, 8 GPUs use tensor parallelism over NVLink. Across nodes, 128 data-parallel groups synchronize gradients over InfiniBand. Each data-parallel group now needs to AllReduce only the gradients for its $1/8$ shard of the model (43.75 GB instead of 350 GB), and the TP communication is contained within the fast NVLink domain. The inter-node AllReduce time drops from 14 seconds to approximately 1.75 seconds. If the compute time is still 2.1 seconds, the efficiency improves to $2.1/(2.1+1.75) \approx 54.5$ percent, and with communication-computation overlap, practical efficiency reaches 75–85 percent.

Configuration C: TP-8, PP-4, DP-32. Within each node, 8 GPUs use tensor parallelism. Across 4 nodes, pipeline parallelism divides the model into 4 stages. Across the remaining 32 data-parallel groups, gradient synchronization occurs over InfiniBand. Pipeline parallelism reduces the gradient AllReduce volume further (each stage has roughly $1/4$ of the parameters), and the pipeline communication (forwarding activations between stages) has lower volume than a full AllReduce. The trade-off is the pipeline bubble: at the beginning and end of each microbatch, some pipeline stages are idle while waiting for activations from earlier stages or gradients from later stages. The bubble fraction is approximately $(p-1)/m$, so with 4 pipeline stages and 32 microbatches per training step, the bubble wastes roughly 9 percent of compute.

The three-way comparison reveals a clear pattern: configurations that keep high-bandwidth communication (tensor parallelism) within the fast NVLink domain and push only low-bandwidth communication (data-parallel AllReduce of smaller gradient shards) onto the slower InfiniBand fabric achieve the highest efficiency. *The infrastructure hierarchy dictates the parallelism hierarchy.* The resulting principle is called hierarchy-aware parallelism, and it is a common approach for large-scale training systems.

Definition 5.11: Hierarchy-aware parallelism

Hierarchy-Aware Parallelism is the strategy of mapping different parallel execution modes to the physical bandwidth tiers of the cluster.

1. **Significance**: It ensures that high-frequency synchronization (for example, tensor parallelism) stays on the fastest links (NVLink), while lower-frequency tasks (for example, data parallelism) use slower tiers (InfiniBand). This alignment maximizes scaling efficiency (η_{scaling}) by reducing exposed communication time $(T_{\text{comm}}(N))$ and latency (L_{lat}).
2. **Distinction**: Unlike Uniform Parallelism, which treats all node-to-node links as equal, Hierarchy-Aware strategies respect the Bandwidth Cliffs between die, node, and rack boundaries.
3. **Common pitfall**: A frequent misconception is that any model can be sharded across any number of nodes. In reality, if the Hierarchy Mapping is wrong (for example, sharding a large tensor across a slow inter-rack link), the communication time will dwarf the compute time, making the scale-out useless.

The interaction also flows in the reverse direction: the choice of parallelism strategy influences infrastructure design. A training system that uses TP-8, PP-4, DP-32 generates a communication pattern where the most bandwidth-intensive traffic (tensor-parallel AllReduce) is confined to within each node, the moderate-bandwidth traffic (pipeline stage communication) flows between groups of

4 neighboring nodes, and the lowest-bandwidth traffic (data-parallel AllReduce of reduced gradient shards) flows across the full cluster.

The layered communication pattern favors a hierarchical network topology where nearby nodes have higher bandwidth between them (a "locality-aware" topology) over a flat topology where all node pairs have equal bandwidth (a uniform fat-tree). Rail-optimized and hierarchical fat-tree designs exploit this locality, placing the nodes that communicate most frequently on the same switch or in the same rack, minimizing the number of switch hops for the most bandwidth-intensive traffic.

The practical implication for infrastructure procurement is that the network topology must be co-designed with the parallelism strategy, not selected independently. An organization that purchases a flat fat-tree fabric (optimized for any-to-any communication) but trains exclusively with hierarchy-aware parallelism (where most traffic is local) has over-provisioned the network's global bandwidth while potentially under-provisioning local bandwidth. Conversely, an organization that purchases a rail-optimized fabric (optimized for local communication) but later needs to run Mixture-of-Experts models with AllToAll communication (which requires global bandwidth) will find the fabric inadequate. The network fabric, which represents 10–15 percent of total system cost, must be matched to the anticipated workload mix, and changing the fabric after deployment is prohibitively expensive and disruptive.

5.7.1 The 3D training loop

Training a 175B parameter model requires **3D parallelism**: the coordinated composition of data, pipeline, and tensor parallelism across thousands of devices.[9] This approach does not merely sum the benefits of individual parallelism strategies; it composes them geometrically to match the physical topology of the hardware.

[9] **3D Parallelism**: Named after the three orthogonal axes of decomposition: (1) Data Parallelism (batch), (2) Pipeline Parallelism (depth), and (3) Tensor Parallelism (layer width). Organizations visualize their training fleets as a 3D grid (d, p, t), where the product $N_{\text{total}} = d \times p \times t$ equals the total GPU count. This geometric perspective is essential for balancing the tiered bandwidth constraints of high-bandwidth clusters.

Consider a training fleet configured with Tensor Parallelism (TP) of 8 GPUs, Pipeline Parallelism (PP) of 16, and Data Parallelism (DP) of 128. This configuration uses 16,384 GPUs (8 GPUs $\times$ 16 $\times$ 128) organized into a hierarchy of bandwidth domains.

The training step begins at the data-parallel level. Each of the 128 model replicas receives a distinct slice of the global batch. Within each replica, the model is split across 16 pipeline stages (nodes), with micro-batches flowing sequentially from the embedding layer on Node 0 to the loss calculation on Node 15. At the finest granularity, within each node, the 8 GPUs fuse into a single "super-accelerator" via TP. Every matrix multiplication in the forward pass is fractured across these devices, which must exchange partial results via high-bandwidth NVLink after every operation. For a 175B-scale hidden dimension, the activation payload is roughly 201.3 MB; the resulting tensor-parallel traffic is about 4.2 GB per pipeline stage per microbatch in the forward pass, or 8.5 GB including the backward pass. Across the full 96-layer replica, the forward tensor-parallel traffic is about 67.6 GB. This traffic is the highest-intensity in the system, but its exposed latency can remain small relative to compute because the chips communicate over 600 GB/s–900 GB/s local bandwidth.

The backward pass inverts this flow and exposes the critical dependencies between parallelism dimensions. As gradients flow backward through the pipeline, nodes exchange activation gradients point-to-point. This traffic is relatively light—roughly 201.3 MB per stage boundary—allowing it to traverse slower inter-node InfiniBand links without stalling the pipeline. The true bottleneck emerges at the end of the step: data-parallel synchronization. The full model's gradient state is still about 350 GB, but in a 3D-parallel layout each data-parallel group synchronizes the corresponding parameter shards for its tensor- and pipeline-parallel replica rather than every GPU moving the full tensor. Gradient bucketing groups ready layer-gradient tensors into larger messages as backward computation proceeds; when the bucket schedule and cross-sectional bandwidth line up, synchronization runs under useful computation instead of extending the critical path.

The architectural imperative is *bandwidth matching*: the communication volume of each algorithm must map inversely to the latency of the hardware interconnects. Chatty, blocking TP communication stays within the NVLink domain (600 GB/s or higher). Serialized, point-to-point PP transfers traverse the cluster spine at InfiniBand speeds. The massive but infrequent DP synchronization amortizes across the full training step. Attempting to run TP across racks, or DP without gradient accumulation, can violate this hierarchy and leave the 16,000-GPU fleet waiting for data to traverse the wire. This bandwidth-matching principle completes the feasibility check introduced in section 5.1.5.

5.7.2 Hybrid-parallelism worked example

Applying this bandwidth-matching principle to physical infrastructure transforms cluster design into a placement problem: each parallelism dimension must sit on the network tier that can tolerate its traffic. Tensor parallelism is the most latency-sensitive because it launches frequent AllReduce operations inside transformer blocks, so it belongs on the intra-node NVLink domain (600 GB/s–900 GB/s). Pipeline parallelism moves activation tensors only at stage boundaries, so it can span nearby nodes over InfiniBand (50 GB/s–100 GB/s). Data parallelism produces the largest payload, the gradient synchronization, but it occurs once per step and can be overlapped with backward computation. Memory capacity per device (80 GB–80 GB) sets the hard limit for every placement.

For a DGX A100-style deployment, this reasoning leads to a concrete layout. Fix tensor parallelism at $t = 8$ so the bandwidth-heavy matrix-multiplication collectives stay within one 8-GPU node. Map pipeline parallelism across nodes in the same high-bandwidth rack or island, often $p = 8$ or $p = 16$ depending on the memory footprint. Use data parallelism for the remaining scale-out dimension across pods. The result is not a generic rule but a consequence of matching each traffic pattern to the cheapest fabric tier that can carry it.

With the placement fixed, the next question is whether the static model state fits inside each accelerator's HBM budget.

The memory budget for training a 175B-parameter model is dominated by model states and requires aggressive sharding to fit within the 80 GB HBM capacity of H100-class accelerators. The FP16 weights alone consume approximately 350 GB ($175 \times 10^9 \times 2$ bytes). If we relied solely on Tensor Parallelism with 8-way sharding, each GPU would hold a 43.75 GB slice of the weights. However, the optimizer state presents a larger hurdle. In the simplified optimizer-state convention used for this 3D-parallelism budget, Adam's FP32 momentum and variance consume 8 bytes per parameter, adding about 1.4 TB globally; the fuller 12-byte accounting that also includes FP32 master weights would be about 2.1 TB. Even with 8-way tensor parallelism, the simplified combined weight and optimizer state would exceed 218.8 GB per GPU, causing an Out-Of-Memory (OOM) error. Consequently, we must employ Pipeline Parallelism (p) to further partition the model layers. With a hybrid configuration of tensor parallelism 8 and pipeline parallelism 16, the simplified static memory footprint drops to 13.7 GB per GPU (often budgeted as roughly 15 GB after framework buffers), leaving the remaining HBM available for the dynamic activation memory (A) generated during the forward pass, which scales linearly with micro-batch size and sequence length. Under the fuller 12-byte Adam convention, the same 8-way tensor, 16-stage pipeline split would be 19.1 GB per GPU.

Once the configuration fits in memory, the same layout has to satisfy the communication hierarchy that motivated it. Each parallelism dimension imposes a distinct traffic profile on the network. Tensor parallelism is the most chatty, requiring two `AllReduce` operations for every transformer block (one for the Attention projection, one for the MLP) in both the forward and backward passes. These messages are relatively small but occur thousands of times per step, making them strictly latency-bound and necessitating NVLink. Pipeline parallelism, in contrast, involves point-to-point transfers of activation tensors (size $B_\mu \times S \times d_{\text{model}}$) only at the boundaries of the pipeline stages. While these messages are moderate in size, they occur less frequently, making them manageable over standard InfiniBand links. Data parallelism generates the largest burst of traffic, requiring a global `AllReduce` of the entire 350 GB gradient buffer. However, this communication occurs only once per global batch update. By using gradient bucketing to overlap this transmission with the compute-intensive backward pass, the effective cost of DP communication can often be hidden, provided the cluster maintains sufficient cross-sectional bandwidth.

The last cost is not bandwidth but scheduling: a pipelined model is only efficient when enough micro-batches are in flight to keep all stages busy. This remaining cost is the pipeline bubble developed in section 5.5.2.2. Applying its bubble fraction $\frac{p-1}{m+p-1}$ to this hybrid layout, a GPT-175B configuration with $p = 16$ stages and $m = 32$ micro-batches wastes $\frac{15}{47} \approx 31.9\%$ of theoretical compute capacity, nearly one-third. The 1F1B scheduling that reclaims activation memory, and the larger micro-batch counts it makes practical, drive that fraction down asymptotically.

5.7.3 Blackwell-class scaling example

The Blackwell architecture provides a concrete example of how accelerator packaging changes the 3D parallelism trade-offs. First, NVLink 5 provides 1.8 TB/s of bidirectional bandwidth per

GPU, doubling the intra-node capacity of the Hopper generation. This allows for larger tensor parallelism (t) groups (for example, $t = 16$ or $t = 32$ across multiple nodes) with lower latency overhead. Second, Blackwell adds FP4 Tensor Core support, improving low-precision throughput and memory efficiency for supported workloads; whether FP4 is used for training depends on the numerical recipe and software stack.

For a 1 trillion parameter model, a Blackwell-class configuration can use $t = 16$ (spanning two 8-GPU nodes via NVLink Switch) and $p = 32$, reducing the total number of pipeline stages and associated bubble overhead relative to a 16,384-GPU Hopper-scale fleet reference configuration. The 10 TB/s die-to-die interconnect within the Blackwell GPU further collapses the distinction between intra-chip and intra-package communication, allowing the two reticle-limited dies to function as a single high-bandwidth tensor parallel unit. The general systems lesson is that improving local bandwidth shifts pressure outward: once intra-node communication is less binding, rail-optimized topologies (section 3.5.3) and All-to-All optimization become more important for larger Machine Learning Fleet configurations.

The same placement constraints can be made operational through design-space search.

Example 5.2: Automated design space search (Tier 3 optimizer)

Scenario: An engineering team needs to schedule Archetype A (a 175B parameter model) on a cluster of 8,192 H100 GPUs. Manually searching the 3D-parallelism space (TP $\times$ PP $\times$ DP) is error-prone: a split that maximizes DP might exceed the 80 GB HBM capacity, while a split that maximizes TP might saturate the NVLink interconnect.

Approach: Instead of trial and error, we invoke a constrained design-space search over TP, PP, and DP factorizations, implemented here as the Tier 3 Optimizer (`ParallelismOptimizer`) in our physics engine, configured to maximize Model FLOPs Utilization (MFU). The optimizer performs a constrained grid search over all valid algebraic factorizations and, in under a second, selects the best strategy under the stated objective:

- **Tensor parallelism (TP)**: 8 GPUs
- **Pipeline parallelism (PP)**: 8
- **Data parallelism (DP)**: 128

Systems insight: The optimizer formalizes what engineers discover empirically: TP should match the intra-node GPU count (8 GPUs) to avoid traversing the slower inter-node fabric, while the best pipeline depth under this objective is only PP=8 once tensor parallelism and sharded state make the memory budget fit. Deeper pipelines would reduce per-stage memory further, but they also increase the pipeline bubble and reduce MFU. This automated synthesis achieves a projected 25.9 MFU, demonstrating that empirical engineering heuristics often reflect structural mathematical laws.

Adjusting any one dimension shifts the pressure onto the other two, making the factorization TP $\times$ PP $\times$ DP a tightly coupled system rather than three independent knobs.

Checkpoint 5.4: Hybrid 3D parallelism

Verify your understanding of how parallelism strategies combine:

- ☐ **Dimension roles**: Identify which dimension in a **TP=8, PP=16, DP=128** configuration shards layers within a single node.
- ☐ **Bubble vs. depth**: Determine how moving from **PP=16** to **PP=8** changes the "Pipeline Bubble" fraction.
- ☐ **DP vs. PP**: Explain the trade-off, under a 1,024-GPU budget, between increasing **DP** (Data Parallelism) and increasing **PP** (Pipeline Parallelism).
- ☐ **Latency sensitivity**: Identify which dimension, **TP**, **PP**, or **DP**, is most sensitive to **inter-node latency**.

The MFU values need a historical utilization baseline to show how published systems approached 50 percent utilization. Figure 5.16 traces the evolution of Model FLOPs Utilization across published training systems from 2020 to 2024. The progression from GPT-3's 21 percent MFU to PaLM's 46 percent MFU reflects not improvements in raw hardware speed but advances in the parallelism strategies, communication overlap techniques, and scheduling optimizations discussed throughout this chapter. The plateau near 40–46 percent reveals that the theoretical ceiling imposed by communication overhead, pipeline bubbles, and memory management remains formidable even in highly optimized hybrid-parallel systems. Notably, Meta's Llama 3 training at 16,384 H100 GPUs achieved slightly lower MFU (41 percent) than the same model at 8,192 GPUs (43 percent), confirming that the scaling tax described in section 5.4.2 is not merely theoretical but measurable in large published runs.

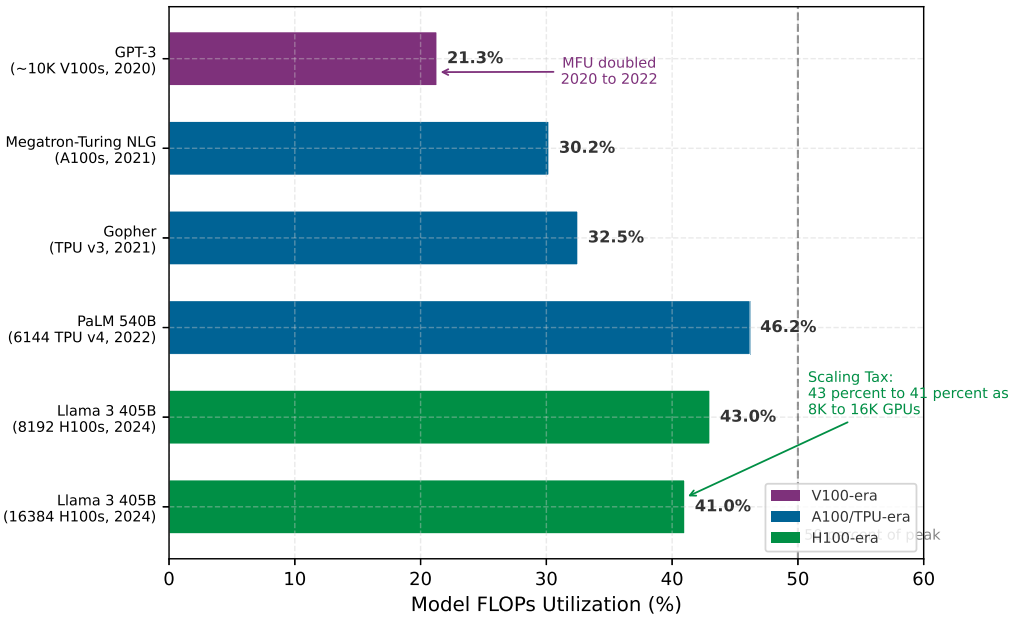


Figure 5.16: The MFU Progression: Model FLOPs Utilization across published training systems, ordered chronologically. MFU doubled from 21 percent (GPT-3, 2020) to 46 percent (PaLM, 2022) through advances in hybrid parallelism and communication overlap, then plateaued near 40–45 percent for large-scale runs. The Llama 3 data points illustrate the scaling tax: MFU drops from 43 percent at 8,192 GPUs to 41 percent at 16,384 GPUs as communication overhead grows with cluster size. Data sources: Chowdhery et al. (2022), Dubey et al. (2024).

5.8 Multi-Model Training: RLHF and Alignment

The parallelism strategies examined so far assume a single model being trained on a single objective. Reinforcement Learning from Human Feedback (RLHF) and its variants break this assumption by requiring multiple models to coordinate within a single training loop, each with different memory footprints, compute profiles, and gradient requirements. At a high level, RLHF generates model outputs, scores them with preference-derived reward signals, regularizes updates against a reference model, and then updates the policy; Proximal Policy Optimization (PPO) implements this online loop with separate policy, reference, reward, and value models, while Direct Preference Optimization (DPO)-style methods remove much of that rollout and value-model machinery. This creates a heterogeneous fleet management problem that cannot be solved by any single parallelism strategy and represents a qualitatively different distributed systems challenge from standard pretraining.

5.8.1 The multi-model coordination problem

Standard pretraining involves one model, one loss function, and one gradient stream. RLHF alignment, by contrast, orchestrates a system of models that interact during every training step. In

Proximal Policy Optimization (PPO) (Schulman et al. 2017), as used in InstructGPT-style RLHF systems (Ouyang et al. 2022), four distinct models must operate in concert. Table 5.7 separates those models by role, execution mode, and memory burden.

Table 5.7: RLHF Model Coordination Roles: PPO-style RLHF coordinates models with different execution modes and memory footprints, so placement decisions must separate trainable policy and value state from inference-only reference and reward state.

Model	Role in PPO-style RLHF	Execution mode	Memory burden
Policy model	The model being aligned and optimized.	Training mode with gradients, optimizer moments, and activations.	For a 70B parameter model in mixed precision with Adam, memory is approximately 70B $\times$ (2 + 2 + 12) = 1,120 GB, the same budget as standard pretraining.
Reference model	A frozen copy of the pretrained policy that supplies the KL-divergence penalty.	Inference mode only.	FP16/BF16 parameters without gradients or optimizer state; a 70B model needs about 70B $\times$ 2 = 140 GB, roughly 8$\times$ less than the training configuration.
Reward model	A preference-trained scorer that produces scalar rewards for generated sequences.	Inference mode only.	Often smaller than the policy; a 13B reward model requires approximately 26 GB in FP16, but it must process every generated sequence.
Value model	The PPO critic that estimates expected future reward.	Training mode, either separate or sharing a policy backbone.	A full-size separate critic can add another 1,120 GB; smaller models or shared-backbone designs reduce this to 200–400 GB.

The aggregate memory demand of the four-model PPO system dwarfs standard pretraining. A naive co-location of a 70B policy, 70B reference, 13B reward, and 13B value model requires approximately 1,494 GB of accelerator memory, before accounting for the KV caches and intermediate activations generated during sequence generation. On H100 GPUs with 80 GB of HBM each, this system requires a minimum of 19 GPUs for parameter storage alone. Once generation-phase KV caches (which grow linearly with output sequence length) and training-phase activations are included, the practical minimum rises to 64–128 GPUs for a single RLHF training instance.

5.8.2 Infrastructure asymmetry: Training vs. inference models

The defining infrastructure challenge of RLHF is not the *total memory footprint* but the *asymmetry* between the models' compute profiles. The policy and value models require full backward passes with gradient computation, activation checkpointing, and optimizer updates—compute-intensive operations that benefit from tensor parallelism and high arithmetic intensity. The reference and reward models, by contrast, perform only forward passes: they are inference workloads embedded within a training loop, with memory access patterns dominated by KV cache management rather than gradient accumulation.

The asymmetry creates a placement dilemma. Co-locating training and inference models on the same GPUs wastes compute during the generation phase (when the training models sit idle) and wastes memory during the gradient phase (when the inference models' parameter storage could be reclaimed for activations). Separating them onto dedicated GPU pools eliminates waste but introduces network latency for reward queries—each generated token batch must traverse the interconnect to reach the reward model and return a scalar signal before the policy gradient can be computed.

The generation phase itself introduces a sequential bottleneck absent from standard pretraining. RLHF requires the policy model to generate complete sequences (typically 256–2,048 tokens) autoregressively before computing rewards and policy gradients. Autoregressive generation is memory bandwidth bound, not compute bound: each token requires a full forward pass through the model to produce a single output token. For a Llama-2-70B-style grouped-query attention policy, the KV cache grows by approximately $2 \times N_L \times H_{\text{KV}} \times d_{\text{head}} \times 2$ bytes per generated token per sequence. With 80 layers, 8 KV heads, 128-dimensional heads, 1,024 generated tokens, and a batch of 256 prompts, the cache consumes 85.9 GB: $2 \times 80 \times 8 \times 128 \times 1024 \times 256 \times 2$ bytes. A dense multi-head attention model using the full 8,192-wide hidden state for K and V would yield 687.2 GB. Even with

GQA, the cache uses about one H100 GPU's worth of HBM for intermediate attention state that is discarded after reward computation.

RLHF systems address this asymmetry through temporal multiplexing. During the generation phase, the cluster behaves like an inference system: the policy model generates sequences while the reference model computes log-probabilities, and the dominant resource is HBM reserved for token-by-token attention state. During the training phase, the same fleet switches back to training mode: gradients are computed through the policy and value models using the 3D parallelism configuration developed earlier in the chapter. Serving systems handle that phase with request batching and KV-cache placement; here, the important point is the mode switch. RLHF is not one steady training loop, but an alternation between inference-shaped work and training-shaped work, and standard training frameworks rarely manage that transition by themselves.

5.8.3 DPO: Simplifying the fleet

Direct Preference Optimization (DPO) (Rafailov et al. 2023) eliminates the reward model and value model entirely by reformulating the alignment objective as a classification loss over preference pairs. Instead of generating sequences, computing rewards, and estimating advantages, DPO directly optimizes the policy to assign higher log-probability to preferred responses over dispreferred ones, using the reference model only to compute a KL-divergence regularization term.

The infrastructure implications are substantial. DPO reduces the multi-model system from four models to two: the policy model (training mode) and the reference model (inference mode). For the 70B policy with 13B reward and value models quantified below, static parameter memory drops from about 1,494 GB to 1,260 GB, a 16 percent reduction. If the PPO value model is policy-sized, the reduction is much larger: about 2,406 GB to 1,260 GB, or roughly 48 percent. DPO also eliminates the autoregressive generation phase entirely. Training operates on a fixed dataset of (prompt, preferred response, dispreferred response) triples, restoring the standard pretraining data pipeline: fixed-length sequences, deterministic batching, and no sequential token-by-token generation. The training loop becomes a standard supervised learning step with a modified loss function, amenable to the same 3D parallelism, gradient accumulation, and communication overlap techniques used for pretraining.

The trade-off is capability. DPO operates on a static preference dataset, meaning the policy cannot explore new responses and receive feedback during training. PPO's online generation allows the policy to improve iteratively on its own outputs, potentially discovering better strategies that the static dataset does not contain. For deployment scenarios where the preference data comprehensively covers the target distribution, DPO's infrastructure simplification dominates. For scenarios requiring adaptive exploration (training models to solve novel reasoning tasks, for example), PPO's online feedback loop may justify the 2× infrastructure overhead.

5.8.4 Quantitative analysis: PPO vs. DPO resource requirements

To make the infrastructure trade-off concrete, consider aligning a 70B policy model on a cluster of 256 H100 GPUs (80 GB HBM each).

Napkin Math 5.9: RLHF infrastructure budget: PPO vs. DPO

Problem: Aligning a 70B policy model requires co-locating it with a 70B reference model and, for PPO, a 13B reward model and a 13B value model. Under 3D parallelism (TP=8 GPUs, PP=4, DP=8), how much per-GPU memory does each approach consume, and how much headroom does DPO free by eliminating the reward model, value model, and generation-phase KV cache?

PPO memory budget (per-GPU, with TP=8 GPUs, PP=4, DP=8 for policy)

The policy model under 3D parallelism with TP=8 GPUs and PP=4 distributes its 1,120 GB training state across 32 GPUs, yielding 35 GB per GPU. The reference model, requiring only 140 GB for inference, can be sharded across a separate pool of 8 GPUs at 17.5 GB each, or co-located with the policy GPUs at an additional 4.4 GB per GPU (140 GB / 32 GPUs). The 13B reward model adds 26 GB shared across its pool. The 13B value model in training mode adds approximately 208 GB (13B × 16 bytes/param) across its pool.

Total static memory (co-located policy + reference on 32 GPUs): 35 GB + 4.4 GB ≈ 39.4 GB per GPU, leaving ∼ 40.6 GB for activations and KV cache. With generation-phase KV caches for a rollout batch containing 256 sequences and 1,024 tokens inside one $TP \times PP$ policy replica, each policy GPU must reserve an additional 85.9 GB/32 GPUs ≈ 2.7 GB for its KV cache shard. The remaining ∼ 37.9 GB constrains the micro-batch size during the training phase.

DPO Memory Budget (same parallelism configuration)

The policy model uses the same 35 GB per GPU. The reference model adds 4.4 GB per GPU (co-located). No reward model, no value model, no KV cache for generation.

On the policy GPUs, total static memory is 35 GB + 4.4 GB = 39.4 GB per GPU, identical to PPO's co-located policy/reference footprint. Cluster-wide, however, DPO avoids the PPO reward and value model memory, and it also removes the generation-phase KV cache burden. The full ∼ 40.6 GB remaining on each policy GPU is available for training activations, permitting modestly larger micro-batches (about 1.07× under this co-located policy/reference memory budget) or reducing the need for activation checkpointing that PPO requires.

Throughput Comparison

PPO's two-phase design (generate then train) introduces a fundamental throughput penalty. If generation consumes 60 percent of the step time (typical for autoregressive decoding of long sequences), the training hardware achieves only 40 percent utilization during an RLHF step. DPO, operating as a standard training loop, achieves the same 40 percent–55 percent MFU as pretraining. Table 5.8 compares the two regimes side by side.

Table 5.8: PPO vs. DPO Memory and MFU: Per-GPU memory and effective training MFU for PPO and DPO on the same accelerator budget.

Metric	PPO (70B policy)	DPO (70B policy)
Models required	4 (policy, ref, reward, value)	2 (policy, reference)
Total parameter memory	∼ 1,494 GB	∼ 1,260 GB
Minimum GPUs (memory)	64–128 GPUs	32–64 GPUs
Generation phase	Yes (sequential, BW-bound)	None
Effective training MFU	15–25%	40–55%
Data pipeline	Online generation	Static preference pairs

Systems insight: In this 13B reward/value-model scenario, DPO reduces static parameter memory by 15.7 percent and doubles the effective compute utilization by eliminating the reward model, value model, and autoregressive generation phase. The choice between PPO and DPO is not only a modeling preference but an infrastructure constraint: organizations with limited GPU budgets may prefer DPO because it restores the workload to a fixed-data training loop. If the PPO value model is policy-sized rather than 13B, the memory reduction approaches the larger 2,406 GB to 1,260 GB comparison discussed earlier.

5.8.5 Sequence length variance and batching challenges

Both PPO and DPO face a data engineering challenge absent from standard pretraining: extreme variance in sequence length. Pretraining typically uses fixed-length sequences (2,048 or 4,096 tokens, padded or packed to fill each position), enabling uniform batch shapes and predictable memory consumption. RLHF training data consists of variable-length prompts (10–500 tokens) concatenated with variable-length completions (50–2,048 tokens), producing sequence lengths with 10–50× variance within a single batch.

Length variance creates two interacting problems. Fixed-size batching pads all sequences to the maximum length in the batch, wasting compute on padding tokens. A batch containing one 2,048-token sequence and fifteen 128-token sequences wastes 88 percent of the compute on padding.

Dynamic batching groups sequences by similar length to minimize padding, but introduces load imbalance across data-parallel workers: one worker may receive a batch of long sequences consuming 60 GB of activation memory while another processes short sequences using only 8 GB, causing the short-sequence worker to stall at the synchronization barrier while the long-sequence worker completes.

RLHF systems mitigate this through a combination of sequence packing (concatenating multiple short sequences into a single fixed-length input with attention masking to prevent cross-contamination) and adaptive micro-batching (dynamically adjusting the number of sequences per micro-batch based on the aggregate token count rather than the sequence count). These techniques can recover much of the compute lost to padding variance but add complexity to the data pipeline that standard pretraining frameworks do not provide. The interaction between variable-length data and distributed synchronization remains an active area of systems engineering research, because every worker must process the same number of tokens per step to maintain gradient consistency.

RLHF is therefore not a detour from parallelism strategy; it is the case that shows why no strategy is sufficient by itself. The policy model may need tensor and pipeline parallelism, the reference and reward models behave like inference services, the value model may require training state, and the rollout data pipeline changes shape from step to step. With that stress test in view, the final comparison can return to the general engineering question: identify the binding constraint first, then choose the parallelism pattern that moves the least data while fitting the required state.

5.9 Parallelism Strategy Comparison

A parallelism strategy is useful only if it moves the binding constraint without creating a larger one. The feasibility filter previewed by the decision tree in section 5.1.5 named those constraints; now that tensor, pipeline, and hybrid parallelism have been developed, the same filter can be stated as engineering signals rather than a generic ranking. For a new 50-billion parameter model, the decisive questions are which memory state fits, how often communication occurs, where the pipeline bubble appears, and which abstraction cost the engineering team can tolerate. Table 5.9 turns those questions into engineering signals.

Table 5.9: Parallel Training Strategies: Parallelism strategy selection is an engineering feasibility test: identify the binding constraint, the communication primitive it creates, the placement decision it forces, and the measurement that proves the strategy is helping.

Strategy	Binding constraint	Primary movement	Placement decision	Validation signal
DDP	Model state fits on one accelerator	AllReduce gradients every step	Replicate full model across data-parallel workers	Step time dominated by compute, not synchronization
FSDP/ZeRO	Model state exceeds one accelerator	Shard and gather parameters or optimizer state	Place shards to minimize gather and reduce-scatter cost	Peak memory remains below device budget
Tensor parallelism	One layer or tensor operation exceeds local capacity	AllReduce or AllGather within layer blocks	Keep tightly coupled partitions on fast intra-node links	Layer latency stays below activation handoff cost
Pipeline parallelism	Layer stack fits only when split by stage	Activations and gradients between stages	Balance stages so bubbles are smaller than useful work	Bubble fraction falls as microbatch count increases
Hybrid parallelism	More than one constraint binds at once	Multiple collectives on different axes	Align data, tensor, pipeline, and shard groups to topology	MFU improves without exceeding memory or network budget

Figure 5.17 converts the same filter into a decision tree. It begins with memory fit because capacity failure is nonnegotiable, then asks whether data scale alone justifies replication. The tree is intentionally simplified; hardware heterogeneity, communication bandwidth, and workload imbalance enter as second-order checks after the binding constraint has been identified.

The table and flowchart are intentionally coarse, but their purpose is precise: identify the binding constraint before choosing the implementation stack.

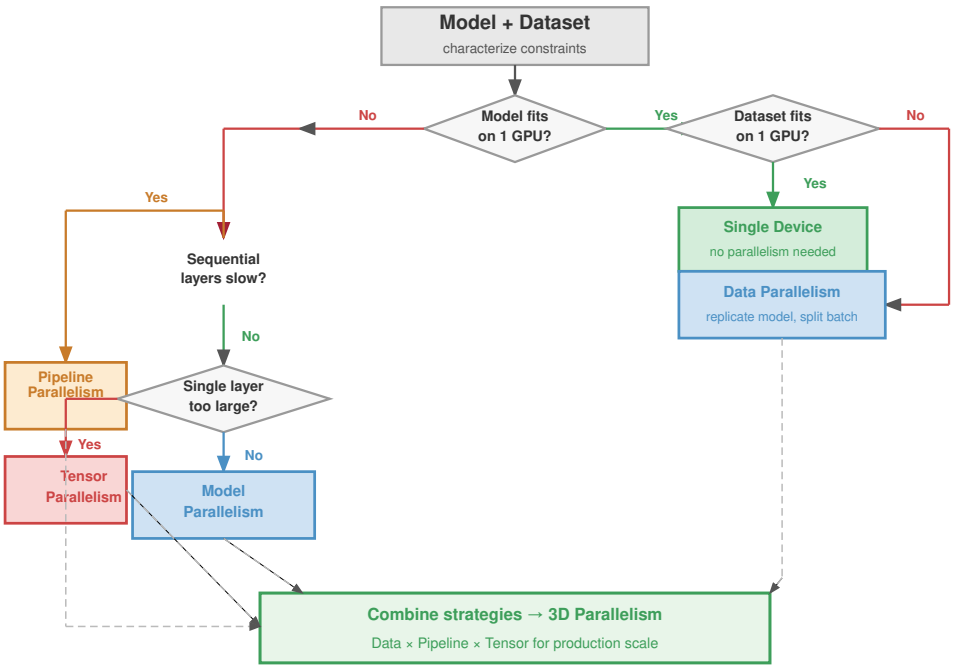


Figure 5.17: Parallelism Strategy Decision Tree: A systematic selection guide based on two key questions: Does the model state fit in single-device memory, and does the required data throughput justify multiple replicas? Models exceeding device memory require model parallelism or sharding; workloads whose model fits but whose throughput target exceeds one device benefit from data parallelism; significant constraints in both dimensions demand hybrid approaches. While simplified, this framework captures the primary decision points before practitioners must consider secondary factors like hardware heterogeneity and workload imbalance.

5.10 From Principles to Systems

Choosing DDP, FSDP/ZeRO-style sharding, tensor parallelism, or pipeline parallelism is a decision about which constraints the runtime owns and which constraints the engineer must manage directly. The parallelism strategies examined throughout this chapter (gradient averaging, AllReduce synchronization, tensor splitting, pipeline scheduling) translate into deployed training systems through this layered abstraction hierarchy.

At the data-parallel layer, the simplest distributed training abstraction wraps a model so that the framework automatically replicates it across available accelerators, splits each batch, and averages gradients after the backward pass. Historically, parameter-server systems organized this work with server nodes that store shared parameters while workers push and pull updates (Li et al. 2014). As section 5.5.3 established, that design concentrates gradient bandwidth at the server tier, so dense synchronous training favors the decentralized collectives below.

Large-scale data parallelism eliminates this bottleneck by replacing the central server with decentralized AllReduce. Each worker participates symmetrically in the reduction: every device both sends and receives gradient chunks at its own link rate, distributing the bandwidth load across all nodes rather than concentrating it. The framework initializes a process group that maps workers to the physical topology, selects an appropriate collective algorithm (ring, tree, or hierarchical) based on the detected interconnect, and inserts gradient synchronization hooks into the backward pass automatically. Gradient bucketing further improves efficiency by grouping small tensors into larger messages before transmission, and computation-communication overlap allows the AllReduce for early layers to proceed while later layers are still computing gradients. These optimizations can reach high parallel efficiency at moderate scale—not because the API is simple, but because the underlying runtime makes topology-aware decisions that a manual implementation would otherwise need to reproduce.

Model and pipeline parallelism require a fundamentally different abstraction because the framework must manage cross-device tensor placement and the sequential data flow between partitions. The core principle is explicit device assignment: the engineer specifies which layers reside on which devices, and the framework handles the activation transfers between devices during the forward pass and the gradient flow in reverse during backpropagation. Explicit device assignment makes the sequential dependencies of model parallelism visible, since each downstream device must wait for its upstream neighbor to complete. The engineer must reason about pipeline bubbles and load balance at the architecture level rather than hiding them behind an opaque wrapper.

For large-scale model parallelism, the key architectural insight is that tensor splitting and pipeline scheduling require different levels of framework support. Tensor parallelism replaces standard linear layers with column-parallel and row-parallel variants that automatically insert AllReduce operations at the correct points in the transformer computation graph, as described in section 5.5.2.3. Pipeline parallelism adds microbatch scheduling logic that interleaves forward and backward passes across stages to minimize bubble overhead. Memory-efficient sharding integrates ZeRO-3 style parameter partitioning by wrapping model layers with automatic AllGather operations before each forward pass and ReduceScatter after each backward pass. The critical design decision is which abstraction levels to compose: pure data parallelism suffices when the model fits in memory, memory-efficient sharding extends data parallelism to memory-constrained regimes, and full tensor or pipeline parallelism becomes necessary when individual layers or the full model depth exceed single-device capacity.

Across these abstraction layers, distributed training ultimately reduces to a small set of collective communication primitives. Rather than exploring the network mechanics of how these primitives physically route data across the cluster, which is extensively covered in Chapter 6, framework abstractions focus on *when* and *why* these primitives are invoked.

The primitives compose into the communication patterns that define each parallelism strategy. Data parallelism uses one AllReduce per training step. Full-shard FSDP with resharding uses about $3N_L$ collectives per step (two AllGathers and one ReduceScatter for each of N_L layers). Tensor parallelism uses 2 AllReduce operations per transformer block on the critical path. The choice of primitive, its message size, and its frequency relative to computation determine whether the system operates in the compute-bound or communication-bound regime. *No framework abstraction changes the underlying physics*: the efficiency of distributed training ultimately depends on physical interconnect bandwidth, memory capacity, and synchronization latency.

5.11 Fallacies and Pitfalls

Many engineering teams who scale cluster capacity by $4\times$ find that their training iterations take longer to complete, reflecting a fundamental misunderstanding of distributed training physics. The following misconceptions capture errors that waste compute resources and delay research.

Fallacy: *Linear speedup is achievable with sufficient engineering effort.*

Amdahl's Law establishes hard limits: any sequential component bounds maximum speedup regardless of parallelism. In distributed training, gradient synchronization is inherently sequential since all gradients must be collected before any update proceeds. As section 5.4 demonstrates, the scaling efficiency equation $\eta_{\text{scaling}} = 1/(1 + N(T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}})/T_{\text{compute}})$ reveals how communication and synchronization overhead dominate as N increases. Even with perfect overlap and ideal algorithms, communication overhead grows with cluster size. For data parallelism, AllReduce time increases logarithmically with tree algorithms or linearly in the latency term with ring algorithms as GPU count grows. A 1000-GPU cluster will never train $1000\times$ faster than a single GPU; achieving $500\times$ speedup would be exceptional, and 100–$200\times$ is more typical for communication-heavy workloads. Organizations that budget projects assuming linear scaling inevitably miss deadlines and overspend on compute.

Pitfall: *Optimizing MFU without considering scaling efficiency.*

A team achieves 50 percent MFU on a single node, scales to 1,024 GPUs, and expects 512 GPU-equivalents of useful work. Scaling efficiency at 1,024 GPUs can be near 50 percent for communication-heavy workloads, so actual useful throughput is roughly $0.50 \times 0.50 = 0.25$ of peak, or 256 GPU-equivalents: half the expected value. MFU and scaling efficiency are independent multiplicative factors. Optimizing one without measuring the other produces capacity estimates

that are off by 2× or more. Capacity planning at fleet scale requires reporting both metrics together and tracking their product as "useful goodput."

Fallacy: *Hyperparameters tuned on small clusters transfer directly to large-scale training.*

Engineers tune hyperparameters on 8-GPU workstations then deploy to 256-GPU clusters expecting identical behavior. At scale, convergence patterns change fundamentally. The most critical hyperparameter is learning rate: as section 5.3 explains, batch size increases proportionally with GPU count in data parallelism, requiring learning rate adjustments. The "linear scaling rule"[10] (Goyal et al. 2017) suggests $\eta_{\text{large}} = \eta_{\text{base}} \times (B_{\text{large}}/B_{\text{base}})$, but this relationship holds only within bounds. As models scale, they eventually encounter the critical batch size[11], where adding more data per step yields diminishing returns in convergence.

Beyond the critical batch size, this relationship breaks down in a model-, data-, and optimizer-dependent way. A team that takes a small-cluster learning rate, multiplies it mechanically with GPU count, and jumps directly to a much larger global batch may see lower final quality or slower sample efficiency even though hardware throughput improved. Warmup schedules, weight decay adjustment, layer-wise optimizers such as LARS/LAMB, and careful momentum tuning can recover accuracy in some regimes, but require systematic experimentation at target scale. Organizations that skip these scaling studies waste thousands of GPU-hours on suboptimal runs.

Pitfall: *Adding GPUs to data-parallel jobs without modeling communication.*

Engineers assume more GPUs always accelerate training. At scale, statistical efficiency limits overwhelm hardware gains. As section 5.3 establishes, data parallelism increases effective batch size proportionally with GPU count ($B_{\text{total}} = N \times B_{\text{local}}$), but gradient quality grows sublinearly beyond model-specific thresholds. A 100K-sample batch may provide only 2× the gradient information of a 10K-sample batch, not 10×, because samples become redundant within the loss landscape. The critical batch size defines where marginal returns collapse: the chapter's earlier examples put ResNet-50 around 8K–16K and BERT-Large around 32K–65K, with exact thresholds depending on optimizer, schedule, and target quality. Beyond this threshold, doubling GPU count doubles cost but provides minimal convergence acceleration. In a representative cost model, a 1024-GPU run that converges in 18 hours at $45,000 compute cost may be worse than a 512-GPU run that converges in 19 hours at $22,000, demonstrating how exceeding critical batch size wastes resources without meaningful time savings.

Fallacy: *Memory capacity alone determines the parallelism strategy.*

Engineers see that a 70B model exceeds 80 GB memory and immediately choose tensor parallelism or pipeline parallelism to split weights. In a deployed run, the effective strategy depends on the interaction between memory pressure, computation patterns, and communication topology. As section 5.9 explains, tensor parallelism splits each layer across devices with AllReduce synchronization per layer, achieving even memory distribution but placing communication on the critical path. Pipeline parallelism assigns complete layers to stages with point-to-point transfers between stages, reducing per-step communication but introducing pipeline bubble overhead that wastes 10–30 percent of cycles. For a 175B model on 64 A100 GPUs where tensor parallelism degree-8 enables training, pipeline parallelism with 8 stages achieves 23 percent higher throughput due to reduced all-to-all communication despite similar memory footprints. The decision requires profiling communication patterns and bubble overhead, not just checking if weights fit in memory.

Pitfall: *Applying FSDP or ZeRO without measuring efficiency trade-offs.*

Engineers adopt FSDP universally after reading that it "reduces memory and enables larger models". In a deployed run, sharding introduces 10–25 percent communication overhead that only pays off when memory pressure justifies it. FSDP reduces memory footprint by sharding optimizer state, gradients, and optionally parameters across GPUs, but requires AllGather operations before each forward pass and ReduceScatter after backward pass. For a 7B model on A100-80 GB with batch size 4, standard DDP achieves 145 samples/second while FSDP achieves only 118 samples/second (19 percent slower) because the model fits comfortably without sharding and the added communication overhead provides no benefit. FSDP provides value when model plus optimizer state exceeds single-GPU memory, when enabling larger per-GPU batch sizes justifies the overhead, or when ZeRO-Offload to CPU memory extends capacity. A 65B model that cannot fit on 80 GB becomes trainable with FSDP ZeRO-3, accepting 15 percent throughput loss to enable training at all. Applying FSDP universally without measuring memory pressure wastes performance.

10 **Linear Scaling Rule**: Established by Goyal et al. (2017) at Facebook AI Research, who trained ResNet-50 on ImageNet in one hour across 256 GPUs with a batch size of 8,192 while matching small-batch accuracy. The rule – multiply learning rate by k when batch size increases by k – works only while the large-batch approximation remains valid. Above the workload's measured critical-batch-size regime, gradient noise drops below the useful signal scale and additional parallelism yields diminishing convergence returns (McCandlish et al. 2018; Shallue et al. 2019).

11 **Critical Batch Size**: The gradient-noise-scale view introduced by McCandlish et al. treats the largest useful batch size as a measurable statistic that varies by domain, model, optimizer, and training phase (McCandlish et al. 2018). Scaling beyond that measured point improves hardware throughput but does not proportionally reduce the number of optimization steps needed to reach target quality, collapsing the scaling efficiency (η_{scaling}).

Fallacy: *Parallelism overhead is roughly constant regardless of model size.*

Engineers benchmark parallelism strategies on convenient small models then apply conclusions to large-scale training. At scale, the ratio between computation and communication time changes dramatically with model size, inverting strategic decisions. AllReduce communication time depends primarily on gradient tensor size and network bandwidth, growing roughly linearly with parameter count, while forward and backward pass computation time grows superlinearly due to larger matrix operations. For a 1B parameter model where forward/backward pass takes 50 ms and AllReduce takes 25 ms, communication overhead consumes 33 percent of step time. For a 70B parameter model where forward/backward takes 2400 ms and AllReduce takes 180 ms, communication overhead drops to 7 percent despite the gradient size being 70$\times$ larger. Decisions made on small models ("pipeline parallelism's 15 percent bubble overhead makes it always slower than data parallelism") can invert at scale where data parallelism's communication overhead reaches 25–40 percent. Reliable strategy selection requires either profiling at target scale or analytical models that account for how computation scales as $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$ while communication scales as $\mathcal{O}(n)$.

Pitfall: *Treating scaling efficiency as a property of the hardware alone.*

Vendor benchmarks publish "85 percent scaling efficiency at 1,024 GPUs" and procurement teams plan capacity as if that number is a cluster constant. Scaling efficiency is a joint property of four things: the workload's communication-to-computation ratio, the parallelism strategy (data, pipeline, tensor, hybrid), the network topology (fat-tree vs. torus vs. dragonfly), and the software stack (NCCL version, kernel fusion, overlap quality). Reference numbers from large-language-model training benchmarks may overstate scaling for graph neural networks with irregular communication, and they may understate scaling for embedding-heavy recommendation models with mostly local computation. Capacity planning should use measured scaling on the target workload, not a vendor hero benchmark.

Fallacy: *Gradient accumulation is free.*

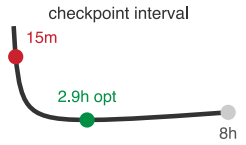
Engineers use gradient accumulation to simulate larger batch sizes, reducing synchronization frequency from every step to every K steps. The technique appears cost-free since it eliminates $(K-1)/K$ of synchronization events. In a real training loop, accumulation introduces update latency, memory-residency, and numerical precision risks. Standard implementations accumulate into one resident gradient buffer rather than allocating a separate full gradient tensor per microstep; for a 7B model, that FP16 gradient buffer is still 14 GB and must remain live throughout the accumulation window. If the implementation retains computation graphs, overlaps multiple outstanding micro-batches, or increases local microbatch size, activation memory can grow further and exhaust HBM. Effective optimizer-step latency increases proportionally with accumulation steps, so accumulating 8 steps means optimizer updates occur 8$\times$ less frequently, potentially slowing convergence despite higher throughput. Most critically, accumulated FP16 gradients risk overflow when summing hundreds of gradient tensors, particularly in early training when loss values are large. A team training a transformer model with 16-step gradient accumulation in FP16 experienced loss spikes and divergence at step 1200; switching to 4-step accumulation with more frequent synchronization resolved the instability despite higher communication costs. Gradient accumulation trades communication frequency for update latency, memory residency, and numerical stability.

Pitfall: *Using fixed checkpoint intervals regardless of system characteristics.*

Engineers checkpoint distributed training "every hour" or "every 1000 steps" based on intuition rather than analysis. Chapter 7 derives the Young-Daly checkpoint law (principle 12); the training lesson here is that checkpoint cadence is not a constant. It depends on the mathematical relationship between checkpoint write cost and failure rate.

The formula sets the optimal checkpoint interval to $\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$, where T_{write} is checkpoint write time and $\text{MTBF}_{\text{system}}$ is mean time between system failures. For a 1024-GPU cluster at the canonical 50,000-hour per-GPU MTBF, the cluster-level $\text{MTBF}_{\text{system}}$ is 48.8 hours ($\text{MTBF}_{\text{GPU}}/N$). With a 5-minute checkpoint time, the optimal interval is approximately 171.2 minutes (~2.9 hours), with an unavoidable checkpoint-plus-rework tax of 5.8 percent. Checkpointing every 15 minutes "to be safe" raises the total tax to 33.6 percent, while checkpointing every 8 hours risks losing significant work on failure. For larger models where checkpoint time increases to 15 minutes due to model size and storage bandwidth, the optimal interval shifts again. The cost of guessing scales with the

cluster: a 1024-GPU run loses approximately $13,638 per day to excessive checkpointing when it uses a 15-minute interval instead of the Young-Daly optimum.



The optimal checkpoint cadence balances write overhead against rework risk.

5.12 Summary

This chapter introduced the "scaling wall": the point where adding more GPUs eventually makes training slower rather than faster. Distributed training is not a simple hardware problem; it is a constraint satisfaction problem governed by the interaction between model size, batch size, and interconnect bandwidth.

The 3D Parallelism Cube (figure 5.18) is a useful framework for scaling large models. Data parallelism unrolls the outer loop of training to scale throughput; tensor parallelism vectorizes the inner loops of matrix multiplication to fit memory; and pipeline parallelism stages sequential layers to reduce communication frequency. Together, these strategies map models larger than any single memory bank onto a fleet of accelerators with finite bandwidth.

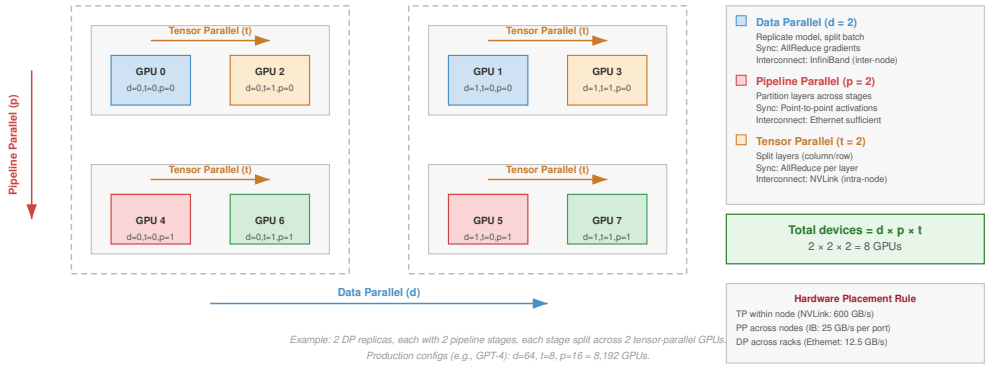


Figure 5.18: 3D Parallelism Strategy Space: Concrete 8-GPU configuration filling out a 3D-parallel coordinate grid. Each GPU is placed at a (d, p, t) coordinate combining Data Parallelism (d — replicate models across workers), Pipeline Parallelism (p — partition layers vertically across stages), and Tensor Parallelism (t — split individual operations horizontally within a layer). A single training step coordinates $d \times p \times t$ GPUs, with tensor-parallel communication within layers, pipeline communication at microbatch stage boundaries, and data-parallel synchronization once per optimizer step.

Ultimately, the choice of parallelism is a loop transformation applied by the cluster-level compiler. By matching logical communication patterns to physical hardware hierarchies, the system moves from the "linear scaling regime" of small clusters to the "communication-bound" reality of very large accelerator fleets.

Throughout this chapter, the partitioning strategies were developed directly for dense large-model training and recommendation-style workloads. Table 5.10 also shows how the same constraint logic reaches the federated edge case, where the bottleneck shifts from accelerator memory and fabric bandwidth to unreliable devices, privacy, and intermittent connectivity.

Lighthouse 5.3: Distributed archetype spectrum

The operating point in the 3D Parallelism Cube shifts depending on the system's primary bottleneck:

Table 5.10: Partitioning strategy by archetype: The operating point in the 3D-parallelism cube shifts with the dominant bottleneck.

Archetype	Primary Partitioning Strategy	The Logic
Archetype A (GPT-4/Llama-3)	Hybrid 3D Parallelism	Combine Tensor (width), Pipeline (depth), and Data (throughput) to fit 3.5 TB of weights.
Archetype B (DLRM at Scale)	Embedding Sharding	Partition massive 10 TB+ tables across a Parameter Server fleet; use sparse AllToAll updates.
Archetype C (Federated MobileNet)	Federated Learning	The same logic extends to coordinating on-device MobileNet updates across unreliable, privacy-constrained edge devices; Chapter 11 develops this regime.

The parallelism strategies explored throughout this chapter (data, tensor, pipeline, and expert) provide the conceptual toolkit for partitioning any training workload across a cluster. The key insight is that these strategies are not mutually exclusive alternatives but complementary dimensions of a unified optimization space. Systems such as Megatron-LM achieve efficient scaling precisely because they combine multiple strategies, using tensor parallelism within nodes, pipeline parallelism across node groups, data parallelism for throughput, and expert parallelism for capacity scaling.

Key Takeaways: Parallelism relocates the tax

- **Splitting work moves the bottleneck**: Distributed training does not make the underlying computation smaller; it converts memory pressure into communication volume, synchronization delay, or idle pipeline time. The winning strategy is the split that sends overhead to the least binding part of the fleet.
- **Data parallelism ends at convergence**: Replicas scale throughput cleanly only while larger global batches still improve optimization. Past the critical batch size, AllReduce cost and reduced gradient noise turn "more workers" into slower or less stable learning unless schedules, warmup, and accumulation change.
- **Sharding buys memory with messages**: ZeRO and FSDP make 100B+ parameter models feasible by partitioning optimizer state, gradients, and parameters. The price is a stricter communication schedule of ReduceScatter and AllGather operations that must be overlapped or hidden.
- **Tensor parallelism belongs near NVLink**: Tensor parallelism splits matrix operations inside layers, so it needs the high-bandwidth intra-node fabric that A100 and H100 NVLink provide. Stretching that traffic across racks turns a memory-capacity solution into a communication bottleneck.
- **Pipeline parallelism trades bytes for bubbles**: Layer staging reduces communication frequency, but fill and drain slots leave accelerators idle. Microbatching with $m \gg p$ is the mechanism that turns model depth into throughput rather than pipeline slack.
- **The 3D cube is a hardware map**: Real frontier training combines data, tensor, pipeline, expert, and sharded parallelism because no single axis fits the model and the fleet. Logical groups must map to HBM, NVLink, InfiniBand, and failure domains together.

Beneath the cube and its three axes lies a single trade. No way of splitting a model makes the underlying work smaller; each only moves it, turning a memory limit into a communication cost or a communication cost into idle time. Data, tensor, and pipeline parallelism spend communication to win the memory headroom no single accelerator has, and coordination is the tax paid to keep the split consistent. This is displacement of overhead, the law the rest of the volume turns on: the execution tax of scale cannot be removed, only relocated among compute, communication, and

coordination. The cube is the map of where the tax can be sent; the engineering is choosing the destination that binds least.

What's Next: From logic to traffic

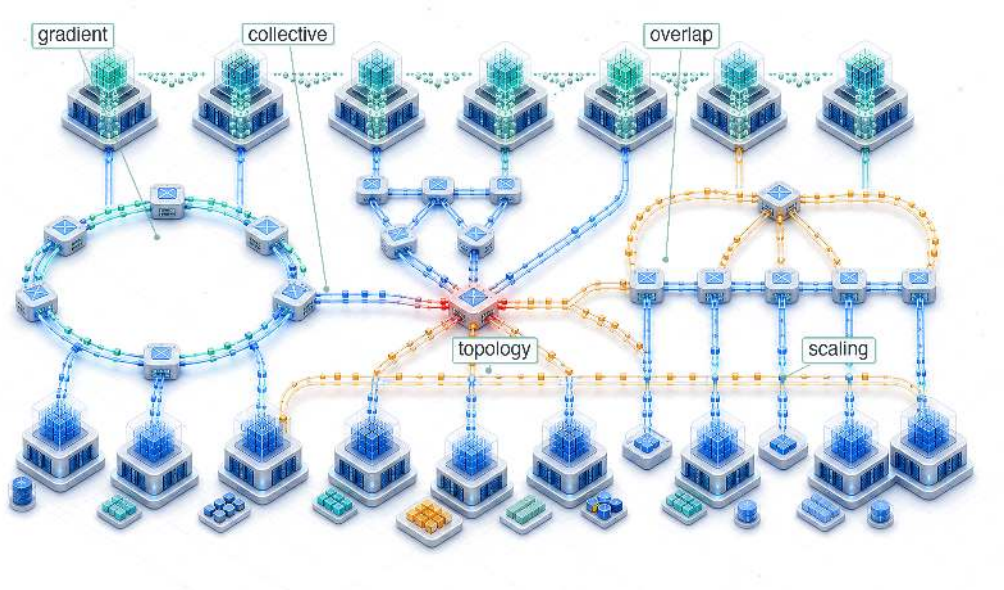
We have defined the *logical* traffic patterns of the Machine Learning Fleet, the *how* of splitting the math. These logical patterns, however, eventually hit the *physical* wires of the data center. In Chapter 6, we open the black box of the collective operations (AllReduce, AllToAll) that make this consistency possible. We move from the logic of *what* to send to the mechanics of *how* to route it through rings, trees, and rails.

Research Questions: For further inquiry

- How should an engineer choose among data, tensor, pipeline, expert, and sharded parallelism when adding accelerators begins to reduce scaling efficiency?
- When does increasing global batch size stop improving time-to-quality, even if more hardware is available?
- How should logical parallel groups map onto HBM, NVLink, InfiniBand, storage, and failure domains?
- How much extra communication should a memory-sharding design accept before it stops improving time-to-train?

6

Collective Communication

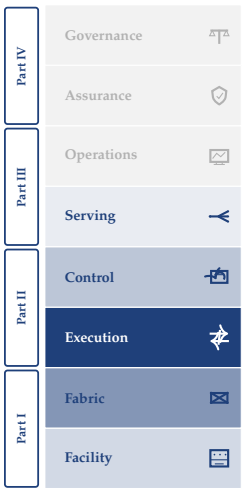


6.1 From Parallelism to Communication Patterns
6.2 Mapping the Terrain: Network Performance Modeling
6.3 Choosing the Vehicle: Collective Operation Primitives
6.4 Engineering the Flow: AllReduce Algorithms
6.5 Hierarchical Communication
6.6 Gradient Compression Under Bandwidth Scarcity
6.7 The Communication Library Landscape
6.8 Communication-Computation Overlap
6.9 Fallacies and Pitfalls
6.10 Summary

Purpose

Why does communication between machines become the fundamental constraint that governs large-scale machine learning systems?

Computation scales by adding processors; communication scales by moving data between them. These scale differently: adding a processor increases aggregate compute linearly, but coordinating that processor with all others moves more total data and, for the most general synchronization patterns, multiplies the logical connections among workers quadratically. At sufficient scale, the time spent exchanging gradients, activations, and parameters exceeds the time spent computing them. This crossover point is not a bug to be fixed but a fundamental property of distributed systems that determines which parallelization strategies work, which model sizes are trainable, and which organizations can operate at the largest scales. The physics of light-speed delays, bandwidth limits, and energy costs of data movement constrains communication as firmly as transistor physics constrains computation, yet communication is far less intuitive to reason about, making it the hidden bottleneck that undermines systems designed by those who understand only the compute side. In C^3 terms, collective communication forms the instruction set of the fleet: the physical operations that govern how compute is penalized by communication.



Learning Objectives

- Apply communication cost models to bound collective latency, bandwidth, and message-size crossover points
- Match collective communication primitives to parallelism strategies and model archetypes
- Compare ring, tree, butterfly, double-tree, and hierarchical reduction algorithms using scale-dependent cost models
- Analyze communication-library reality gaps by comparing theoretical costs with measured latency, bandwidth, and topology mapping
- Design topology-aware collective schedules for NVLink, InfiniBand, rail-optimized, torus, and in-network reduction fabrics
- Evaluate compression, sparsification, and error feedback by balancing communication savings against convergence risk
- Implement overlap strategies with bucket fusion, asynchronous operations, and layer-by-layer gradient scheduling

6.1 From Parallelism to Communication Patterns

When 10,000 GPUs need to apply one weight update, the splits created by data, tensor, pipeline, and expert parallelism have to be made consistent again. The workers are not sending arbitrary messages; they are executing choreographed collective operations. In the fleet stack shown in figure 1.13, those operations sit in the Distribution Layer, where logical parallelism becomes physical traffic.

The 3D Parallelism Cube assumes that replicas, shards, stages, and experts can exchange gradients, activations, parameters, and tokens quickly enough for the optimizer to see one coherent training run. That assumption is the handoff from parallelism to communication: every way of splitting the model relocates work onto the network.

The gap between that assumption and the wire reveals a fundamental asymmetry in *how* computation and communication scale. Computation is local: each GPU works on its own data, so aggregate compute grows with the number of GPUs. Communication is global: keeping the fleet synchronized requires information to cross physical links with finite latency, bandwidth, topology, and energy. The central case is gradient synchronization; from there the mechanics become concrete through alpha-beta cost models, collective primitives, AllReduce schedules, topology-aware routing, compression, and overlap.

Definition 6.1: Gradient synchronization

Gradient Synchronization is the collective communication step in synchronous data-parallel training in which every worker contributes its locally computed gradient tensor to an aggregate reduction, then receives the same reduced result so all model replicas apply an identical update.

1. **Significance**: A 70B-parameter model in BF16 generates 140 GB of gradient data per worker per step. Synchronizing across 1,000 GPUs via ring AllReduce at 50 GB/s per link requires approximately $2 \times 140/50 \approx 5.6$ s of communication per step, making the bandwidth (data-movement) term large enough to dominate the iron law unless overlap, hierarchy, or compression reduces the exposed transfer.
2. **Distinction**: Unlike parameter-server approaches (where all workers send gradients to a centralized aggregator whose bandwidth scales as $\mathcal{O}(N)$), ring AllReduce distributes the communication across all workers so that each worker's per-step communication cost stays constant at $2 \times (N-1)/N \times M$ regardless of cluster size.
3. **Common pitfall**: A frequent misconception is that ring AllReduce behaves like an all-to-all exchange. In ring AllReduce, each worker communicates with neighbors according to a schedule, and the per-node communication volume is approximately constant as the

cluster grows. The scaling pressure comes from latency steps, topology, and the gradient volume per step, not from every worker opening a distinct stream to every other worker.

For standard synchronous data-parallel SGD, gradient synchronization is not a design convenience; it is the mechanism that preserves one shared optimization trajectory. If different GPUs apply different gradient updates to their local copies of the model, the copies diverge. After enough steps, the models on different GPUs represent entirely different functions, and the training process no longer approximates stochastic gradient descent on the global loss. Synchronization ensures that all copies remain identical (within floating-point precision) at every step, preserving the theoretical convergence guarantees of the optimization algorithm via the AllReduce[1] primitive.[2]

The volume of data that must be synchronized is proportional to the model size. A model with P parameters stored in BF16 (2 bytes per parameter) generates $2P$ bytes of gradient data per training step per GPU. For a 70 billion parameter model, this is 140 GB of gradients that every GPU must send and receive.[3]

At large scale (hundreds of billions of parameters across thousands of GPUs), gradient synchronization can dominate training step time unless aggressive optimization techniques are applied. The next step is to model that data movement as physics rather than as an API call.

6.1.1 The physics of data movement

Before designing algorithms, we must understand the physical constraints governing data movement. Section 3.3 establishes the wire-level physics behind these algorithms: the speed of light sets a latency floor[4] (roughly 5 μs per kilometer in fiber), the bandwidth-distance product limits how far a fast link can reach before it needs optics (PAM4 signaling and copper-vs-optics reach are developed in section 3.3.1), and kernel-bypass transports such as RDMA and GPUDirect RDMA strip the per-message software tax down to a few microseconds (section 3.4.1). Collective communication adds the energy cost of moving a bit, which the algorithms must respect as firmly as latency and bandwidth.

Moving data costs energy that scales with distance, as figure 6.1 illustrates. Concrete reference values are: local SRAM at roughly 0.5 pJ/bit, NVLink (on-package PCB) at the tens of pJ/bit, and inter-node InfiniBand at the hundreds-to-thousand-plus pJ/bit. The figure uses the higher published estimates that include link-, switch-, and transceiver-side power; chapter prose elsewhere uses the lower bound of each range. Both views agree on the central point: the energy cost climbs by two-to-three orders of magnitude between SRAM and inter-node fabric.

At the exascale (tens of thousands of GPUs), the power budget for communication rivals the power budget for computation itself. A 10,000-GPU cluster exchanging 1 GB of gradients per step at 30 pJ/bit consumes approximately 4.8 kJ per AllReduce (accounting for the factor of 2 in data movement), a nontrivial fraction of the total per-step energy budget.

These three constraints interact multiplicatively. Latency sets the floor for every message regardless of size. Bandwidth caps the throughput for large transfers. Protocol overhead adds a per-message tax that penalizes fine-grained communication, which is why the kernel-bypass transports recalled above[5] matter for collective performance. A quick AllReduce estimate makes their combined cost concrete for a realistic training scenario.

Napkin Math 6.1: AllReduce cost for a 70B model

Problem: A 70B parameter model trains with data parallelism across 64 GPUs connected by InfiniBand NDR (50 GB/s per port). Each GPU computes gradients in BF16 (2 bytes per parameter, common for Llama-class training). How long does one AllReduce take?

Step 1: Size the gradient payload. Each GPU produces a full gradient tensor: $7 \times 10^{10} \times 2$ bytes $= 140$ GB.

1 **AllReduce**: A compound term from MPI (Message Passing Interface) indicating a Reduce operation (summing data from all nodes to one) followed by a Broadcast (sending the sum to all nodes). Ring-based AllReduce optimizes this by performing both phases simultaneously in $2(N-1)$ steps, ensuring that every GPU ends with the global sum without any single node becoming a bottleneck.

2 **Parameter Server (PS)**: Google's DistBelief used a parameter-server architecture for large-scale neural-network training, and later parameter-server systems made the server-side bandwidth and consistency trade-offs explicit as worker counts grew (Dean et al. 2012; Li et al. 2014). The collective-communication lesson is that star-like aggregation concentrates traffic in the server tier, whereas peer-to-peer collectives such as Ring AllReduce distribute that traffic so each worker's bandwidth cost can remain bounded.

3 **Ring AllReduce**: The algorithm dates to the HPC collective-communication literature; Patarasuk and Yuan analyzed bandwidth-optimal AllReduce algorithms for workstation clusters, Baidu's Andrew Gibiansky popularized ring AllReduce for deep learning in February 2017, and Horovod plus NVIDIA NCCL helped make it a common distributed-training primitive (Patarasuk and Yuan 2009; Gibiansky 2017; Sergeev and Balso 2018; Jeaugey 2017).

4 **The Speed of Light Constraint**: Light travels through optical fiber at approximately 200,000 km/s, or 200 meters per microsecond. In a massive data center where cables between racks span 100 meters, the "wire delay" alone contributes 500 ns to every message—a physical limit that no amount of better networking hardware can reduce.

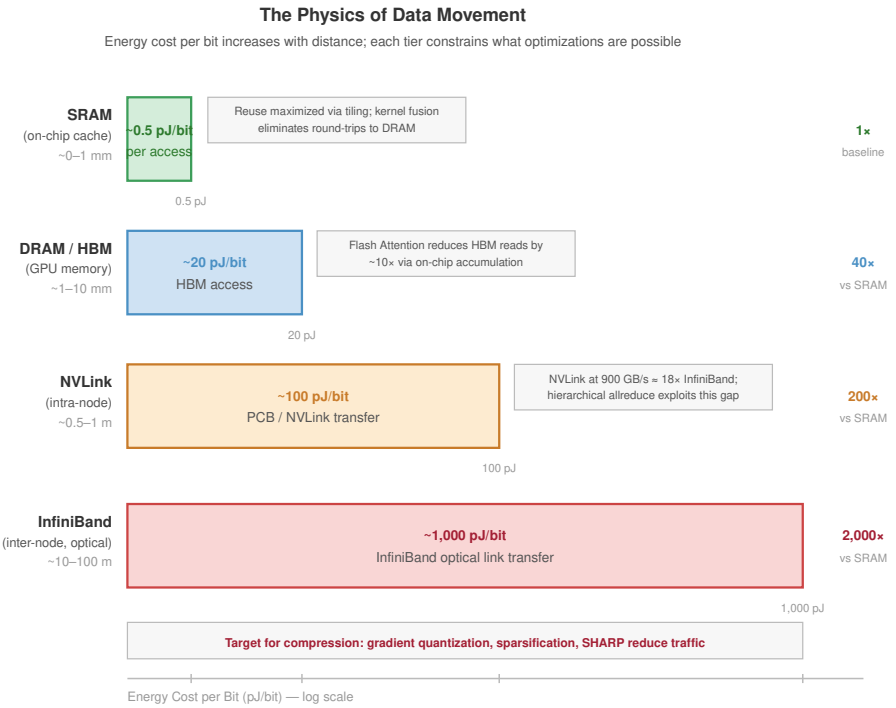


Figure 6.1: The Data Movement Energy Hierarchy: The energy cost of moving a single bit climbs by orders of magnitude across the system hierarchy — from SRAM through HBM to intra-node NVLink to inter-node InfiniBand. The figure plots higher-bound published estimates (NVLink in the tens of pJ/bit, InfiniBand near 1{,}000 pJ/bit) that include switch and transceiver overheads. The chapter prose elsewhere uses the lower-bound estimates for back-of-the-envelope calculations. Either way, the gradient makes data locality the primary driver of sustainable and efficient distributed ML system design.

5 **Zero-Copy Communication**: RDMA (Remote Direct Memory Access) allows the NIC to transfer data directly from the application's memory on one node to the application's memory on another, bypassing the operating system's kernel buffers. For a 140 GB gradient exchange, zero-copy avoids moving 280 GB of data between the CPU and main memory, reclaiming significant memory bandwidth (BW).

Step 2: Apply the Ring AllReduce bandwidth formula.

$$T_{\text{bandwidth}} = 2 \cdot \frac{N-1}{N} \cdot \frac{n}{\beta}$$

Substituting: $T_{\text{bandwidth}} = 1.96875 \times 140$ GB / 50 GB/s $\approx$ 5,512.5 ms.

Step 3: Add the latency term.

$$T_{\text{latency}} = 2(N-1) \cdot \alpha$$

Substituting: $T_{\text{latency}} = 126 \times 1.5$ µs = 0.2 ms.

Step 4: Total communication time. Total: $T_{\text{AllReduce}} \approx$ 5,512.5 ms + 0.2 ms $\approx$ 5,512.7 ms.

Systems insight: The gradient AllReduce alone takes over five seconds. This is pure communication overhead added to every training step. At this scale, communication dominates the step time unless overlapped with backward pass computation. This is why large-scale training systems pipeline AllReduce with the backward pass, launching communication for early layers while later layers are still computing.



Gradient synchronization devours 30 to 70 percent of each step at scale.

The calculation reveals why data movement, not computation, becomes the governing constraint at scale. A single AllReduce on a 70B model's gradients consumes seconds of wall-clock time, during which all GPUs would otherwise sit idle. This asymmetry between local computation (which parallelizes perfectly) and global coordination (which requires physical data movement) motivates every collective algorithm that follows. Halving that multi-second AllReduce would save thousands

of GPU-hours over a typical training campaign, translating directly to reduced cost and faster time to deployment.

The cost analysis also explains why the choice of collective algorithm matters far more than most practitioners realize. Using a suboptimal algorithm that achieves only 60 percent of theoretical bandwidth (a common outcome with poor topology mapping) would inflate the five-second AllReduce to over nine seconds, adding several seconds of pure waste to every training step. Because this waste propagates through the entire fleet, communication algorithms occupy the Distribution Layer of the fleet stack: the Infrastructure Layer below provides raw bandwidth through NVLink, InfiniBand, and network topologies (covered in Chapter 3), while the Serving Layer above depends on efficient gradient synchronization to complete training runs that produce deployable models. When communication algorithms fail to saturate the available bandwidth, training takes longer, serving models are delivered later, and the entire fleet operates below its economic potential.

Figure 6.2 quantifies how this communication overhead compounds as the fleet grows. At 8 GPUs within a single NVLink-connected node, communication consumes roughly 25 percent of each training step because the 900 GB/s interconnect bandwidth keeps pace with gradient volume. As the cluster expands to 64 GPUs across 8 nodes, the transition to InfiniBand (50 GB/s per port) shifts the balance: communication dominates at approximately 50 percent of step time, with an additional 5 percent lost to synchronization barriers. At a 4,096-GPU scale, communication and synchronization overhead together consume 80 percent of the training step, leaving only 20 percent for useful computation. This progression explains why collective algorithms matter: without hierarchical collectives, gradient compression, and communication-computation overlap, large-scale training would spend the vast majority of its multi-million-dollar compute budget waiting for data to arrive.

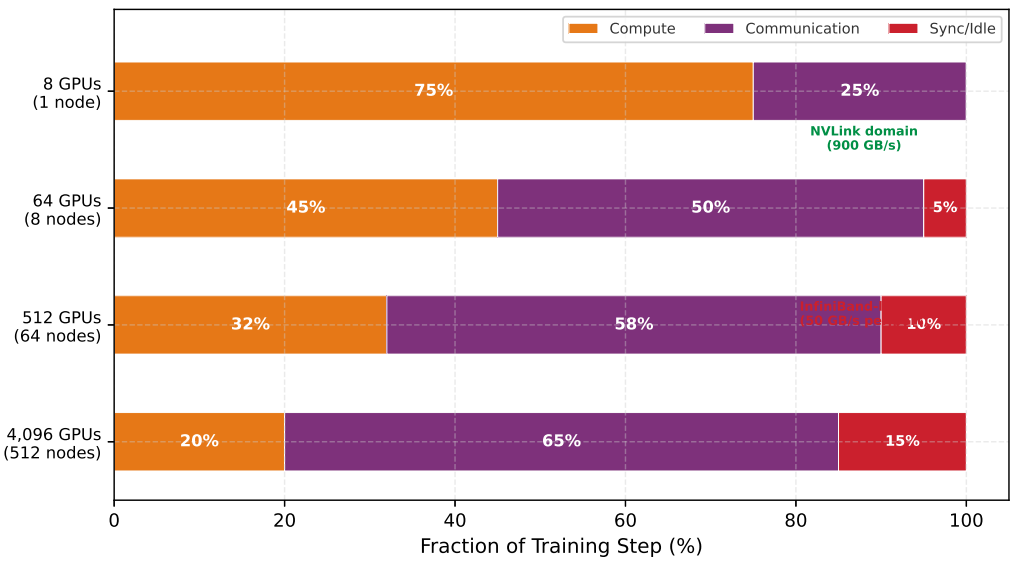


Figure 6.2: The Compute-Communication Timeline: As training scales from 8 GPUs (one node) to 4,096 GPUs, communication grows from approximately 25 percent to over 65 percent of each training step. This shift from NVLink-dominated to InfiniBand-limited communication drives the design of all collective optimization strategies. Proportions derived from H100 cluster measurements with a 70B parameter model using data parallelism with Ring AllReduce.

6.1.2 The gradient's travel manifest

The journey begins at the moment the backward pass completes. On a single machine, that was the end of the story: the weights were updated, and the neuron learned. At production scale, however, the gradient is born into isolation. It exists on one GPU, while the "truth" of the model is distributed across thousands. To achieve global convergence, the gradient must find its peers.

The specific "travel manifest" for this journey is dictated by the parallelism strategy chosen in Chapter 5. The choice of *how the math is split* determines *how the data moves*. For our Lighthouse

Archetypes (section 1.6.1), these manifests differ fundamentally. For Archetype A (GPT-4/Llama-3), our gradient is part of a massive, dense tensor that must meet every other gradient in the fleet to compute a global average, so its primary vehicle is the AllReduce. For Archetype B (DLRM at Scale), the DLRM workload[6], our gradient or activation is sparse and targeted: it does not need to meet everyone, only the specific GPU that holds its embedding shard. Mixture-of-experts (MoE) routing creates the same targeted pattern, sending each token to the GPU that hosts its assigned expert, so both rely on the AllToAll.

Understanding this mapping is essential: the *what* of parallelism directly determines the *how* of communication. At large scale, these strategies are not mutually exclusive. A single training run for a large language model typically employs 3D parallelism (combining data parallelism, tensor parallelism, and pipeline parallelism simultaneously), which means multiple collective primitives execute concurrently on overlapping subsets of GPUs. Tensor parallelism drives AllReduce operations within each node (over NVLink), pipeline parallelism drives point-to-point sends between pipeline stages (often between nodes), and data parallelism drives AllReduce operations across groups of nodes (over InfiniBand).

Each primitive operates on a different **process group**, a subset of the total GPU population that participates in that particular collective. Classical MPI uses a communicator as the object that names such a participating set; its communicator size is the number of ranks in that set. MPI libraries already selected collective algorithms based on communicator size and message size (Thakur et al. 2005). GPU communication libraries inherit that algorithm-selection problem and add the further challenge of coordinating overlapping process groups without creating contention between concurrent collectives.

Table 6.1 previews the mapping from parallelism strategy to collective primitive. The primitive names are introduced formally in the next section; for now, read the table as a traffic map that says who must exchange data with whom.

Table 6.1: The Travel Manifest: How the high-level math of Chapter 5 manifests as low-level traffic patterns.

Parallelism Strategy	The Gradient's Goal	Primary Primitive	Primary Constraint
Data Parallelism	Meet everyone, compute global average	AllReduce	Bandwidth (Large payloads)
FSDP/ZeRO	Find shards, reconstruct the whole	AllGather + ReduceScatter	Bandwidth (High frequency)
Tensor Parallelism	Quick handshake within the node	AllReduce	Latency (Speed is life)
Pipeline Parallelism	Handoff to the next neighbor	Point-to-Point (Send/Recv)	Latency (Sequential dependencies)
Expert Parallelism (MoE)	Targeted routing to a specialist	AllToAll	Latency + Contention

In this table, Expert Parallelism refers to the mixture-of-experts (MoE)[7] architecture pattern. The mapping shows that different parallelism strategies impose fundamentally different communication patterns. Data parallelism and FSDP generate large, bandwidth-bound messages that benefit from ring-based algorithms and hierarchical decomposition. Tensor and pipeline parallelism generate small, latency-bound messages that benefit from tree-based algorithms and low-overhead software stacks. Expert parallelism generates all-to-all traffic patterns that stress the network's bisection bandwidth. To reason quantitatively about these differences, we need a model of network performance.

6.2 Mapping the Terrain: Network Performance Modeling

A data center engineer cannot predict how long it takes to send ten megabytes across a cluster without knowing two distinct variables: the fixed startup tax and the per-byte transit fee. As the gradient begins its journey, it immediately encounters the physical reality of the data center network.

6.2.1 The alpha-beta cost model: Startup tax and transit fee

Every message our gradient sends obeys the linear cost model $T(n) = \alpha + n/\beta$—a fixed startup tax plus a per-byte transit fee. Those two terms decide every algorithm choice in this chapter.

6 **DLRM**: Meta's 2019 architecture for click-through rate prediction, where embedding tables can exceed 100 GB and must be sharded across workers. Each forward pass triggers AllToAll to exchange sparse embedding lookups, creating a communication pattern where message sizes are small (hundreds of KB) but fan-out is $\mathcal{O}(N)$, making DLRM one of the most latency-sensitive distributed workloads in production.

7 **Mixture-of-Experts (MoE)**: An architecture where each token activates only a subset of specialized subnetworks (experts), reducing per-token FLOPs while maintaining total model capacity. The systems trade-off is stark: MoE replaces the bandwidth-bound AllReduce of dense models with latency-bound AllToAll, shifting the communication bottleneck from β to α and creating $\mathcal{O}(N^2)$ contention that limits practical cluster size.

Definition 6.2: α-β model (Hockney model)

α-β Model (Hockney Model) is the linear communication cost model $T(n) = \alpha + n/\beta$ that decomposes message transfer time into a fixed startup latency (α, the per-message overhead) and a message-size-dependent bandwidth term (n/β, proportional to bytes transferred), enabling algorithm designers to predict when message fusion or gradient compression will improve throughput (Hockney 1994).

1. **Significance**: For InfiniBand NDR at $\alpha \approx 2\,\mu$s and $\beta \approx 50\,$GB/s, the crossover size $n^* = \alpha \cdot \beta \approx 100\,$KB. A 4 KB routing message is far below n^*, so the startup tax dominates; fusing 100 such messages into one 400 KB message reduces communication cost from $100\alpha = 200\,\mu$s to one $\alpha + n/\beta \approx 10\,\mu$s, a 20× improvement. A 140 GB gradient tensor is far above n^*, so bandwidth dominates and reducing payload bytes is the effective lever.
2. **Distinction**: Unlike idealized throughput models that treat bandwidth as the sole communication cost, the α-β model reveals that N small messages of size n/N cost up to $N\times$ more than one large message of size n when $n/N \ll n^*$, explaining why NCCL fuses small AllReduce calls and why MoE routing algorithms buffer tokens before launching collectives.
3. **Common pitfall**: A frequent misconception is that gradient compression always helps. If the compressed gradient size remains well above n^*, compression reduces the bandwidth term but leaves the latency term unchanged. Even shrinking a 70B model's gradient payload from 140 GB to 1.4 GB leaves the message firmly bandwidth-bound; it does not turn the operation into a low-latency exchange.

The two parameters have distinct physical meanings. Latency (α) is the fixed start-up cost to send a message regardless of size, covering software overhead (kernel launch, NCCL initialization), PCIe traversal, and network switching time. Bandwidth (β) is the sustained data transfer rate in bytes per second. The **critical message size** $n^* = \alpha \cdot \beta$ marks the crossover point: messages smaller than n^* are latency-bound; messages larger are bandwidth-bound. Section D.1 works this model through concrete message-size regimes and roofline analysis, so a reader who wants the full derivation behind the crossover can follow it there; here we establish the parameters and apply them directly.

Table 6.2 shows typical values for data center interconnects, and the critical-size column carries the load-bearing pattern: intra-node NVLink stays latency-bound up to several hundred kilobytes, whereas the inter-node fabrics cross over near 100 KB, so a message large enough to be bandwidth-bound inside a node can still be latency-bound once it travels between nodes.

Table 6.2: Interconnect Performance Parameters: Effective per-message launch latency (α, including the software stack), per-direction link bandwidth (β, the one-way deliverable rate that enters $T(n) = \alpha + n/\beta$; NVLink's datasheet 900 GB/s is the bidirectional total, so its β is the 450 GB/s one-way rate), and the computed crossover size $n^* = \alpha \cdot \beta$ at each row's α midpoint, where communication transitions from latency-bound to bandwidth-bound.

Interconnect	Latency (α)	Bandwidth (β)	Critical Size (n^*)
NVLink 4.0 (intra-node)	1–2 µs	450 GB/s	~0.7 MB
InfiniBand NDR 400 Gbps	1–3 µs	50 GB/s (per port)	~100 KB
InfiniBand HDR 200 Gbps	2–5 µs	25 GB/s	~87.5 KB
PCIe Gen5 (GPU CPU)	2–5 µs	64 GB/s	~224 KB
Ethernet 100 Gbps (RoCE)	5–10 µs	12.5 GB/s	~93.7 KB

Applying the critical message size formula to a concrete workload reveals which optimization strategy matters most:

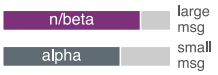
Napkin Math 6.2: The critical message size

Problem: Consider a cluster using InfiniBand NDR 400 Gbps with $\alpha = 2$ µs and $\beta = 50$ GB/s. At what message size does optimizing for bandwidth start to matter more than optimizing for latency?

Math:

$n^* = \alpha \cdot \beta = 2 \times 10^{-6}$ s $\times\ 5 \times 10^{10}$ B/s $= 100$ KB

Systems insight: Messages under 100 KB (like MoE tokens, pipeline activations) are **latency-bound**: buy lower-latency switches and reduce software overhead. Messages over 100 KB, such as large language model (LLM) gradients, are **bandwidth-bound**: buy more bandwidth and compress the data. Applying the wrong optimization wastes money without improving performance.



Small messages are latency-bound, large ones bandwidth-bound; the cost reverses.

The critical message size separates two distinct operating regimes. Below it, small messages such as MoE routing tokens or scalar reductions are latency-bound ($n < n^*$), where time is dominated by α and optimization focuses on fusion (batching small messages), topology (reducing hop count), and software-stack tuning (kernel bypass via RDMA). Above it, large messages such as LLM gradients or optimizer states are bandwidth-bound ($n \gg n^*$), where time is dominated by n/β and optimization shifts to compression (lower precision or sparsity), algorithm choice (Ring vs. Tree), and link aggregation (multi-rail NICs). The distinction between latency-bound and bandwidth-bound communication is the diagnostic skill to practice.

Checkpoint 6.1: Alpha-beta diagnostics

Verify your understanding of network performance regimes:

- ☐ **Bound regime**: Classify a 10 KB message sent over a link with $\alpha = 2$ μs and $\beta = 10$ GB/s as latency-bound or bandwidth-bound.
- ☐ **Software overhead**: Determine which type of parallelism benefits more when "Software Overhead" (α) is reduced by half: Data Parallelism or Tensor Parallelism.
- ☐ **Critical message size**: Describe how the critical message size ($n^* = \alpha\beta$) changes when networking upgrades from 100 Gbps to 400 Gbps while software latency stays constant.
- ☐ **NIC scaling**: Assess the True or False claim that, in the bandwidth-bound regime, doubling the number of NICs per node effectively doubles the transmission speed for large tensors.

6.2.2 The LogP model

The α-β model assumes the processor is idle during communication. For pipelined systems where we overlap communication with computation, this assumption fails. The **LogP model** (Culler et al. 1993) extends α-β with four parameters:

- L_{lat} **(Latency)**: The time for a message to traverse the network (similar to α).
- o **(Overhead)**: The CPU/GPU time spent initiating or receiving a transfer. During this time, the processor cannot compute, making this the nonoverlappable cost. In a distributed training context, o is the aggregate of PyTorch or JAX dispatch overhead, CUDA kernel launch time, tensor memory registration with the RDMA stack, and occasionally Python Global Interpreter Lock (GIL) contention when the communication thread competes with the training loop. These software layers account for why measured NCCL overhead is 25–50 μs per collective even when the wire-level α is only 1–3 μs; the NCCL comparison later in this section makes the gap concrete.
- g **(Gap)**: The minimum time interval between consecutive message injections (inverse of message rate). This models link contention.
- N_{rank} **(Rank count)**: The number of ranks in the communication group.

LogP distinguishes network latency (L_{lat}, which can be hidden) from processor overhead (o, which *cannot*). A system can overlap communication with computation only if the compute kernel runs longer than the overhead. A small overlap calculation makes this distinction concrete:

Napkin Math 6.3: Hiding communication behind computation

Problem: A training pipeline attempts to overlap gradient AllReduce with the next layer's backward pass. The backward pass takes 500 µs. The AllReduce has network latency 100 µs but processor overhead $o = 50$ µs to initiate and $o = 50$ µs to receive. Can the communication be hidden?

Math:

1. **Overlappable portion**: Network latency $L_{\text{lat}} = 100$ µs (data in flight while GPU computes).
2. **Non-overlappable portion**: $2o = 100$ µs (GPU busy initiating/receiving).
3. **Compute available**: 500 µs.
4. **Hidden**: All 100 µs of L_{lat} can overlap with compute.
5. **Exposed**: The 100 µs of $2o$ overhead cannot overlap.

Result: Effective time is 100 µs + max(500 µs, 100 µs) = 600 µs. The network latency is hidden, but the processor overhead remains exposed. Figure 6.3 shows this overlap visually.

Systems insight: The α-β model captures the total communication time. The LogP model reveals *how much of it can be hidden*. When designing pipelined training, optimize for low o (kernel bypass, GPUDirect) rather than high β alone.

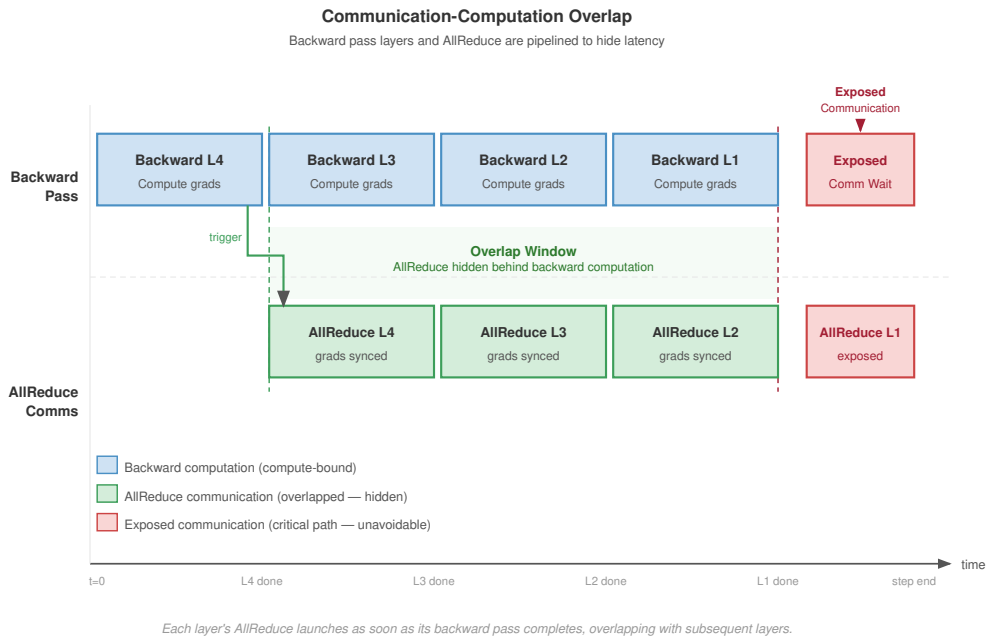


Figure 6.3: Communication-Computation Overlap: By pipelining collective operations with the backward pass, systems can hide network latency behind arithmetic execution. Overlap is successful only when the nonoverlappable overhead (o) is minimized, allowing the bulk transfer term to proceed while the GPU computes the next layer's gradients.

The choice between models depends on the analysis context. The α-β model is the right tool for back-of-envelope calculations, algorithm selection (Ring vs. Tree), and cases where communication is *blocking* (synchronous barriers). Its strength is simplicity, since the two parameters can be measured

directly with a point-to-point bandwidth test and a zero-byte message latency test. Its weakness is the assumption that the processor is idle during communication, which makes it overly pessimistic when overlap is possible. The LogP model earns its extra parameters when the analysis turns to **pipelined execution**, compute-communication overlap, or debugging why a theoretically fast algorithm underperforms (often high o). Its distinction between network latency L_{lat} (which can be hidden) and processor overhead o (which cannot) determines whether a given overlap strategy will actually hide communication. The cost is that measuring o accurately requires profiling tools such as NVIDIA Nsight Systems, because o depends on the specific communication library and GPU driver stack.

In practice, most engineering calculations start with the α-β model for initial sizing and algorithm selection, then refine with LogP analysis when communication-computation overlap is the target optimization. Both models share a common limitation: they assume a single flow on a single link. Real communication patterns involve multiple simultaneous flows competing for shared bandwidth, which can cause congestion that neither model captures. For congestion-sensitive workloads (particularly AllToAll for MoE), empirical benchmarking on the target cluster remains the necessary validation step.

6.2.3 Putting the model to work: Llama 70B communication budget

The alpha-beta model becomes most valuable when applied to real training configurations. Consider a concrete scenario: training a Llama-class 70B parameter model using data parallelism across 128 GPUs spanning 16 nodes of 8 GPUs each. The gradient tensor is 140 GB in BF16 (70 billion parameters at 2 bytes each, common for Llama-class training). During each training step, this entire gradient must be synchronized across all workers.

The calculation uses the bandwidth hierarchy before the chapter formalizes it: reduce as much data as possible over the fast NVLink tier inside each node, send only the reduced shard over the slower InfiniBand tier, then reconstruct the result locally. The primitive names become precise in the hierarchy section; the engineering idea is already visible in the bandwidth budget.

Using BF16 gradients (a common practice that halves communication volume to 140 GB), the contrast is stark. Routing the full gradient over the slow inter-node fabric, as a flat Ring AllReduce does, costs roughly 5,557 ms. Confining most of the traffic to NVLink and sending only the reduced shard over InfiniBand cuts that to approximately 1200.8 ms. The difference is not marginal; it determines whether communication can be hidden behind computation or whether it becomes the critical path.

The per-phase breakdown behind these numbers (the intra-node reduction, the reduced inter-node exchange, and the intra-node redistribution) requires the collective primitives this section has not yet introduced. Section 6.5.1 names those primitives and works the full three-phase derivation; the budget here establishes only that the bandwidth hierarchy is worth respecting, and by how much.

6.2.4 Theory vs. practice: The NCCL reality gap

The bandwidth-latency trade-off (principle 11) provides useful first-order predictions, but real communication libraries introduce overheads that the idealized α-β model does not capture. NCCL, a widely used GPU communication library, adds protocol negotiation, memory registration, and internal pipelining that modify the effective α and β values (Jeaugey 2017; NVIDIA 2026b). Table 6.3 compares one-message α-β payload predictions against measured NCCL performance for common message sizes on an 8-node DGX H100 cluster (64 GPUs, InfiniBand NDR 400G).

Table 6.3: α-β Predictions vs. Measured NCCL Performance: For small messages, NCCL's protocol overhead (memory registration, channel setup, kernel launch) inflates the effective latency by 7–8× over the bare-wire α. For large messages, NCCL achieves within 8–15 percent of theoretical bandwidth, validating the one-message payload model in the bandwidth-bound regime. Values are illustrative reference numbers for an effective NCCL payload-transfer model on 8-node DGX H100 clusters with InfiniBand NDR 400G; actual collective timings vary with algorithm, topology, NCCL version, and tuning.

Message Size	α-β Prediction	Measured NCCL	Ratio (Measured/Predicted)	Explanation
1 KB	~3.1 µs	~25 µs	~8.1×	NCCL protocol setup dominates

Message Size	α-β Prediction	Measured NCCL	Ratio (Measured/Predicted)	Explanation
64 KB	~4.4 µs	~30 µs	~6.9×	Still latency-bound; NCCL overhead
1 MB	~23.1 µs	~40 µs	~1.7×	Transitioning to bandwidth-bound
64 MB	~1.3 ms	~1.6 ms	~1.2×	NCCL approaches theoretical bandwidth
1 GB	~20 ms	~23 ms	~1.15×	Bandwidth-dominant; NCCL nearly optimal
10 GB	~200 ms	~215 ms	~1.07×	Large payloads saturate the wire

The table reveals two critical lessons. First, the α-β model underestimates small-message latency by 7–8× because it accounts only for wire-level propagation, not the software stack overhead. For latency-sensitive operations (tensor parallelism AllReduce, MoE token routing), the effective α is 5–10× higher than the physical wire latency. Second, for large messages the model is accurate to within 8–15 percent, confirming that bandwidth is the binding constraint and that NCCL's internal optimizations (channel pipelining, kernel fusion) successfully saturate the available links.

This reality gap has practical consequences for algorithm selection. The crossover point between Ring and Tree AllReduce shifts upward in practice because the effective α is larger than the wire-level value. Engineers who use textbook α values will underestimate latency costs and may choose Ring when Tree would perform better. A robust practice is to measure the effective α on the specific cluster by benchmarking small-message AllReduce latency, then use that measured value in all subsequent calculations.

6.3 Choosing the Vehicle: Collective Operation Primitives

If a GPU simply opens a socket and sends a massive gradient to another GPU, the entire cluster will rapidly collapse into an unmanageable web of deadlocks and congestion. With the terrain mapped, the gradient must now choose its vehicle: strictly choreographed group exchanges known as collective operations, a vocabulary standardized by MPI and inherited by modern ML communication libraries (Message Passing Interface Forum 2015). Figure 6.4 illustrates four of the central primitives in this vocabulary: AllReduce, AllGather, ReduceScatter, and AllToAll. Each panel reads as a before-and-after across the process group, with one row per rank: blue cells mark the input each rank starts with, green cells mark the result it ends with, and orange marks the AllToAll routing that shuffles unique data between every pair of ranks. Two further primitives, Broadcast (rank 0 sends to all) and Reduce (all aggregate to rank 0), are foundational building blocks defined in the prose below and used implicitly in the figure's compound patterns. Section D.2 formalizes the semantics and latency and bandwidth complexity of each primitive; the prose here introduces them through their workload use cases, and the reader who wants the formal complexity bounds before proceeding can establish them there first.



Definition 6.3: Collective operation

Collective Operation is a distributed communication pattern in which all processes in a group participate simultaneously to aggregate, broadcast, or redistribute data, with the correctness guarantee that each participant receives the result prescribed by the collective's semantics regardless of message ordering or arrival time.

1. **Significance**: The right collective algorithm determines whether communication scales with cluster size or remains constant. Ring AllReduce achieves bandwidth-optimal $2(N-1)/N \times M/\beta$ per node (constant in N), while a naive reduce-then-broadcast approach costs $\mathcal{O}(N \times M/\beta)$, making it 500× worse for $N = 1024$. Algorithm selection thus directly determines whether the BW term in the iron law is the training bottleneck.

2. **Distinction**: Unlike point-to-point communication (where one sender and one receiver exchange data independently), a collective operation coordinates the entire process group—every participant must invoke the collective before any can complete it, and the library guarantees a consistent result even when different processes contribute different data.
3. **Common pitfall**: A frequent misconception is that one collective algorithm suits all message sizes. Ring AllReduce achieves near-optimal bandwidth for large gradient tensors but has $\mathcal{O}(N)$ latency—for small messages like MoE routing decisions (~4 KB), a tree-based algorithm reduces latency to $\mathcal{O}(\log N)$ steps at the cost of suboptimal bandwidth utilization (the bandwidth-optimal recursive halving-doubling alternative is covered in section 6.4.5), requiring workload-specific algorithm selection rather than a single default.

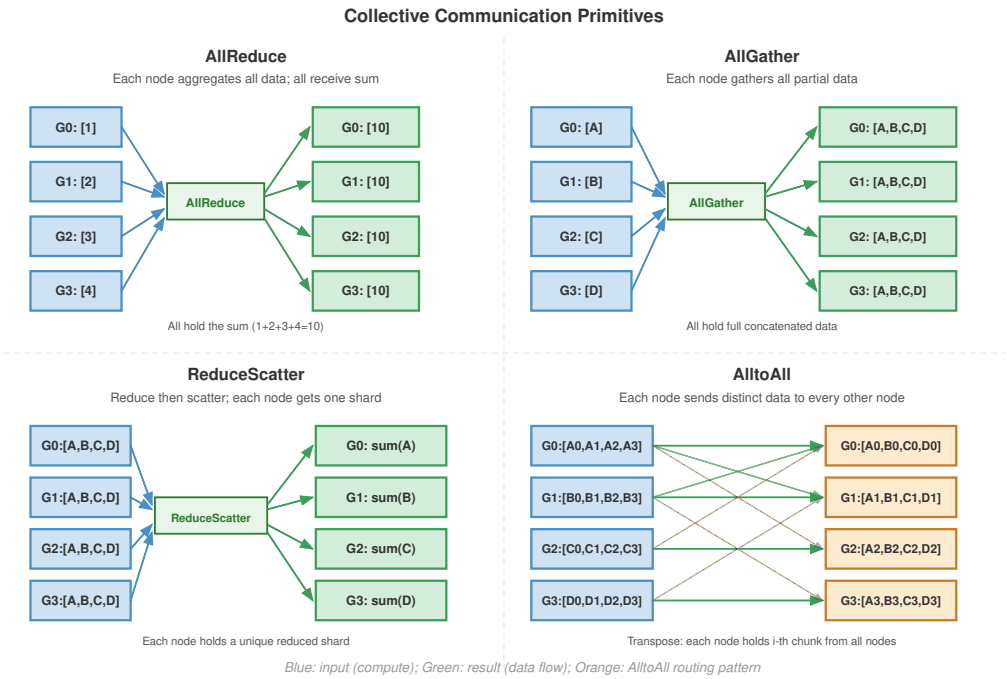


Figure 6.4: Four Central Collective Primitives: Standardized patterns for group communication in distributed systems. (1) AllReduce: all aggregate and all receive the result. (2) AllGather: all send to all and concatenate. (3) ReduceScatter: all aggregate and results are distributed in shards. (4) AllToAll: every process sends unique data to every other process. The two foundational primitives Broadcast (rank 0 sends to all) and Reduce (all aggregate to rank 0) compose the patterns shown here and are defined in the surrounding prose.

The primitive choice is the first scaling decision because it fixes which ranks must coordinate, which data must move, and which communication term will dominate. Different model architectures stress different operations, and selecting the wrong primitive for a workload creates unnecessary bottlenecks.

6.3.1 The six core primitives

The six primitives in table 6.4 form a decision ladder: use the narrowest operation that preserves the model's mathematics, because every broader pattern adds participants, barriers, or contention.

Table 6.4: The Six Collective Primitives: What each primitive computes, its primary use, and its communication cost, ordered from the narrowest pattern (Broadcast) to the most general (AllToAll).

Primitive	What it does	Primary use case	Communication cost
Broadcast	One sender transmits data to all receivers.	Distribute initial model weights from rank 0 at startup, across all parallelism strategies.	$\mathcal{O}(\log N)$ latency (tree); $\mathcal{O}(M)$ bandwidth.
Reduce	Aggregate data from all workers (sum, min, max) to a single root.	Aggregate validation metrics such as loss and accuracy to a logging process.	$\mathcal{O}(\log N)$ latency (tree); $\mathcal{O}(M)$ bandwidth.
AllReduce	Aggregate from all workers, then distribute the result to all ($y_i = \sum_{j=0}^{N-1} x_j$ for all i).	Data parallelism (section 5.3): synchronize gradients so every GPU computes the same update.	Ring is bandwidth-optimal at $2\frac{N-1}{N}\frac{M}{\beta}$ but $\mathcal{O}(N)$ latency; Tree gives $\mathcal{O}(\log N)$ latency with worse bandwidth.
AllGather	Each worker's data goes to all; the result concatenates every input ($x_i \to [x_0, \ldots, x_{N-1}]$).	Sharded data parallelism (FSDP/ZeRO): collect sharded parameters before a forward or backward pass.	Ring $\frac{N-1}{N}\frac{M}{\beta}$; resident data grows from an M/N shard to the full M.
ReduceScatter	Reduce across workers, but scatter so each keeps a distinct chunk (worker i receives the i-th block of $\sum x_j$).	Sharded data parallelism: reduce gradients while keeping them sharded to save memory.	Ring $\frac{N-1}{N}\frac{M}{\beta}$; the inverse of AllGather.
AllToAll	The most general pattern: worker i sends a distinct chunk to each worker j, a distributed matrix transpose.	MoE routing tokens to experts; DLRM exchanging embedding lookups across workers.	$\frac{N-1}{N}M$ per worker, but $\mathcal{O}(N^2)$ logical connections make contention the scaling limit.

Ring achieves AllReduce's bandwidth-optimal cost while Tree trades bandwidth for logarithmic latency; section D.2.1 develops the formal definition and a worked example comparing the two, so a reader who wants that derivation can take the full treatment there (Patarasuk and Yuan 2009; Thakur et al. 2005).

A key insight is that AllReduce can be decomposed into ReduceScatter followed by AllGather. This is not merely a mathematical equivalence; it is precisely how Ring AllReduce works internally (the Scatter-Reduce phase is a ReduceScatter, the AllGather phase is an AllGather). ZeRO-style sharding, realized in PyTorch as FSDP[8], exploits this decomposition by AllGathering parameter shards before a module needs them and ReduceScattering gradients after backward so each rank retains only its shard for the optimizer step (Rajbhandari et al. 2020). This shards parameters, gradients, and optimizer state across workers, reducing persistent model-state memory roughly by $N\times$, while temporary full-parameter buffers and prefetching determine peak memory.

The same AllGather/ReduceScatter pair also appears in sequence parallelism: an AllGather reconstructs the activation view needed for a local operation, and a later ReduceScatter redistributes the result along the sequence dimension. That example will matter later because it shows that a primitive's semantics stay fixed even when the tensor dimension being sharded changes.

This ladder matters because the primitive determines the synchronization shape before any algorithmic optimization begins. AllReduce creates global agreement, AllGather and ReduceScatter trade memory for repeated shard movement, and AllToAll replaces a bandwidth problem with fan-out and contention.

The contrast between AllToAll and AllReduce highlights a fundamental difference in how collective operations scale with cluster size.

8 **FSDP (Fully Sharded Data Parallel)**: PyTorch's fully sharded implementation follows the ZeRO-3 idea of sharding parameters, gradients, and optimizer states across N accelerators (Rajbhandari et al. 2020). The trade-off is communication frequency: full sharding adds layer-level AllGather and ReduceScatter operations instead of relying only on data parallelism's step-level AllReduce, making it more sensitive to the α overhead.

> **Systems Perspective 6.1: AllToAll vs. AllReduce: Why scale differs**
>
> While AllReduce scales efficiently because it can be pipelined in a ring (where each node only talks to its neighbor), AllToAll is fundamentally harder to scale.
>
> In an AllToAll, every process has a unique piece of data for every other process. This creates $\mathcal{O}(N^2)$ logical connections. At the hardware level, this leads to **network contention**: if 1024

GPUs all try to send data to different targets simultaneously, the "Fat-Tree" or "Spine" switches in the data center become the bottleneck.

This is why Expert Parallelism (MoE) and large-scale recommendation systems often hit a "communication wall" much earlier than standard data-parallel models. The algorithm choice (AllReduce vs. AllToAll) determines the scaling ceiling.

To make the AllToAll pattern concrete, consider a Mixture of Experts model with 64 experts distributed across 8 GPUs (8 experts per GPU). During each forward pass, a gating network assigns each input token to one or more experts. The tokens assigned to experts on remote GPUs must be physically moved to those GPUs before computation can proceed, and the results must be moved back afterward.

Napkin Math 6.4: AllToAll for MoE token routing

Problem: A MoE model processes a batch of 4096 tokens across 8 GPUs (512 tokens per GPU). Each token is a 2048-dimensional hidden state in BF16 (4.096 KB per token). The gating network assigns each token to exactly 1 of 64 experts (8 experts per GPU). Assuming uniform routing (each expert receives 64 tokens), how much data does each GPU send and receive?

Math:

Each GPU holds 512 tokens that need to reach 64 different experts across 8 GPUs. With uniform routing, each GPU sends 64 tokens to each of the other 7 GPUs (keeping 64 tokens local).

- Data per GPU-to-GPU transfer: 64 tokens $\times$ 4.096 KB/token = 262.144 KB
- Total data sent per GPU: 7 $\times$ 262.144 KB = 1.84 MB
- Total data received per GPU: 1.84 MB (symmetric)

Latency Analysis (InfiniBand NDR, $\alpha = 3$ µs, $\beta = 50$ GB/s):

Each of the 7 transfers is a 262.144 KB message. From the α-β model: T_{transfer} = 3 µs + 5 µs = 8 µs per transfer.

If serialized: 7 $\times$ 8 µs = 56 µs. If all 7 transfers run in parallel (full-duplex, non-blocking): $\approx$ 8 µs.

Systems insight: The per-transfer sizes in this MoE example sit near the critical-message-size boundary, so startup overhead remains comparable to serialization time. This is why MoE scaling is often constrained by α (latency), fan-out, and contention rather than raw β (bandwidth) alone, and why MoE systems benefit from low-latency switches, RDMA, message fusion, and topology-aware routing. The AllToAll also runs twice per layer (once for token dispatch, once for result collection), doubling the startup tax.

In practice, MoE routing is rarely perfectly uniform. Popular experts receive more tokens than unpopular ones, creating load imbalance that translates to communication imbalance. If one expert receives 3$\times$ more tokens than average, the GPU hosting that expert receives 3$\times$ more incoming data, creating a hotspot that can stall the entire collective (since AllToAll is a barrier operation). Large MoE systems address this through auxiliary load-balancing losses that penalize the gating network for routing too many tokens to any single expert, and through capacity factors that cap the maximum number of tokens an expert can accept (dropping overflow tokens). These techniques trade a small amount of model quality for communication balance, which is the right trade-off at scale.

Table 6.5 maps these primitives to the parallelism strategies introduced in Chapter 5 and the lighthouse archetypes introduced in section 1.6.1.

Table 6.5: Collective Operation Selection Guide: Matching training strategies to their primary collective operations enables efficient distributed communication design. MoE and DLRM serve as canonical "lighthouse" workloads throughout this book, representing sparse expert architectures and recommendation systems respectively. Pipeline Parallelism uniquely relies on point-to-point rather than collective communication.

Training Strategy	Primary Collective	Bottleneck Characteristic
Data Parallelism	AllReduce	Bandwidth-bound (large gradients)
FSDP/ZeRO-3	AllGather, ReduceScatter	Bandwidth-bound, high frequency
Tensor Parallelism	AllReduce, AllGather	Latency-bound (requires NVLink)
Pipeline Parallelism	Point-to-Point (Send/Recv)	Latency-bound (microbatch handoffs)
Sequence Parallelism	AllGather, ReduceScatter	Bandwidth-bound (activation exchange)
MoE (Experts)	AllToAll	Latency-sensitive + contention
DLRM (RecSys)	AllToAll	Latency & Bandwidth (sparse lookups)

6.3.2 FSDP communication patterns



FSDP trades one step-level collective for two per-layer collectives.

The FSDP/ZeRO strategy deserves special attention because its communication pattern differs fundamentally from standard data parallelism (Rajbhandari et al. 2020). In standard data parallelism, each GPU holds a complete copy of the model, computes gradients locally, and synchronizes via a single AllReduce at the end of the backward pass. Full sharding eliminates this redundancy by partitioning the model parameters across GPUs, so each GPU holds only its local shard outside the moments when a layer must be reconstructed.

Because each GPU now holds only $1/N$ of the parameters, it must reconstruct the full tensor before it can use a layer, and that reconstruction is what transforms the communication pattern. Before each layer's forward pass, the GPU calls AllGather to collect the shards from all other GPUs. After the backward pass, the gradients are reduced and redistributed via ReduceScatter, so each GPU holds gradients only for its own parameter shard. The sharded parameters can then be discarded (freed from memory) until the next forward pass needs them.

The consequence is a displacement of overhead: sharding relieves the memory bottleneck but raises communication frequency. Standard data parallelism communicates once per training step (a single AllReduce of the full gradient). FSDP communicates twice per layer per step (AllGather in forward, ReduceScatter in backward), but each communication is smaller (only that layer's parameters, sharded across ranks). For a model with N_L layers, FSDP issues $2N_L$ collective operations per step instead of 1; across all those operations, the total communication volume can be comparable to the single full-gradient AllReduce, but it is spread across many smaller operations.

The higher operation count makes FSDP more sensitive to latency (α) than standard data parallelism. If each of the $2N_L$ collectives pays the full NCCL startup overhead (25–50 μs per operation from table 6.3), the aggregate overhead for a 100-layer model is 5–10 ms, which can represent a meaningful fraction of step time. FSDP implementations mitigate this through prefetching (launching the next layer's AllGather while the current layer is computing) and communication stream pipelining (using dedicated CUDA streams for communication that overlap with compute streams).

Selecting the right collective algorithm at each invocation requires understanding these asymmetric patterns. The AllGather operations in FSDP are medium-sized (one layer's parameters, typically 10–100 MB for transformer models) and latency-sensitive (because computation blocks until the full parameters are available). The ReduceScatter operations are of similar size but can overlap with the next layer's backward computation. This asymmetry means the AllGather operations benefit more from low-latency algorithms (Tree or hybrid) while the ReduceScatter operations can use bandwidth-optimal algorithms (Ring) because their latency is hidden behind computation.

The collective primitives catalog above defines *what* must be communicated; the question now becomes *how* to execute these primitives efficiently on real networks. The most performance-critical primitive, AllReduce, admits multiple algorithmic implementations whose latency and bandwidth characteristics differ by orders of magnitude. The choice of algorithm can change AllReduce time by an order of magnitude, making this the single most consequential implementation decision in distributed training.

6.4 Engineering the Flow: AllReduce Algorithms

A correct AllReduce must give all 1,000 GPUs the exact sum of all 1,000 one-gigabyte gradients, and the algorithm that delivers it sets the cluster's scaling ceiling. The naive answer—every GPU sends to one central server, whose network card instantly saturates—fails first, which makes it the right place to start.

6.4.1 Naive approaches vs. the bandwidth bottleneck

Consider a naive implementation using a Parameter Server (Star topology). All N workers send their gradients to rank 0; rank 0 sums them and sends the result back. The constraint is rank 0's bandwidth: it must receive $N \times M$ bytes and send $N \times M$ bytes, so the time grows as $T \propto N \times M/\beta$. At 1,000 GPUs, this makes the central rank the scaling bottleneck rather than the arithmetic.

This naive approach captures the pressure that early distributed ML frameworks had to manage: parameter-server systems receive, aggregate, and redistribute model updates through a server tier, while practical implementations shard the parameter space to spread that load (Dean et al. 2012; Li et al. 2014). Sharding reduces the per-server traffic from $N \times M$ to $N \times M/K_{\text{srv}}$ (where K_{srv} is the number of servers), but the server tier still grows as workers and model state grow, and it introduces additional complexity in parameter partitioning and consistency management.

The fundamental limitation is that any star-topology approach concentrates traffic at a central point. Regardless of how many servers participate, the aggregate traffic through the central tier scales as $\mathcal{O}(N \times M)$. To achieve true scalability, we need algorithms where the communication volume per node is constant regardless of N. This property, known as **bandwidth optimality**, is the design target for scalable collective algorithms.

War Story 6.1: When the parameter server became the bottleneck

Context: In 2017, Alexander Sergeev and Mike Del Balso's team at Uber open-sourced Horovod as part of Uber's Michelangelo ML platform, where TensorFlow training jobs needed to scale across many GPUs without forcing every model author to rewrite large portions of training code (Sergeev and Balso 2018).

Failure mode: TensorFlow's stock distributed training relied on a parameter-server pattern that concentrated gradient traffic through a central tier and imposed non-negligible communication overhead. Scaling beyond a single node required heavy code modification, and the star-topology aggregation degraded as worker counts grew.

Consequence: Horovod adopted Baidu's draft ring-allreduce implementation and wrapped efficient ring reduction behind a small API surface—just a few lines added to existing TensorFlow code (Gibiansky 2017; Sergeev and Balso 2018). The bandwidth bottleneck moved off the central server and onto bandwidth-optimal collective communication, and distributed training became practical for teams without distributed-systems expertise.

Systems lesson: A collective algorithm is also a developer interface. Scaling improves when the communication primitive is both bandwidth efficient and easy enough that teams actually use it correctly.

6.4.2 The bandwidth-optimal lower bound

Before examining specific algorithms, it is useful to establish a theoretical lower bound. In any correct AllReduce, every GPU starts with M bytes of local data and ends with M bytes of globally reduced data. Each byte of the final result incorporates information from all N GPUs, which means every GPU must receive at least $M \cdot (N-1)/N$ bytes of "new" information (the contributions from all other GPUs). Symmetrically, each GPU must send at least $M \cdot (N-1)/N$ bytes (its own contribution to the other GPUs' results).

The minimum total transfer per GPU is therefore $2 \cdot M \cdot (N-1)/N$ bytes (send plus receive). Dividing by the link bandwidth β gives the **bandwidth lower bound**:

$$T_{\text{bandwidth}}^{\text{min}} = \frac{2(N-1)}{N} \cdot \frac{M}{\beta}$$

As N grows large, this approaches $2M/\beta$, which is independent of N. An algorithm that achieves this bound is bandwidth-optimal. Ring AllReduce achieves this bound exactly; simple tree variants do not (Patarasuk and Yuan 2009; Thakur et al. 2005). This distinction has practical consequences: on a 64-GPU cluster synchronizing a 1 GB gradient, the bandwidth difference between an optimal and a merely logarithmic algorithm amounts to several milliseconds per step, which accumulates to hours over a multi-day training run.

6.4.3 Ring AllReduce

Ring AllReduce[9] arranges nodes in a logical ring ($0 \to 1 \to ... \to N-1 \to 0$). It achieves bandwidth optimality by pipelining: every node sends and receives simultaneously on every link (Patarasuk and Yuan 2009).

The algorithm splits the vector of size M into N chunks and then alternates communication and local reduction in two named phases: Scatter-Reduce followed by AllGather. Algorithm 6.1 states the mechanics before the figure and trace make the same dataflow concrete.

Algorithm 6.1 Ring AllReduce

Require: N ranks in a logical ring; each rank i starts with tensor G_i of M bytes, split into N chunks
Ensure: every rank holds $G = \sum_{i=0}^{N-1} G_i$
1: assign the ring order $0 \to 1 \to ... \to N-1 \to 0$
2: **for** $N-1$ scatter-reduce steps **do**
3: each rank sends one chunk (or partial sum) right, receives one from the left, and adds it into the matching local sum
4: **end for**
5: each rank now owns one fully reduced chunk of G
6: **for** $N-1$ all-gather steps **do**
7: each rank forwards a reduced chunk around the ring and stores the chunks it receives
8: **end for**
9: **return** the complete reduced tensor G, held on every rank

Across the $2(N-1)$ send/receive rounds each rank moves M/N bytes per round, for $2(N-1)M/N$ bytes sent and the same received: the bandwidth term reaches the information-theoretic lower bound, while the latency term grows as $2(N-1)\alpha$. The data flow during the Scatter-Reduce phase is illustrated in figure 6.5.

The key property visible in figure 6.5 is that every link carries data simultaneously in every step, leaving no link idle. This uniform link utilization is what makes Ring AllReduce bandwidth-optimal: each node sends exactly $2(N-1)/N \cdot M$ bytes total across both phases, matching the information-theoretic lower bound. Section D.3.1 works through side-by-side examples that compare Ring, tree, and recursive halving-doubling and show how the best algorithm shifts with message size, giving the reader concrete crossover numbers to weigh against the per-algorithm derivations developed here.

To make the ring algorithm concrete, consider a step-by-step trace with 4 GPUs reducing a vector of 4 elements by summation. Each GPU i starts with a local gradient vector $g_i = [a_i, b_i, c_i, d_i]$. The vector is split into 4 chunks (one per GPU), and the algorithm proceeds through two phases.

Example 6.1: Ring AllReduce: Step-by-step trace (4 GPUs)

Setup: 4 GPUs in a ring $(0 \to 1 \to 2 \to 3 \to 0)$. Each GPU holds a vector of 4 values. We use concrete numbers for clarity:

- GPU 0: $[1, 5, 3, 7]$
- GPU 1: $[2, 6, 4, 8]$
- GPU 2: $[3, 7, 5, 9]$
- GPU 3: $[4, 8, 6, 10]$

9 **Bandwidth-Optimal**: Ring AllReduce achieves the information-theoretic lower bound of $2(N-1)/N \cdot M/\beta$ bytes per node, meaning no correct AllReduce algorithm can move fewer bytes per node regardless of topology or strategy. This optimality holds because every GPU must receive $M(N-1)/N$ bytes of "new" information from other GPUs and contribute the same amount, and Ring saturates every link in every step.

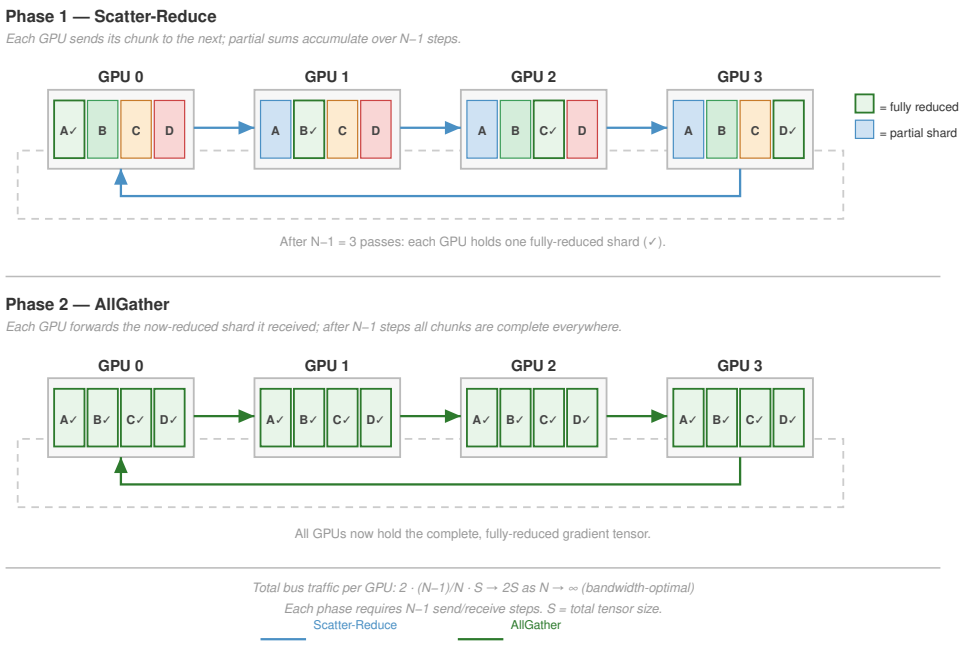


Figure 6.5: Ring AllReduce Data Flow: Nodes arranged in a logical ring exchange chunks in two phases: a Scatter-Reduce phase (each node accumulates one chunk's partial sum) followed by an AllGather phase (complete sums circulate to all nodes). The figure shows the ring laid out horizontally with a wrap-around bus closing the loop; every link is used simultaneously, achieving bandwidth optimality.

Expected result (element-wise sum): $[10, 26, 18, 34]$.

Each GPU "owns" one chunk: GPU 0 owns chunk A (element 0), GPU 1 owns chunk B (element 1), and so on.

Phase 1: Scatter-Reduce (3 steps)

Each GPU sends one chunk or partial sum to its right neighbor, which adds the received value to its local copy. Table 6.6 traces the three sequential steps across all four GPUs:

Table 6.6: Ring AllReduce Scatter-Reduce trace: Three sequential steps across 4 GPUs.

Step	GPU 0 sends	GPU 1 sends	GPU 2 sends	GPU 3 sends	After receive and add:
1	chunk A (1) → GPU 1	chunk B (6) → GPU 2	chunk C (5) → GPU 3	chunk D (10) → GPU 0	GPU 0: D=17, GPU 1: A=3, GPU 2: B=13, GPU 3: C=11
2	chunk D (17) → GPU 1	chunk A (3) → GPU 2	chunk B (13) → GPU 3	chunk C (11) → GPU 0	GPU 0: C=14, GPU 1: D=25, GPU 2: A=6, GPU 3: B=21
3	chunk C (14) → GPU 1	chunk D (25) → GPU 2	chunk A (6) → GPU 3	chunk B (21) → GPU 0	GPU 0: B=**26**, GPU 1: C=**18**, GPU 2: D=**34**, GPU 3: A=**10**

After Phase 1, each GPU holds the *complete sum* for exactly one chunk: GPU 0 has the global sum for element B (26), GPU 1 for C (18), GPU 2 for D (34), GPU 3 for A (10).

Phase 2: AllGather (3 steps)

Each GPU sends its fully reduced chunk around the ring so all GPUs receive all results. Table 6.7 shows the three forwarding steps that complete the collective:

Table 6.7: Ring AllReduce AllGather trace: Three sequential steps in which each GPU forwards its fully-reduced chunk around the ring.

Step	Action	Result
4	Each sends its complete chunk to right neighbor	Each GPU now has 2 of 4 final chunks
5	Continue forwarding	Each GPU now has 3 of 4 final chunks
6	Final forwarding	All GPUs hold $[10, 26, 18, 34]$

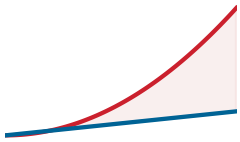
The total data movement per GPU is fixed by the ring schedule: in each of the $2(N-1) = 6$ steps, each GPU sends exactly $M/N = 1$ element. Total per GPU: $6 \times 1 = 6$ elements sent, 6 received.

The trace leaves two mechanics to verify before moving on: the $2(N-1)$ step count, and that every rank sends and receives in each step.

Checkpoint 6.2: Ring AllReduce mechanics

Verify your understanding of bandwidth-optimal reduction:

- ☐ **Step count**: Calculate the number of **sequential steps** required for a ring of N GPUs to complete the full AllReduce.
- ☐ **Bandwidth optimality**: Explain why Ring AllReduce is bandwidth-optimal, including how many times each byte of the total message M traverses the network per GPU.
- ☐ **Straggler impact**: Determine the impact on total collective completion time when one GPU in the ring is 50 percent slower than the others.
- ☐ **Send/receive duality**: Assess the True or False claim that every GPU is both a sender and a receiver in every step of Ring AllReduce.



Ring latency grows with N; tree stays logarithmic.

The trace above illustrates the key property of Ring AllReduce: at every step, every link in the ring is active, with data flowing in the same direction. No GPU ever sits idle, and no link is underutilized. This uniform link utilization is what makes Ring bandwidth-optimal.

The performance follows directly from the algorithm structure. Each node sends and receives $\frac{M}{N}$ bytes in each of the $2(N-1)$ steps.

$$T_{\text{ring}} = \underbrace{2(N-1)\alpha}_{\text{Latency Term}} + \underbrace{2\frac{N-1}{N}\frac{M}{\beta}}_{\text{Bandwidth Term}}$$

- **Bandwidth**: As $N \to \infty$, the term approaches $2M/\beta$. This is theoretically optimal (each byte must be sent once and received once).
- **Latency**: The latency scales linearly with N. For 10,000 nodes, 20,000 sequential hops creates massive latency. This is why Ring is bad for small messages.

6.4.4 Tree AllReduce

Ring AllReduce pays a high price in latency for its bandwidth optimality. When a cluster grows to thousands of GPUs and the message size is moderate (a few megabytes), the $2(N-1)\alpha$ latency term dwarfs the bandwidth term, and the algorithm spends most of its time in sequential hops rather than in useful data transfer. To address this linear latency, Tree AllReduce uses a binary tree structure that works in two phases. In the reduce phase, leaves send to their parents, which sum the incoming values and pass them up, reaching the root in $\log_2 N$ steps. In the broadcast phase, the root sends the result back down to the leaves in another $\log_2 N$ steps.

Tree AllReduce has worse bandwidth efficiency because of link underutilization. In Ring AllReduce, every node sends and receives simultaneously, so all N links are active at every step. In Tree AllReduce, only a fraction of links are active at any time, and that fraction shrinks at every level. At the leaves, $N/2$ nodes send to $N/2$ parents, leaving half the nodes as idle receivers; at the next level up, $N/4$ nodes send to $N/4$ parents, leaving three-quarters idle; at the root, only two nodes communicate while the remaining $(N-2)$ sit idle. The result is that while Ring achieves near-100 percent link utilization for large messages on a balanced ring, a simple tree can leave many links idle and concentrate traffic near the root or interior links.

The resulting time complexity depends on the specific tree variant. A simple reduce-then-broadcast tree has logarithmic latency but nonuniform traffic and root/interior bottlenecks; recursive-doubling-style variants can incur $\mathcal{O}(\log N)$ full-message bandwidth per rank. The simplified model below captures the latency advantage and the potential bandwidth penalty of tree-like collectives:

$$T_{\text{tree-like}} \approx \underbrace{2\log_2 N \cdot \alpha}_{\text{Latency}} + \underbrace{2\log_2 N \frac{M}{\beta}}_{\text{Bandwidth penalty in full-message exchange variants}}$$

The latency term is logarithmic ($\mathcal{O}(\log N)$): for 1024 nodes, Ring needs 2046 steps while Tree needs only 20, and that 100× reduction in steps makes Tree the preferred algorithm for latency-sensitive collectives such as tensor parallelism's per-layer AllReduce, where message sizes are small but frequency is high. The bandwidth term carries the penalty, because tree-like algorithms can underutilize links or concentrate traffic on interior links, and recursive-doubling-style full-message exchanges add a $\log_2 N$ bandwidth factor; these penalties make naive tree variants a poor choice for data parallelism's large gradient AllReduce, where bandwidth efficiency determines whether the network is saturated or partially idle.

This bandwidth penalty is catastrophic for large-scale data parallelism when the implementation repeatedly exchanges full messages or creates root/interior bottlenecks. For a 70B-parameter model's 140 GB gradient AllReduce across 1,000 GPUs, a bandwidth-inefficient tree-like variant can erase the latency advantage that motivated it. Optimized double-tree algorithms exist precisely to keep logarithmic latency without imposing that naive bandwidth penalty.

Tree AllReduce is, however, the correct choice when latency is the primary bottleneck, typically for smaller collectives with small message sizes. The canonical use case is a latency-sensitive AllReduce required by tensor parallelism, which operates on activations or weight gradients within a single layer. For an 8-GPU group inside a node communicating a small 1 MB tensor, the latency drops from Ring's $2(8-1)\alpha = 14\alpha$ to Tree's $2\log_2(8)\alpha = 6\alpha$. The bandwidth penalty is modest at that size, so reducing sequential startup steps can win.

6.4.5 Recursive halving-doubling (butterfly)

Ring and Tree represent two extremes of the latency-bandwidth trade-off: Ring is *bandwidth-optimal but latency-poor*, while Tree is *latency-optimal but bandwidth-poor*. A third approach combines the logarithmic latency of Tree with better bandwidth utilization. The **Recursive Halving-Doubling** algorithm (sometimes called the Butterfly algorithm) operates in $\log_2 N$ rounds. In each round k, every GPU exchanges data with a partner at distance 2^k in the logical numbering, and the message size halves (in the ReduceScatter phase) or doubles (in the AllGather phase).

In the ReduceScatter phase (first $\log_2 N$ rounds), GPU i partners with GPU $i \oplus 2^k$ (XOR of indices), and they exchange half of their current data. After receiving, each GPU sums its half with the received half, then discards the other half. After $\log_2 N$ rounds, each GPU holds M/N bytes of the fully reduced result. The AllGather phase reverses the process: in each round, partners exchange their reduced chunks, doubling the data each GPU holds until all GPUs have the complete result.

The performance of Recursive Halving-Doubling is:

$$T_{\text{butterfly}} = 2\log_2 N \cdot \alpha + 2\frac{N-1}{N} \cdot \frac{M}{\beta}$$

This achieves the best of both worlds: logarithmic latency ($\mathcal{O}(\log N)$, like Tree) and bandwidth-optimal data movement ($2(N-1)/N \cdot M/\beta$, like Ring). The catch is that it requires nonneighbor communication (GPU i must communicate with GPU $i \oplus 2^k$, which may be physically distant), and it requires N to be a power of two. For clusters where N is not a power of two, additional complexity is needed to handle the irregular cases.

Recursive halving-doubling is used in MPI-style collective implementations and is useful as a theoretical point in the latency-bandwidth trade-off (Thakur et al. 2005). NCCL's implementation families include ring, tree-derived, hierarchical, NVLink/NVSwitch-aware, and pattern-aware choices, with exact names and availability changing by version and topology (Jeaugey 2017; NVIDIA 2026b). The durable point is not the label on a particular release: practical communication libraries select algorithms through a topology-aware cost model because nonlocal communication patterns can create contention on shared network links that erodes theoretical advantages.

Sequence parallelism and Mixture of Experts routing expose the topology sensitivity of recursive halving-doubling most sharply. In sequence parallelism, AllGather reconstructs activation shards and ReduceScatter redistributes them along the sequence dimension—the inter-rank exchange schedule follows the same distance-doubling logic as the butterfly algorithm. In MoE routing, each token must reach any of N expert GPUs, creating the same fan-out pattern: a butterfly-style schedule produces cross-boundary pairings in the final round because token dispatch must traverse the full cluster diameter. The following eight-GPU ReduceScatter trace shows exactly how those long-distance crossings emerge.

To make this concrete, consider the ReduceScatter phase for 8 GPUs, which proceeds in $\log_2(8) = 3$ rounds. In round $k = 0$, each GPU i partners with GPU $i \oplus 1$, pairing neighbors: (0,1), (2,3), (4,5), (6,7). They exchange half their data and reduce. In round $k = 1$, the distance doubles: GPU i partners with GPU $i \oplus 2$, creating pairs (0,2), (1,3), (4,6), (5,7). In round $k = 2$, the distance doubles again: GPU i partners with GPU $i \oplus 4$, creating pairs (0,4), (1,5), (2,6), (3,7).

The problem with this pattern becomes clear on real hardware. The round $k = 2$ pairings force communication across physical boundaries. If GPUs 0–3 are on one node and 4–7 are on another, this final round creates cross-node traffic between the two nodes. For our 1,000-GPU cluster (approximated as 1,024 for the algorithm), the final round pairs GPU i with GPU $i \oplus 512$, forcing GPUs in the first half of the cluster to communicate with partners in the second half and potentially flooding the high-level network fabric that connects server racks. This topology-oblivious communication pattern is why Butterfly's theoretical optimality does not translate to practical superiority on large hierarchical clusters.

6.4.6 Double binary tree

NCCL can use a **Double Binary Tree** for many message sizes and topologies, which addresses the bandwidth inefficiency of a standard binary tree without requiring the nonlocal communication of

Butterfly (Jeaugey 2017; NVIDIA 2026b). The idea is to construct two independent binary trees that together cover all links, then run both trees simultaneously, each carrying half the data.

In a standard binary tree, at each level only half the links are active (the other half are idle because those nodes are receiving, not sending). By constructing a second, complementary tree (rooted at a different node, with edges that cover the links unused by the first tree), both trees can operate in parallel. Each tree carries $M/2$ bytes, and since their link utilization is complementary, the aggregate link utilization approaches 100 percent, matching Ring's bandwidth efficiency while retaining Tree's $\mathcal{O}(\log N)$ latency.

The combined performance is approximately:

$$T_{\text{double-tree}} \approx 2\log_2 N \cdot \alpha + \frac{2M}{\beta}$$

In practice, the bandwidth term approaches $2M/\beta$ (optimal) while maintaining logarithmic latency. This makes Double Binary Tree a strong choice across a wide range of message sizes and cluster counts, which is why communication libraries select it in many configurations. The algorithm requires careful construction of the two complementary trees to ensure they do not create link contention, a problem that NCCL's topology-aware graph search addresses during initialization.

The double binary tree's broad competitiveness across message sizes explains why PyTorch DDP and FSDP bucket sizes (typically 25–100 MB) frequently fall into exactly the crossover territory where neither Ring nor pure Tree is clearly optimal. A bucket at 25 MB sits near the Ring-Tree crossover for medium-scale clusters, and a bucket at 100 MB begins to favor Ring on slow networks. Double binary tree handles this middle ground well, which is one reason NCCL dynamically selects it for many DDP/FSDP gradient synchronization calls rather than committing unconditionally to Ring.

Table 6.8 summarizes the AllReduce algorithm comparison and the performance characteristics of the four algorithms. As figure 6.6 illustrates, the choice between algorithms depends on message size: Tree dominates for small messages while Ring wins for large ones.

Table 6.8: AllReduce Algorithm Comparison: Each algorithm occupies a different point in the latency-bandwidth trade-off space. Ring and Butterfly achieve bandwidth optimality with different latency characteristics, while Double Binary Tree provides the best practical compromise.

Algorithm	Latency	Bandwidth	Bandwidth Optimal?	Constraint
Ring	$\mathcal{O}(N)\alpha$	$2\frac{N-1}{N}\frac{M}{\beta}$	Yes	None
Tree	$\mathcal{O}(\log N)\alpha$	$\mathcal{O}(\log N)\frac{M}{\beta}$	No	None
Butterfly	$\mathcal{O}(\log N)\alpha$	$2\frac{N-1}{N}\frac{M}{\beta}$	Yes	$N = 2^k$
Double Tree	$\mathcal{O}(\log N)\alpha$	$\approx \frac{2M}{\beta}$	Near-optimal	Complementary tree construction

6.4.7 The algorithm crossover point

The crossover point in figure 6.6, where Ring overtakes Tree, follows from setting $T_{\text{ring}} = T_{\text{tree}}$ and solving for M. The full time equations make the trade-off explicit:

$$T_{\text{ring}} = 2(N-1)\alpha + 2\frac{N-1}{N}\frac{M}{\beta}$$

$$T_{\text{tree}} = 2\log_2 N \cdot \alpha + 2\log_2 N \cdot \frac{M}{\beta}$$

For large N, $(N-1) \approx N$ and $\frac{N-1}{N} \approx 1$, so the ring pays a latency term proportional to N while moving each byte close to the bandwidth limit:

$$T_{\text{ring}} \approx 2N\alpha + \frac{2M}{\beta}, \quad T_{\text{tree}} \approx 2\log_2 N \cdot \alpha + \frac{2\log_2 N \cdot M}{\beta}$$

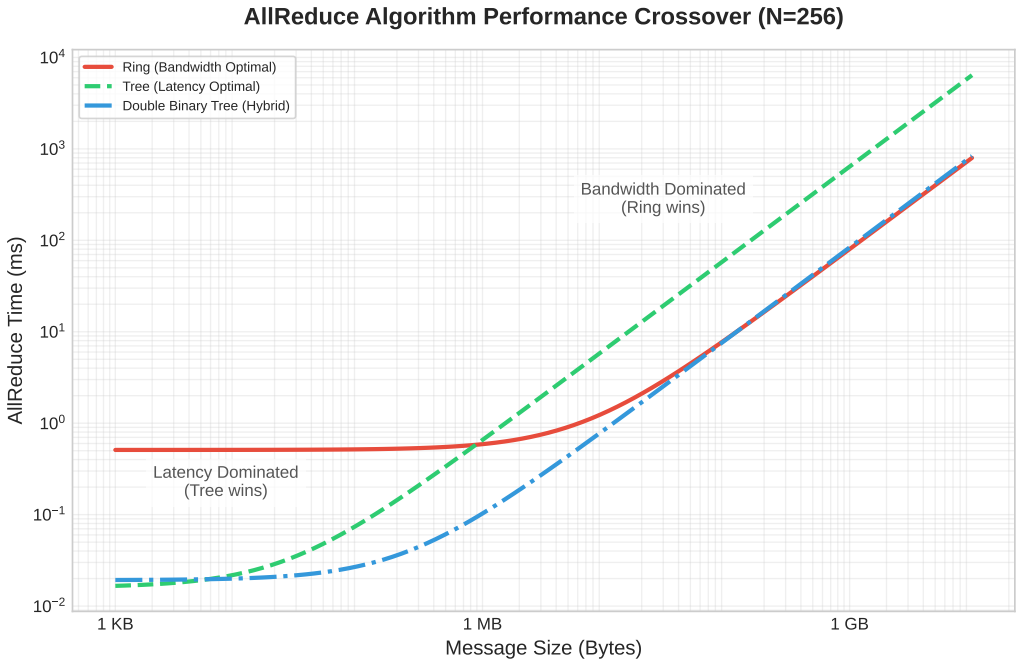


Figure 6.6: Algorithm Crossover in Collective Communication: Performance analysis of Ring, Tree, and Double Binary Tree AllReduce algorithms across message sizes (1 KB to 10 GB) for a 256-GPU cluster with 200 Gbps interconnect.

The tree has the opposite shape: logarithmic startup cost, but each message byte moves through more stages. Setting the estimates equal isolates the message size where the cheaper latency path stops compensating for the extra bandwidth work:

$$2N\alpha + \frac{2M}{\beta} = 2\log_2 N \cdot \alpha + \frac{2\log_2 N \cdot M}{\beta}$$

$$\frac{2M}{\beta} - \frac{2\log_2 N \cdot M}{\beta} = 2\log_2 N \cdot \alpha - 2N\alpha$$

$$\frac{2M}{\beta}(1 - \log_2 N) = 2\alpha(\log_2 N - N)$$

Since $N \gg \log_2 N$ for large clusters, $(\log_2 N - N) \approx -N$ and $(1 - \log_2 N) \approx -\log_2 N$, yielding:

$$\frac{2M}{\beta}(-\log_2 N) \approx 2\alpha(-N)$$

$$M_{\text{crossover}} \approx \frac{N \cdot \alpha \cdot \beta}{\log_2 N}$$

For a rough upper-scale mental model, engineers sometimes drop the logarithmic factor:

$$\boxed{M_{\text{crossover}} \approx N \cdot \alpha \cdot \beta}$$

The log-aware crossover formula yields a useful intuition, not a hard selection rule. Above the crossover, message size dominates and Ring's bandwidth efficiency wins; below it, startup latency dominates and Tree's logarithmic depth wins.

For a cluster with $\alpha = 5$ μs, $\beta = 50$ GB/s, and $N = 100$, the direct approximation gives:

$$M_{\text{crossover}} \approx \frac{100 \times 5 \cdot 10^{-6} \times 50 \cdot 10^{9}}{\log_2 100} \approx 3.8 \text{ MB}$$

In practice, NCCL's algorithm selection is more nuanced than a simple two-way split. The Double Binary Tree algorithm offers logarithmic latency with near-optimal bandwidth, making it competitive with both Ring and Tree across a broad range of message sizes. NCCL also considers the number of channels (parallel communication streams), the network topology, and whether inter-node or intra-node links are being used. The effective selection logic resembles a multi-way decision tree indexed by (message size, GPU count, topology type) rather than a single crossover point.

Nevertheless, the crossover formula remains the essential mental model for understanding *why* libraries make the choices they do. When a library selects an unexpected algorithm, the crossover analysis provides the reasoning framework to evaluate whether the choice is correct or whether manual override is warranted. A concrete buffer-size calculation applies this framework to a realistic decision:

Napkin Math 6.5: The ring vs. tree crossover

Problem: A cluster synchronizes a 1 MB buffer across 64 GPUs. The network has latency $\alpha = 10$ µs and bandwidth $\beta = 10$ GB/s. Which algorithm performs better: Ring or Tree?

Math:

Latency:

1. **Ring Latency**: $2(N-1)\alpha = 2 \times 63 \times 10\ \mu\text{s} = 1260$ µs.
2. **Tree Latency**: $2(\log_2 N)\alpha = 2 \times 6 \times 10\ \mu\text{s} = 120$ µs.

Bandwidth (note the difference):

3. **Ring Bandwidth**: $2\frac{N-1}{N}\frac{M}{\beta} \approx 2 \times \frac{1\text{ MB}}{10\text{ GB/s}} = 196.9$ µs (optimal: each byte sent once).
4. **Tree Bandwidth**: $2\log_2 N \cdot \frac{M}{\beta} = 12 \times \frac{1\text{ MB}}{10\text{ GB/s}} = 1200$ µs (each level sends full message).

Table 6.9 sums the latency and bandwidth contributions into a head-to-head comparison:

Table 6.9: Ring vs. tree AllReduce timing for a 1 MB buffer on 64 GPUs: Latency, bandwidth, and total-time terms with $\alpha = 10\ \mu$s and $\beta = 10$ GB/s.

Algorithm	Latency	Bandwidth	Total
Ring	1260 µs	196.9 µs	1456.9 µs
Tree	120 µs	1200 µs	1320 µs

Tree wins, but only by 9 percent. For this 1 MB message, we are near the crossover point.

Systems insight: Ring's latency penalty (10× worse than Tree) nearly balances Tree's bandwidth penalty (6× worse than Ring). The log-aware crossover estimate predicts $M_{\text{crossover}} \approx N\alpha\beta/\log_2 N = 64 \times 10\ \mu\text{s} \times 10\text{ GB/s}/6 \approx 1.1$ MB. At 1 MB, we are just below crossover, so Tree wins. At 10 MB, Ring would dominate.

The crossover point sits inside the range that DDP and FSDP workloads traverse during normal training. A 1 MB message corresponds to gradients from a single feed-forward layer in a smaller transformer, or an MoE token-routing payload—Tree territory. A 10 MB DDP gradient bucket from a mid-size transformer layer sits near the crossover. A fused DDP bucket covering multiple layers reaches 50–300 MB; a full BF16 gradient tensor for a 70B parameters model reaches 1120 GB—both firmly in Ring territory. Most real training workloads sweep through this entire range as bucket sizes are tuned, and the crossover analysis sets the algorithmic boundary that separates bucket-level Tree operations from full-model Ring synchronizations.

The crossover analysis demonstrates that algorithm selection is not a static choice but depends on the specific combination of message size, cluster scale, and network parameters. In practice, communication libraries like NCCL maintain internal lookup tables that map message size, GPU count, topology, and protocol settings to selected algorithms such as Ring, Tree, or hybrid approaches.

Understanding the underlying crossover math allows engineers to predict when the library's built-in choices may be suboptimal and to override them when necessary.

Checkpoint 6.3: AllReduce algorithm selection

Verify your understanding of Ring vs. Tree AllReduce trade-offs:

- ☐ **Ring vs. Tree**: Explain why Ring AllReduce achieves bandwidth-optimal communication (approaching $2M/\beta$) while Tree AllReduce incurs $\mathcal{O}(\log N)$ bandwidth overhead.
- ☐ **Crossover point**: Calculate $M_{\text{crossover}}$ for a 256-GPU cluster with $\alpha = 5\,\mu$s and $\beta = 50$ GB/s, then determine whether a 10 MB gradient buffer should use Ring or Tree.
- ☐ **Scatter-reduce trace**: Trace through the Scatter-Reduce phase of Ring AllReduce for 3 GPUs with a 3-element vector and verify that each GPU ends with the correct partial sum.
- ☐ **Latency limit**: Explain why Ring AllReduce's $\mathcal{O}(N)$ latency makes it impractical for tensor parallelism, where message sizes are small and latency dominates.

6.5 Hierarchical Communication

The algorithms above assume a flat network where every link has the same bandwidth. Real data centers violate this assumption by an order of magnitude or more. Not all wires are created equal.

Hierarchical communication is the response to that inequality. Intra-node links are abundant and fast, inter-node links are scarce and expensive, and the algorithm must shrink payloads before they cross the scarce tier whenever possible. This is why the same AllReduce can be implemented as a local ReduceScatter, a cross-node reduction, and a local AllGather rather than as one flat exchange across every rank.

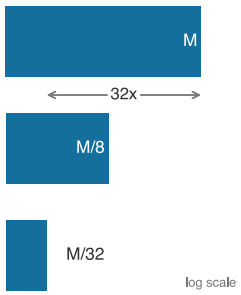
A GPU node delivers an order of magnitude more intra-node bandwidth (NVLink at 900 GB/s) than inter-node bandwidth (InfiniBand at 50 GB/s).

6.5.1 Hierarchical AllReduce

The algorithms above (Ring, Tree, Butterfly, Double Binary Tree) all assume a flat network where every link has the same bandwidth. This assumption holds within a single node (where all GPUs are connected by NVLink at equal bandwidth) but fails by 9× in multi-node clusters. Real clusters are hierarchical networks, with fundamentally different bandwidths at each tier, as table 6.10 quantifies.

Table 6.10: The Bandwidth Hierarchy: Modern GPU clusters exhibit an order-of-magnitude bandwidth gap between intra-node and inter-node communication. Hierarchical algorithms exploit this gap. Relative speed compares per-direction rates (NVLink's 450 GB/s one-way against InfiniBand's 50 GB/s per port); the bandwidth column shows NVLink's bidirectional datasheet total.

Tier	Interconnect	Bandwidth	Relative Speed
Intra-Node	NVLink 4.0	~900 GB/s	9× faster
Inter-Node	InfiniBand NDR 400G	~50 GB/s	1× (baseline)



Hierarchical collectives shrink payload before it crosses slow tiers.

A naive flat Ring AllReduce ignores this structure, potentially routing data across InfiniBand when NVLink would suffice and wasting the scarce inter-node bandwidth. **Hierarchical AllReduce** decomposes the global operation into three phases that respect the bandwidth hierarchy, as shown in figure 6.7.

The three-phase decomposition in figure 6.7 confines the expensive inter-node traffic to Phase 2, where each GPU transmits only M/G bytes instead of M. With $G = 8$ GPUs per node (a typical DGX configuration), this reduces inter-node traffic by 8×, effectively multiplying the scarce InfiniBand bandwidth by the number of GPUs per node.

The phases form one causal chain:

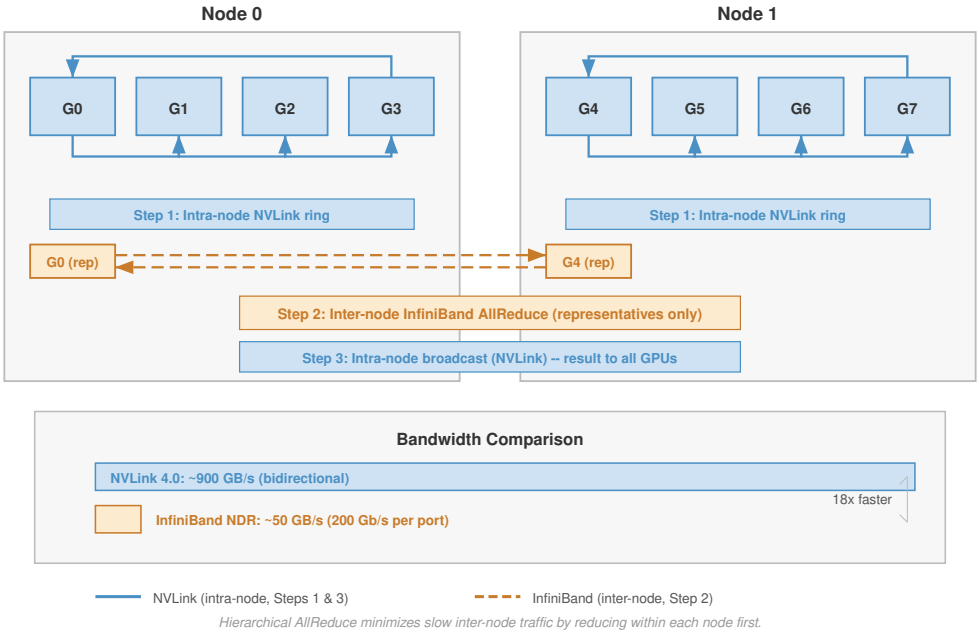


Figure 6.7: Hierarchical AllReduce: Three-phase decomposition that exploits the intra-node/inter-node bandwidth gap. Phase 1 performs intra-node aggregation over NVLink (an NVLink ring acting as a ReduceScatter that leaves each GPU holding a node-shard partial sum), reducing inter-node traffic by a factor of G (GPUs per node). Phase 2 runs an AllReduce across corresponding GPUs using InfiniBand. Phase 3 completes intra-node distribution over NVLink (the AllGather that returns each GPU to the full result). The figure labels these phases by their NVLink/IB mechanics; the prose below names the corresponding collective primitives.

1. **Intra-node ReduceScatter over NVLink**: A local reduction among the G GPUs in each node leaves each GPU with $1/G$ of the partially reduced data, using only abundant intra-node bandwidth.
2. **Inter-node AllReduce over InfiniBand**: Each GPU sends its node shard across corresponding GPU positions, transmitting M/G bytes instead of M.
3. **Intra-node AllGather over NVLink**: Each node redistributes the final shards internally, completing the result without consuming additional inter-node bandwidth.

A simple 8-node budget makes this bandwidth multiplication concrete:

> **Napkin Math 6.6: The hierarchical bandwidth multiplier**
>
> **Problem**: A cluster has 8 nodes, each with 8 GPUs (64 total). The system must AllReduce a 1 GB gradient buffer. Compare flat Ring AllReduce vs. Hierarchical AllReduce.
>
> **Flat ring AllReduce (ignoring hierarchy)**:
>
> - Each GPU sends ~2 GB total (the bandwidth-optimal Ring AllReduce formula).
> - The ring crosses node boundaries multiple times.
> - **Effective bandwidth**: Limited by the slowest link = 50 GB/s (InfiniBand).
> - **Time**: $\approx 2 \times 1$ GB/50 GB/s = 40 ms (bandwidth term dominates).
>
> **Hierarchical AllReduce (3-step decomposition)**:
>
> 1. **Intra-node ReduceScatter**: Each GPU sends 875 MB at 450 GB/s per direction → ~1.94 ms

2. **Inter-Node AllReduce**: Each GPU AllReduces a 1 GB/8 = 125 MB shard over InfiniBand; Ring moves roughly $2(N-1)/N$ times that payload, and after adding the small latency term the phase takes ~4.42 ms at 50 GB/s
   (Only 1/8 of the data crosses InfiniBand!)
3. **Intra-node AllGather**: Each GPU receives 875 MB at 450 GB/s per direction → ~1.94 ms
4. **Total time**: ≈ 1.94 ms + 4.42 ms + 1.94 ms = 8.3 ms

Systems insight: Hierarchical AllReduce achieves a 4.8× speedup by reducing the inter-node payload from 1 GB to 125 MB per GPU before the Ring traffic multiplier. With 8 GPUs per node, we effectively get 8× the apparent inter-node bandwidth. This is why NVIDIA's NCCL and similar libraries often select hierarchical algorithms on multi-node clusters when the topology supports them.

These three phases confine most traffic within each node before crossing the slower inter-node fabric, and the same idea generalizes beyond two levels. Large clusters with multiple racks connected through spine switches introduce a third bandwidth tier (rack-to-rack at reduced bisection bandwidth). At 128 GPUs arranged as 4 racks of 4 nodes of 8 GPUs with 2:1 cross-rack oversubscription, applying the same decomposition at each tier shrinks the cross-rack payload per GPU by 32× compared to the original gradient—roughly 15 ms total versus 160 ms for flat AllReduce. The hierarchical decomposition concentrates traffic where bandwidth is abundant and minimizes traffic where bandwidth is scarce.

6.5.2 In-network reduction: SHARP and beyond

The contrast in figure 6.8 shows the next step: move the reduction work from endpoint GPUs into the switch ASIC so packets are aggregated while they traverse the fabric.

Hierarchical AllReduce reduces the volume of cross-node traffic, but the aggregation still requires multiple network round-trips. NVIDIA's **Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)**[10] implements this idea: instead of gradients traveling to a destination GPU for summation, the InfiniBand switch aggregates partial sums as data packets pass through it.

The benefit is twofold. First, SHARP reduces endpoint memory traffic and can reduce pressure on upper-level links by aggregating packets before they leave a switch tier. In a software-based tree reduction, data arrives at a GPU, is written to memory, summed with local data, and then transmitted to the next level. With SHARP, the switch combines incoming packets in flight and forwards the aggregate for the reduction tree. Second, SHARP eliminates the store-and-forward path through intermediate GPUs. Each software hop adds the full α-β cost; with in-switch aggregation, the reduction happens inside the switch ASIC rather than in endpoint memory.

The practical impact is most pronounced for message sizes where aggregation latency contributes meaningfully to total AllReduce time. For very large messages, bandwidth dominates regardless of algorithm, and SHARP's latency reduction becomes proportionally smaller. For very small messages, fixed switch-processing overhead can limit the benefit. Graham et al. (2020) report multi-fold reduction-bandwidth improvements and smaller application-level gains for PyTorch workloads when SHARP Streaming-Aggregation is enabled on HDR/Quantum InfiniBand switches.

Strong scaling campaigns—training a fixed model across more accelerators without proportionally growing batch size—are precisely the workloads where SHARP's latency reduction translates into step-time reduction. As the per-accelerator batch shrinks, each gradient tensor stays large (proportional to model size) but the backward pass per accelerator shortens, so communication dominates an increasing fraction of step time and the latency term α in the ring formula hits the bottleneck. In-switch aggregation cuts directly into that latency cost. Switch ASIC vendors have explicitly extended InfiniBand ASICs to support ML-native datatypes, including FP16 and BF16, confirming that SHARP is co-evolving with ML workload requirements rather than serving only the legacy HPC floating-point operations that motivated its original design. An ML system team choosing SHARP infrastructure for a scaling campaign should verify FP16/BF16 aggregation support in the specific switch generation, since earlier ASICs supported only FP32 and INT32.

10 **SHARP (Scalable Hierarchical Aggregation and Reduction Protocol)**: In-network computing on Quantum InfiniBand switches that performs reduction operations (such as sum, min, and max) as packets traverse an aggregation tree (Graham et al. 2020). Graham et al. report substantially higher reduction bandwidth than host-based algorithms and describe switch-resource limits on outstanding aggregation groups and trees, which can create contention when many collectives compete for the same in-network resources.

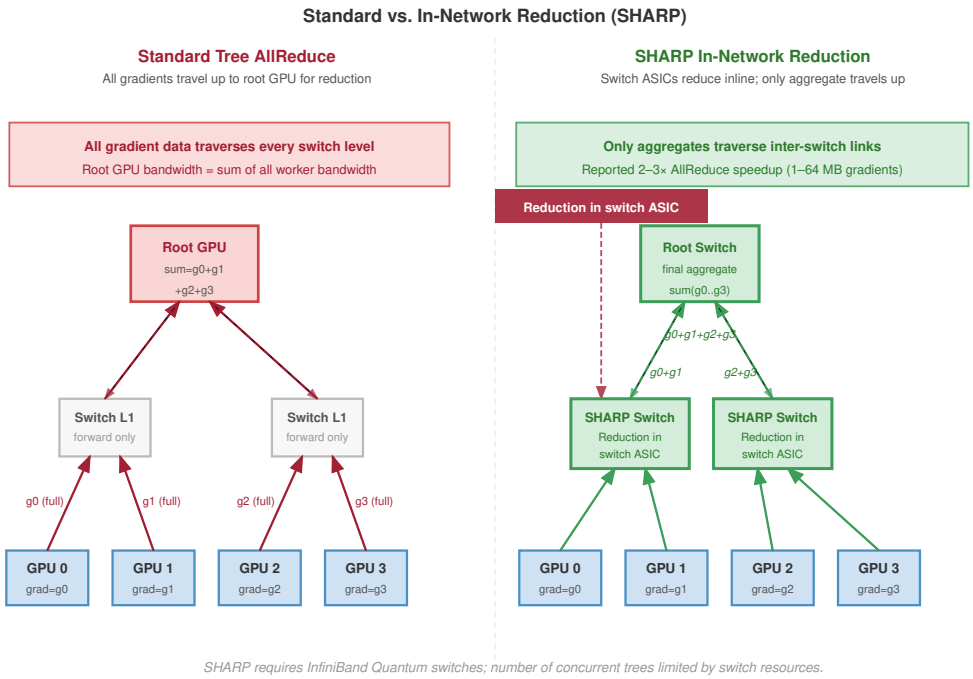


Figure 6.8: In-Network Reduction (SHARP): Side-by-side comparison split by a vertical divider. The left side shows a traditional tree AllReduce performing reduction at intermediate GPUs, incurring multiple store-and-forward delays. The right side shows SHARP offloading the reduction to the network switch ASIC, allowing partial sums to be aggregated at line rate as packets traverse the switch. This eliminates GPU memory traffic and store-and-forward latency, significantly accelerating small-to-medium message collectives.

SHARP does impose constraints. The switch must support the specific reduction operation (typically limited to sum, min, and max on floating-point and integer types). The number of concurrent SHARP aggregation trees is limited by switch resources, so large clusters with many simultaneous training jobs may exhaust SHARP capacity. Additionally, SHARP requires InfiniBand infrastructure; it is not available on Ethernet-based fabrics.

6.5.3 Topology-aware routing

Hierarchical AllReduce reduces inter-node traffic, and SHARP eliminates some of it entirely, but neither technique addresses a subtler problem: how the logical communication pattern maps to the physical network topology. A Ring AllReduce among 64 GPUs creates a logical ring, but which physical links carry each hop determines whether the ring achieves peak bandwidth or creates congestion. Two runs of the same training job on the same hardware can differ by $2\times$ in communication throughput depending on how ranks are assigned to GPUs and how the resulting traffic pattern interacts with the network's physical structure.

Communication libraries like NCCL perform **topology detection** at initialization, running graph search algorithms to discover the physical network structure and find high-bandwidth communication paths (NVIDIA 2026b). The library must determine, for each pair of ranks, whether the peer sits on a local NVLink switch or 100 meters away across an InfiniBand fabric, because the same collective algorithm can differ by $2\times$ or more in throughput depending on how logical ranks map to physical hardware.

The topology detection process begins with hardware enumeration. NCCL queries the PCIe bus to discover GPU placement, NVLink connectivity between GPUs, and NIC-to-PCIe-switch affinity. From this information, it constructs an internal graph where nodes represent GPUs and edges represent physical links with annotated bandwidth and latency. A graph search algorithm then finds communication paths that maximize aggregate bandwidth while minimizing the number of cross-

domain hops (transitions between NVLink, PCIe, and InfiniBand domains). This topology-aware path selection is what allows NCCL to approach theoretical peak bandwidth on well-configured systems, while topology-unaware implementations can leave a large fraction of bandwidth unused.

6.5.3.1 Torus topology (TPU pods)

The dimension-ordered reduction in figure 6.9 shows why torus fabrics use the physical mesh directly instead of pretending the pod is a flat all-to-all network.

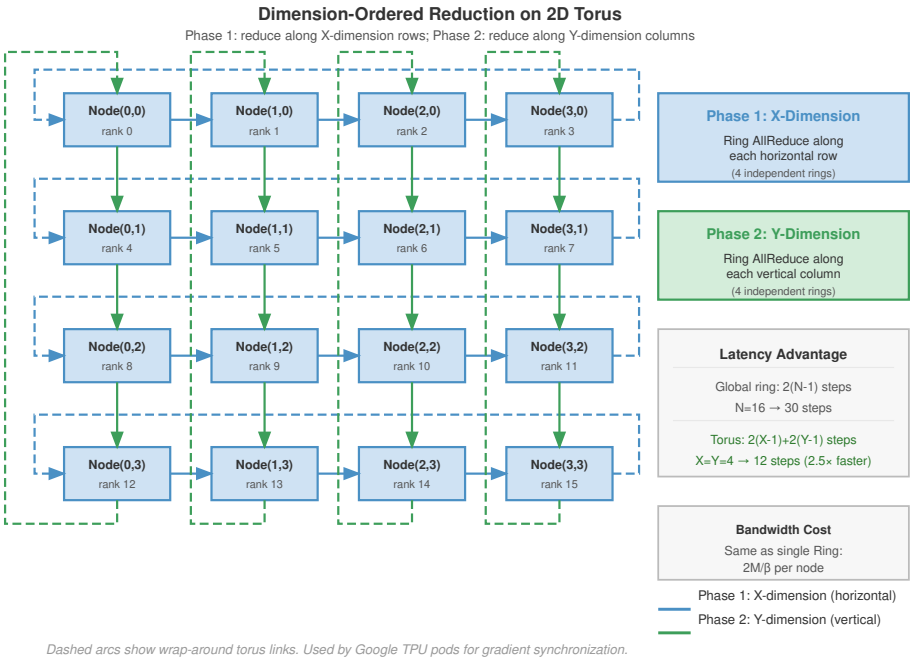


Figure 6.9: Dimension-Ordered Reduction on Torus Topology: In a 2D torus mesh (a simplified representation of 3D TPU pods), AllReduce is decomposed into two sequential steps along the X and Y dimensions. By reducing along one dimension at a time, the system minimizes network diameter and avoids link contention, ensuring that each direct neighbor link operates at full bandwidth.

Google's Tensor Processing Unit (TPU) pods use a 3D torus topology where each TPU connects directly to 6 neighbors $(\pm X, \pm Y, \pm Z)$; figure 6.9 depicts the same dimension-ordered idea as a 2D simplification. Unlike the hierarchical fat-tree topology of InfiniBand clusters, the torus provides uniform, direct connectivity: every TPU chip has the same number of links (6) and the same per-link bandwidth, regardless of its position in the mesh. The optimal AllReduce strategy for this topology is **dimension-ordered reduction**:

1. Reduce along X for each fixed (Y, Z) line, producing partial sums over the X dimension
2. Reduce those partials along Y for each fixed (X, Z) line, producing partial sums over the $X \times Y$ plane
3. Reduce those partials along Z for each fixed (X, Y) line to complete the global result

Each dimension-ordered reduction is itself a Ring AllReduce along one axis-aligned torus ring. For a pod with $X \times Y \times Z$ TPUs, the first step runs independent X-dimension rings, one for each fixed (Y, Z) coordinate, so each participant receives a partial sum over that X line. The second step runs Y-dimension rings to combine those X partials across Y, and the third step runs Z-dimension rings to complete the global reduction. This cascading reduction achieves total bandwidth cost $2M/\beta$ (same as a single Ring) but with latency proportional to $2(X + Y + Z)$ rather than $2(X \times Y \times Z)$, because each dimension's ring is shorter than a global ring.

Dimension-ordered reduction minimizes network diameter and ensures each link carries traffic in only one direction at a time, avoiding congestion. The torus topology also provides natural fault

tolerance through alternate routing paths: if one link in the X-dimension fails, traffic can detour through the Y or Z dimensions (at the cost of increased latency). Google's Accelerated Linear Algebra (XLA) compiler generates dimension-ordered collectives automatically when targeting TPU pods, abstracting the topology details from the user.

6.5.3.2 Rail-optimized routing (NVIDIA DGX)

In NVIDIA DGX systems, each GPU has its own dedicated NIC (network interface card). Rail-optimized routing exploits this by ensuring that GPUs at the same position within their respective nodes communicate only with each other:

- GPU 0 on Node A talks only to GPU 0 on Node B, Node C, and so on.
- GPU 1 on Node A talks only to GPU 1 on other nodes.
- And so on for GPUs 2–7.

This creates 8 independent "rails" of communication that operate in parallel without contention, as table 6.11 illustrates.

Table 6.11: Rail-Optimized Traffic Distribution: Each rail carries $1/8$ of the total traffic independently.

Rail	Participants	Traffic
Rail 0	Node0-GPU0 Node1-GPU0 Node2-GPU0 …	$M/8$ each
Rail 1	Node0-GPU1 Node1-GPU1 Node2-GPU1 …	$M/8$ each
…	…	…
Rail 7	Node0-GPU7 Node1-GPU7 Node2-GPU7 …	$M/8$ each

Rail alignment is critical because without it, all 8 GPUs on a node might try to send to the same remote GPU simultaneously, creating $8\times$ contention on a single NIC. Rail-aligned routing ensures each NIC handles exactly $1/8$ of the traffic, achieving full bisection bandwidth utilization.

Rail-optimized routing and hierarchical AllReduce reinforce each other. In the hierarchical decomposition described earlier, Step 2 (inter-node AllReduce) naturally aligns with rail topology. GPU i on each node communicates only with GPU i on other nodes, which is precisely the rail pattern. This alignment is not coincidental; NVIDIA designed the DGX hardware with rail-optimized communication in mind, and NCCL's hierarchical algorithms can exploit this structure when the topology is detected and ranks are mapped correctly. When the logical communication pattern matches the physical topology, each NIC carries its intended share of traffic and the cluster can approach its theoretical peak bisection bandwidth.

Misalignment between logical and physical topology is a common source of performance degradation that is difficult to diagnose without careful profiling. If process ranks are assigned arbitrarily (for example, by the job scheduler without topology awareness), the hierarchical AllReduce may route cross-node traffic through the wrong NICs, creating hotspots that reduce effective bandwidth by 2–4$\times$. Large deployments use topology-aware rank assignment (configured through NCCL's `CUDA_VISIBLE_DEVICES` and the scheduler's GPU binding policies) to ensure alignment.

The hierarchical approach, combined with topology-aware routing, represents the culmination of the algorithms explored in this section: the α-β model identifies the bottleneck (inter-node bandwidth), hierarchical decomposition reduces traffic across that bottleneck, and rail optimization ensures the reduced traffic flows without contention. Together, these techniques close the gap between theoretical peak bandwidth and achieved bandwidth on well-configured clusters.

6.6 Gradient Compression Under Bandwidth Scarcity

Even after topology tuning and algorithm selection, a system may remain bandwidth-bound because the physical fabric cannot move gradient payloads fast enough. In that regime, sending fewer bits becomes a deliberate design choice rather than a universal last resort. The previous section attacked the communication bottleneck from the scheduling and topology side; payload compression attacks the same bottleneck by changing what crosses the fabric.

The bandwidth wall arises most acutely in two scenarios: training across data centers connected by wide-area networks (where bandwidth is 10–100× lower than InfiniBand), and training on cloud instances with commodity Ethernet networking where RDMA is unavailable or constrained. In these bandwidth-constrained settings, gradient compression techniques can reduce communication volume by 4–1000×, at the cost of introducing noise into the optimization process. The central tension is between bandwidth reduction and the noise tolerance of the optimization process—how much compression the optimizer can absorb before convergence is compromised.

6.6.1 Quantization: Reducing precision

Quantization is the least disruptive compression lever when bandwidth is binding because it keeps every gradient element but reduces the bits used to represent it. Most gradients are computed in FP32 (32-bit floating point) or BF16 (16-bit brain float), so reducing bit-width directly reduces communication volume. The useful question is how much gradient fidelity the optimizer can absorb, and the progression from mild to aggressive quantization illustrates the trade-off between compression ratio and gradient fidelity:

1. **FP16 (16-bit)**: A common baseline for accelerator training. Half the bits of FP32, with minimal impact on convergence for many models. Provides 2× compression over FP32.
2. **INT8 (8-bit)**: Quantize each gradient vector to 256 discrete levels. This requires computing a scaling factor per tensor: $g_{\text{int8}} = \text{round}(g/s)$ where $s = \max(|g|)/127$. The receiver reconstructs $\hat{g} = g_{\text{int8}} \times s$. Provides 4× compression over FP32 but introduces quantization noise proportional to the gradient magnitude.
3. **1-bit SGD**: The extreme case: transmit only the sign of each gradient element (+1 or -1). The receiver reconstructs using a learned or adaptive scaling factor. This achieves 32× compression over FP32 but introduces substantial noise that can degrade convergence without additional mechanisms (Seide et al. 2014; Karimireddy et al. 2019).

6.6.1.1 Block quantization for gradient communication

The quantization progression above uses a single scaling factor per entire tensor, which implicitly assumes the gradient distribution is uniform across the tensor. In practice, gradient distributions are highly nonuniform: attention layers produce gradients with heavy tails, embedding layers produce extremely sparse gradients, and normalization layers produce gradients concentrated near zero. A single scaling factor per tensor is suboptimal when gradient distributions vary across the tensor. Regions with small gradients lose most of their information when quantized with a scaling factor dominated by the tensor's maximum value. **Block quantization** addresses this by dividing the gradient tensor into blocks of b_{block} elements (typically $b_{\text{block}} = 64$ or 128) and computing an independent scaling factor per block. Each block's scaling factor adapts to the local gradient distribution, reducing quantization error at the cost of transmitting $d_{\text{grad}}/b_{\text{block}}$ additional scaling factors.

For a gradient vector of dimension d_{grad} quantized to INT8 with block size b_{block}, the message size is $d_{\text{grad}} \times 1$ byte for the quantized values plus $(d_{\text{grad}}/b_{\text{block}}) \times 4$ bytes for the FP32 scaling factors, or $d_{\text{grad}} + 4d_{\text{grad}}/b_{\text{block}}$ bytes in total. The effective compression is therefore $4d_{\text{grad}}/(d_{\text{grad}} + 4d_{\text{grad}}/b_{\text{block}}) = 4b_{\text{block}}/(b_{\text{block}} + 4)$, which for $b_{\text{block}} = 128$ reaches 3.88×, close to the theoretical 4×. The quality gain comes from replacing the global error bound: block quantization reduces the maximum quantization error from $s \cdot 0.5$, where s is the global scaling factor, to $s_b \cdot 0.5$, where s_b is the block-local scaling factor, which for the heavy-tailed gradient distributions common in attention layers can cut quantization error by 2–5× compared to per-tensor quantization.

Interfaces that fuse quantize-reduce operations make block quantization practical in communication compression, but each reduction in bit-width introduces quantization noise. This noise acts as a biased perturbation to the true gradient direction. For aggressive quantization (INT8 and especially 1-bit), the systematic bias can prevent convergence unless the system carries an error-feedback correction across steps. Quantization should therefore move only as far down the precision ladder as the convergence budget permits.

6.6.2 Sparsification: Transmitting only important gradients

Quantization attacks communication volume by reducing the number of bits per gradient element while transmitting every element. An orthogonal approach, **sparsification**, attacks the problem from the other direction: keep full precision but transmit only a subset of gradient elements, setting the rest to zero.

6.6.2.1 Top-k sparsification

The most common method is **Top-k compression**: for a gradient vector $g \in \mathbb{R}^{d_{\text{grad}}}$, transmit only the K_{top} elements with the largest absolute magnitude, setting the rest to zero (Aji and Heafield 2017; Lin et al. 2018):

$$\text{TopK}(g) = g \odot \mathbf{1}_{|g| \geq |g|_{(K_{\text{top}})}}$$

where $|g|_{(K_{\text{top}})}$ is the K_{top}-th largest element by magnitude. With $K_{\text{top}} = 0.001 \times d_{\text{grad}}$ (keeping only 0.1 percent of elements), this achieves $1000\times$ compression.

The compression headline must be discounted by the cost of encoding the sparse representation. Transmitting a sparse gradient requires sending both the nonzero values and their indices. For a gradient vector of dimension d_{grad} with K_{top} nonzero elements, the encoded message contains K_{top} values (each in FP32 or FP16) plus K_{top} indices (typically INT32). The total message size is $K_{\text{top}} \times (4+4) = 8K_{\text{top}}$ bytes for FP32 values with INT32 indices. The effective compression ratio is therefore $4d_{\text{grad}}/(8K_{\text{top}}) = d_{\text{grad}}/(2K_{\text{top}})$.

For $K_{\text{top}} = 0.001 d_{\text{grad}}$, the encoded message yields a compression ratio of $500\times$, not the $1000\times$ suggested by the kept-element fraction alone, because the index overhead is significant. Reducing the index size to INT16 (for $d_{\text{grad}} < 65536$) or using run-length encoding for structured sparsity patterns can improve the effective ratio.

Top-k is not the only sparsification rule. **Random-k sparsification** selects K_{rand} elements uniformly at random and scales them by $d_{\text{grad}}/K_{\text{rand}}$ to maintain an unbiased gradient estimate. Random-k has the advantage of being an unbiased compressor even without error feedback, because $\mathbb{E}[\text{RandomK}(g)] = g$. However, it introduces higher variance than Top-k because it discards large gradients with the same probability as small ones. In practice, Top-k with error feedback converges faster than Random-k for the same compression ratio, because Top-k preserves the most informative gradient components at each step.

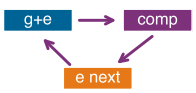
6.6.2.2 The convergence problem

Naively discarding small gradients creates a systematic bias. If a parameter consistently receives small gradients (e.g., 0.01 per step), it will not be updated because 0.01 is always below the Top-k threshold. Over thousands of steps, these "lost" gradients accumulate to a significant error that prevents the model from converging to the true optimum.

The problem is not merely practical; it is a fundamental mathematical obstacle. Without correction, sparsified SGD is a *biased estimator* of the true gradient, and biased gradient descent can converge to arbitrarily wrong solutions. The same problem afflicts aggressive quantization: rounding gradients to INT8 or 1-bit introduces a systematic rounding error that accumulates across training steps. Error feedback addresses this accumulation for quantized gradients (Seide et al. 2014) and for sparsified gradients (Stich et al. 2018; Karimireddy et al. 2019).

6.6.3 Error feedback: The memory of the journey

We solve the conflict between compression and convergence with error feedback. The system applies the compressor immediately to recover bandwidth, then stores the discarded residual in a local accumulator and re-injects it into the next gradient. *The error is deferred, not destroyed.* We maintain a local error accumulator e_t that stores the compression residual.



Error feedback carries the residual into the next compressed step.

Definition 6.4: Error feedback mechanism

Error Feedback is a distributed-training technique for gradient compression that maintains a per-worker residual accumulator e_t, re-injecting the compression error back into the next gradient update so that information deferred by the compressor is never permanently discarded.

1. **Significance**: Top-k sparsification at 1 percent keeps only the largest 1 percent of gradient values by magnitude, reducing AllReduce data volume from 140 GB to 1.4 GB for a 70B model—a 100× payload reduction before sparse-index overhead. Without error feedback, the discarded 99 percent of gradient values create a biased update that can harm convergence; with error feedback, the residual accumulates until deferred components cross the compression threshold in later steps, recovering the convergence guarantees analyzed for memory/error-feedback variants under their assumptions (Stich et al. 2018; Karimireddy et al. 2019).
2. **Distinction**: Unlike lossy compression (which permanently discards sub-threshold gradient components), error feedback is a delayed transmission strategy: the residual e_t telescopes across steps so that $\frac{1}{K}\sum_{t=1}^{K} v_t \to \frac{1}{K}\sum_{t=1}^{K} g_t$ as $K \to \infty$, making the long-run average of transmitted gradients equal to the true gradient average.
3. **Common pitfall**: A frequent misconception is that error feedback is automatic whenever a compressor is used. It is not: the training system must explicitly preserve and reinject the residual. Without that residual path, Top-k and 1-bit quantization introduce systematic bias that can accumulate across steps (Karimireddy et al. 2019).

Error feedback preserves cumulative gradient information over time. Consider what happens over K steps:

$$\sum_{t=1}^{K} v_t = \sum_{t=1}^{K} [(g_t + e_t) - e_{t+1}] = \sum_{t=1}^{K} g_t + e_1 - e_{K+1}$$

If the error accumulator remains bounded (which it does for reasonable compression schemes), then as $K \to \infty$:

$$\frac{1}{K}\sum_{t=1}^{K} v_t \to \frac{1}{K}\sum_{t=1}^{K} g_t$$

The long-run average of transmitted gradients equals the long-run average of true gradients. No gradient information is permanently lost; it is merely delayed. Small gradients that are repeatedly dropped eventually accumulate in e_t until they exceed the compression threshold and get transmitted.

The telescoping property of compression error across time is why error feedback can transform a biased compressor into a convergent training method. Theoretical analyses show convergence guarantees for sparsified SGD with memory and for EF-SGD with broad compression operators under their stated assumptions (Stich et al. 2018; Karimireddy et al. 2019). A short trace shows how error feedback preserves gradient information that naive compression would lose. Table 6.12 runs the trace for a greedy compressor with no residual path, transmitting only values ≥ 0.5:

Table 6.12: Naive compression trace (no error feedback): Five-step worked trace showing how a greedy 1-bit quantizer transmits zero whenever individual gradients fall below the threshold.

Step	True Gradient g_t	Transmitted v_t	Cumulative Transmitted	Cumulative True
1	0.4	0	0	0.4
2	0.3	0	0	0.7
3	0.2	0	0	0.9
4	0.4	0	0	1.3
5	0.3	0	0	1.6

The error-feedback trace in table 6.13 runs the same sequence with a residual accumulator e_t that re-injects the discarded remainder into the next step. The example below works through both traces to show that the deferred information is conserved rather than lost.

Example 6.2: Error feedback mechanism

Problem: A single parameter receives small gradients (all below 0.5) over five training steps. A greedy compressor transmits only values ≥ 0.5 (rounding to nearest integer: values in $[0, 0.5) \to 0$, values in $[0.5, 1.5) \to 1$, etc.). After five steps, how much gradient information does naive compression lose, and how does an error-feedback accumulator recover it?

Naive compression outcome: In table 6.12, after 5 steps the system has transmitted 0 but the true cumulative gradient is 1.6. The parameter never updates, and 100 percent of gradient information is lost.

Error-feedback outcome: In table 6.13, the accumulator e_t telescopes the residual across steps. After 5 steps, the system has transmitted 2 with a remaining error of –0.4. The true cumulative is 1.6, so transmitted + error = 2 + (–0.4) = 1.6 ✓

Systems insight:

1. **No information is lost**: The sum (transmitted + error buffer) always equals the cumulative true gradient.
2. **Small gradients accumulate**: Individual gradients of 0.3–0.4 were too small to transmit alone, but they accumulated until crossing the threshold.
3. **Error oscillates around zero**: The error buffer e_t does not grow unboundedly; it oscillates as gradients are "paid back" through transmission.
4. **Convergence preserved**: Over time, the model receives approximately the correct total gradient, with some delay.

Table 6.13: Error-feedback compression trace: Same five-step sequence under error-feedback compression.

Step	g_t	e_t	$g_t + e_t$	v_t	$e_{t+1} = (g_t + e_t) - v_t$	Cumulative v
1	0.4	0	0.4	0	0.4	0
2	0.3	0.4	0.7	1	–0.3	1
3	0.2	–0.3	–0.1	0	–0.1	1
4	0.4	–0.1	0.3	0	0.3	1
5	0.3	0.3	0.6	1	–0.4	2

6.6.4 1-bit Adam: Compression-aware optimization

Error feedback restores convergence guarantees for any compression scheme, but it treats the optimizer as a black box. The gradient is compressed, transmitted, decompressed, and then fed to the optimizer as if nothing happened. This separation leaves performance on the table: the optimizer maintains internal state (momentum, variance estimates) that contains information about the gradient distribution, yet compression ignores this state entirely. The decision is whether compression should operate on raw gradients or on the optimizer state that already summarizes them.

A more integrated approach, pioneered by Microsoft's DeepSpeed team, compresses the *optimizer's communication* rather than the *raw gradients*. **1-bit Adam** (Tang et al. 2021) observes that the Adam optimizer maintains two momentum states (m_t and v_t) that change slowly between steps. Rather than communicating the full gradient and reconstructing Adam states independently on each worker, 1-bit Adam communicates only the sign of the momentum (1 bit per parameter) plus a per-tensor scaling factor.

The algorithm proceeds in two phases. During a *warmup phase* (typically 15–20 percent of training), standard Adam runs with full-precision communication to establish stable momentum estimates.

Once the momentum variance stabilizes (indicated by the ratio $\text{Var}(m_t)/\text{E}[|m_t|]^2$ falling below a threshold), the algorithm switches to *compressed mode*. In compressed mode, each worker computes its local Adam update, extracts the sign of the momentum difference since the last communication, and transmits 1 bit per parameter plus a single FP32 scaling factor per tensor.

The theoretical justification rests on the observation that Adam's adaptive learning rates concentrate gradient information into the magnitude of the momentum terms. Once these magnitudes stabilize, the *direction* (sign) carries most of the information, while the *magnitude* can be approximated by a single scaling factor. Error feedback is applied to the sign compression to ensure no directional information is permanently lost.

The 1-bit Adam paper reports up to 5× communication-volume reduction while matching the convergence speed of uncompressed Adam, with experiments up to 256 GPUs showing up to 3.3× higher throughput for BERT-Large pretraining and up to 2.9× higher throughput for SQuAD fine-tuning (Tang et al. 2021). The practical lesson is narrower than "compression always wins": the compression overhead must remain small compared with the communication time saved.

The success of 1-bit Adam illustrates a broader principle: compression is most effective when it is co-designed with the optimization algorithm. Compressing raw gradients discards information indiscriminately. Compressing optimizer states exploits the structure of the optimization trajectory, achieving higher compression ratios with less convergence impact.

Napkin Math 6.7: The payback of compression

Problem: A training job takes 40 ms to synchronize a gradient buffer. The system implements 1-bit Adam, which reduces communication volume by 32× but adds 2 ms of CPU/GPU overhead for the compression logic. Scaling factors, optimizer-state coordination, and protocol overhead reduce the realized network gain to 8× effective throughput. What is the actual communication speedup?

Math:

1. **Compressed communication time**: 40 ms divided by 8 = 5 ms.
2. **Total new time**: 5 ms (comm) + 2 ms (overhead) = 7 ms.
3. **Effective speedup**: 40 ms divided by 7 ms ≈ 5.7×.

Systems insight: Compression is a trade of *compute for bandwidth*. It pays off only when $T_{\text{overhead}} < T_{\text{comm}}(N) \times (1 - 1/\text{Ratio})$. In this case, spending 2 ms to save 35 ms of network time is an exceptional trade. However, on a high-speed NVLink network where $T_{\text{comm}}(N)$ is only 1 ms, this same compression logic would slow down training. Always profile the network before adding compression.

The payoff calculation makes compression a conditional optimization: it must save enough communication time to cover both encoding overhead and convergence risk.

Checkpoint 6.4: Gradient compression decisions

Verify your understanding of when and how to apply gradient compression:

- ☐ **Error feedback**: Explain why 1-bit SGD without error feedback causes divergence while 1-bit Adam with warmup converges, comparing how each method handles information loss.
- ☐ **Compression payoff**: Determine whether applying 4× gradient compression improves total step time when the backward pass takes 800 ms and AllReduce takes either 200 ms or 2000 ms.
- ☐ **Error accumulator**: Derive the steady-state behavior of the error accumulator e_t when the true gradient is constant at $g = 0.3$ and the compression threshold is 0.5.

- ☐ **Top-k vs. quantization**: Choose between Top-k sparsification and quantization for a workload, accounting for compression ratio and sparse-index encoding overhead.

6.6.5 Compression trade-offs: Bandwidth vs. convergence

Gradient compression is not free; it trades reduced communication for increased variance or bias in the optimization process. The comparison in table 6.14 summarizes the source-backed patterns from quantization, sparsification, error-feedback, and optimizer-aware compression work (Seide et al. 2014; Aji and Heafield 2017; Lin et al. 2018; Stich et al. 2018; Karimireddy et al. 2019; Tang et al. 2021).

Table 6.14: Gradient Compression Methods: Compression-ratio versus convergence-impact trade-off for the four canonical gradient compressors. The right choice depends on whether bandwidth or convergence speed is the binding constraint; aggressive compressors only pay off once communication dominates iteration time.

Method	Compression Ratio	Convergence Impact	Best Use Case
FP16	2×	Negligible	Common accelerator baseline
INT8 + Error FB	4×	Minor slowdown (~5–10%)	Bandwidth-constrained clusters
Top-k (1%) + Error FB	100×	Moderate slowdown (~10–20%)	Cross-data center training
1-bit + Error FB	32×	Significant slowdown (~20–30%)	Extreme bandwidth constraints

6.6.5.1 When to use compression

The decision rule is not the compression ratio alone. Compression is worthwhile only when the wall-clock time saved per step exceeds the encoding overhead and the extra steps caused by slower convergence. Aggressive compression pays off when communication time dominates compute time (a high $T_{\text{comm}}(N)/(T_{\text{compute}}/N)$ ratio), which typically occurs with smaller models on large clusters; when compute time dominates instead, the convergence slowdown is not worth the savings because the system is not bottlenecked on communication. Independently of that ratio, any lossy compression beyond FP16 should carry error feedback unless the optimizer analysis and convergence tests justify omitting it, since without correction convergence can fail.

The α-β analysis from section 6.2 helps determine when compression pays off: if the gradients are large enough to be bandwidth-bound ($M > n^*$), compression directly reduces wall-clock time. If they are latency-bound ($M < n^*$), compression will not help because the latency term dominates regardless of message size.

The decision of whether compression is worthwhile requires comparing the communication time savings against the convergence penalty. Consider a concrete scenario: a training run requires 100,000 steps to converge without compression, with each step taking 500 ms (300 ms compute, 200 ms communication). The total training time is 50,000 seconds. Applying INT8 compression with error feedback reduces communication time by 4× (from 200 ms to 50 ms per step) but increases the required steps by 10 percent (from 100,000 to 110,000). The new total time is $110{,}000 \times 0.35 = 38{,}500$ seconds, a 23 percent improvement. The compression is worthwhile because the per-step communication savings (150 ms) outweigh the additional steps required.

Now consider the same model on a faster network where communication takes only 30 ms per step. Compression reduces this to 7.5 ms (saving 22.5 ms per step) but still adds 10 percent more steps. The new total time is $110{,}000 \times 0.3075 = 33{,}825$ seconds vs. $100{,}000 \times 0.33 = 33{,}000$ seconds without compression. The compression increases total training time by 2.5 percent, because the per-step savings (22.5 ms) are too small relative to the convergence penalty (10,000 extra steps at 307.5 ms each). This example illustrates why compression should only be applied when the communication-to-computation ratio ($T_{\text{comm}}(N)/(T_{\text{compute}}/N)$) exceeds a threshold that depends on the specific convergence penalty of the chosen method.

6.7 The Communication Library Landscape

Writing a highly optimized, topology-aware, hierarchical Ring AllReduce from scratch in C++ would take a dedicated team of engineers months of effort. The engineering decision is which library

preserves the chapter's model, topology, and overlap assumptions on the target hardware. The preceding sections developed three complementary strategies for taming communication cost; production libraries realize those strategies through four decision axes: the device memory path, topology awareness, portability, and observability.

6.7.1 NCCL: Why it is central to NVIDIA GPU workloads

NVIDIA Collective Communications Library (NCCL)[11] is central to many NVIDIA multi-GPU training deployments because it turns the collective algorithms above into GPU-resident data movement (Jeaugey 2017; NVIDIA 2026b). Its adoption stems from three GPU-specific optimizations that MPI and Gloo cannot replicate without hardware vendor support. Kernel fusion folds the reduction operator (sum, average) directly into the memory-copy kernel, so instead of copying data to a buffer, reducing, then copying the results back, NCCL reduces during the transfer and eliminates the intermediate memory traffic that would otherwise cap HBM bandwidth utilization. Channel pipelining opens multiple parallel communication channels to saturate every network interface at once, so a DGX node with 8 NICs reaches 8× the bandwidth of a single-channel implementation. GPUDirect RDMA lets the network card read directly from GPU memory over PCIe, bypassing the CPU entirely; without it, data would traverse GPU → CPU memory → NIC → network, adding microseconds of latency and consuming CPU cycles. Together these explain why NCCL is usually the first backend to benchmark for NVIDIA GPU collectives before falling back to generic MPI implementations.

Beyond raw performance, NCCL's topology auto-discovery is a critical differentiator. The topology-detection phase from section 6.5.3 is, in NCCL, driven by NVIDIA Management Library (NVML) queries and PCI bus enumeration of NVLink connectivity and NIC placement. The library-specific payoff is hardware-aware algorithm selection: on a DGX H100 with NVSwitch, NCCL recognizes that all 8 GPUs are fully connected and selects NVSwitch-based algorithms that differ from the ring algorithms used on systems with point-to-point NVLink connections.

NCCL also implements automatic algorithm selection based on message size and GPU count. Internally, it maintains tuning logic that maps message size, GPU count, and topology to a selected algorithm or protocol family. Users can override or inspect some built-in choices through documented environment variables such as `NCCL_ALGO`, `NCCL_PROTO`, and `NCCL_DEBUG` when benchmarking reveals suboptimal choices for specific workloads (NVIDIA 2026b). The debug output provides essential visibility for performance debugging.

A newer NCCL capability is limited support for user-defined reduction operators, such as the documented premultiplied-sum reduction for a communicator and datatype (NVIDIA 2026b). This support is narrower than MPI-style arbitrary reductions and custom datatypes, so application-specific aggregation such as outlier clipping usually still requires separate kernels or framework-level logic.

Despite its adoption, NCCL is not without limitations. It is open source but tightly coupled to NVIDIA GPUs and networking, and cannot be used as the native collective library on AMD or Intel accelerators. For organizations building multi-vendor infrastructure or requiring full control over the communication stack, these limitations motivate the use of alternative libraries.

6.7.2 MPI: The HPC foundation

The Message Passing Interface (MPI)[12] standardized collective operations decades before deep learning existed. The MPI Forum's standards history records Version 1.0 in May 1994, defining a vendor-neutral API for point-to-point messaging, collective operations, and process management that became the lingua franca of high-performance computing (Message Passing Interface Forum 2015, 1993). MPI remains relevant for ML systems when portability, CPU-side coordination, or HPC integration is the binding requirement.

For CPU-based distributed computing, MPI implementations (OpenMPI, MPICH, Intel MPI) are mature and well-optimized, with decades of performance tuning across diverse network fabrics. HPC facilities that predate the GPU training era often have MPI deeply integrated into their job schedulers and resource managers, making MPI the path of least resistance for deploying ML workloads on these systems.

11 **NCCL (NVIDIA Collective Communications Library)**: NCCL can approach theoretical peak bandwidth on well-configured NVIDIA clusters by combining GPU-resident collectives, GPUDirect RDMA, and topology-aware path selection (Jeaugey 2017; NVIDIA 2026b). Its NVIDIA-specific design creates a vendor lock-in trade-off: organizations can gain a tuned collective backend for NVIDIA GPU fabrics but lose portability to AMD (RCCL) or Intel (oneCCL) hardware.

12 **MPI (Message Passing Interface)**: Standardized in June 1994 after a three-year community effort, MPI defined the collective operations (AllReduce, Broadcast, Scatter) that ML frameworks later adopted wholesale. For GPU training, standard MPI requires explicit device-to-host copies that negate GPUDirect benefits, which is why NCCL displaced it for GPU collectives while MPI persists for job launch (`mpirun`) and CPU-side coordination.

MPI's specification includes features that NCCL lacks, most notably **persistent collectives** and **non-blocking collective operations** with fine-grained completion semantics. Persistent collectives (introduced in MPI 4.0) allow applications to "preregister" a collective operation, amortizing the setup overhead across thousands of invocations. This is valuable for training loops where the same AllReduce shape repeats at every step. Non-blocking collectives (`MPI_Iallreduce`) provide explicit request handles that can be tested for completion, enabling more flexible overlap patterns than NCCL's asynchronous operations.

MPI also provides **one-sided communication** (MPI_Put, MPI_Get, MPI_Accumulate) that maps naturally to RDMA hardware. These operations allow a process to read from or write to another process's memory without the remote process explicitly participating, enabling communication patterns that are difficult to express with collective operations alone. Parameter server architectures and certain asynchronous training algorithms benefit from one-sided semantics.

The primary limitation for ML practitioners is GPU-awareness. Standard MPI implementations assume host memory buffers. Calling `MPI_Allreduce` on GPU memory typically requires explicit device-to-host copies, which negate the performance advantage of keeping data on the GPU. CUDA-aware MPI extensions (available in OpenMPI and MVAPICH2) can accept GPU pointers directly, but their internal implementations rarely match NCCL's kernel fusion and channel pipelining optimizations. The practical guidance is hardware-specific: use NCCL for NVIDIA GPU-to-GPU collective operations when the hardware stack supports it, and use MPI for CPU collective operations, process management (`mpirun`, `mpiexec`), or cross-platform portability across non-NVIDIA hardware.

6.7.3 Gloo: Cross-platform flexibility

Gloo is Meta's open-source collective communication library, integrated into PyTorch as a backend option alongside NCCL. While NCCL is often the primary choice for performance-critical NVIDIA GPU training, Gloo fills a complementary role in the PyTorch ecosystem: it preserves distributed semantics when portability matters more than saturating GPU interconnects.

Gloo's primary strength is its portability. It supports Linux, macOS, and Windows without requiring CUDA or any vendor-specific runtime. This makes Gloo the natural choice for development and debugging workflows where engineers prototype distributed training logic on laptops or CI servers before deploying to GPU clusters. Gloo's TCP/IP transport works over any network stack, including the loopback interface for single-machine multi-process testing, eliminating the infrastructure requirements that NCCL imposes.

For CPU-only training (data preprocessing pipelines, feature engineering, CPU-based model architectures), Gloo's implementations are competitive with MPI for small-to-medium cluster sizes. Gloo optimizes for the common case in PyTorch distributed training: process groups that perform AllReduce and Broadcast on tensors stored in CPU memory. Its shared-memory transport enables high-bandwidth intra-node communication without network overhead, achieving near-memcpy throughput for local process groups.

Gloo also serves as a fallback backend in PyTorch's `torch.distributed` module. When NCCL is unavailable or inappropriate (non-NVIDIA hardware, missing drivers, unsupported platforms), PyTorch configurations can use Gloo for collective operations. This fallback behavior is valuable for mixed-vendor environments and for ensuring that distributed training code remains portable across hardware configurations.

The primary limitation is performance on GPU clusters. Gloo lacks kernel fusion, GPUDirect RDMA, and NVLink-aware routing, so GPU tensor collectives require explicit device-to-host copies. On NVIDIA GPU clusters, Gloo can achieve far less bandwidth than NCCL for large-message AllReduce operations. For latency-sensitive small-message operations, the gap widens further because Gloo cannot bypass the OS kernel for GPU memory access. The guidance for practitioners is straightforward: use Gloo for development, CPU workloads, and portability; switch to a hardware-optimized backend for performance-critical GPU training.

6.7.4 Library selection guide

The selection matrix in table 6.15 summarizes those axes. The point is not to memorize library names; it is to choose the backend whose memory path, topology model, portability constraints, and debugging visibility match the job.

Table 6.15: Communication Library Selection: Choose based on hardware and workload requirements. Many NVIDIA GPU training deployments use NCCL; Gloo serves as a portable fallback.

Scenario	Recommended Library	Rationale
Multi-GPU training (NVIDIA)	NCCL	GPUDirect, kernel fusion, NVLink-aware
CPU-only distributed training	Gloo or MPI	Mature CPU optimizations
Development/debugging	Gloo	Cross-platform, no CUDA dependency
Mixed vendor GPUs	Gloo (fallback)	NCCL is NVIDIA-specific
HPC integration	MPI + NCCL	MPI for job launch, NCCL for GPU collectives

The table follows directly from the chapter's cost model. A backend is not faster in the abstract; it is faster when its memory path avoids host copies, its topology model matches the fabric, and its failure signals expose the part of α, β, or processor overhead that limits the job.

6.7.5 Vendor-specific libraries

The three libraries above do not exhaust the communication landscape because the same collective algorithm has different effective α, β, and topology constraints on different accelerator platforms. The pattern is consistent: use the backend that understands the device memory path and interconnect on the hardware actually running the job. AMD's **RCCL (ROCm Communication Collectives Library)** mirrors NCCL's API for AMD GPUs and optimizes collectives for the ROCm stack and Infinity Fabric, but benchmarked RCCL deployments may achieve only 70–85 percent of comparable NCCL bandwidth when intra-node fabric maturity or software tuning lags the NVIDIA stack. Intel's **oneCCL (oneAPI Collective Communications Library)** targets Intel accelerator and CPU deployments with its own topology-aware collectives.

The broader pattern is a standardized front-end API with hardware-specific backends underneath it. PyTorch's `torch.distributed` module routes collective calls to NCCL, RCCL, oneCCL, or Gloo based on the configured process group and available backend. In practice, teams often mix backends rather than choosing one globally: a GPU process group may use NCCL for gradient AllReduce while a CPU-side coordination group uses Gloo for barriers, scalar broadcasts, or sampler state. The chapter's algorithms remain universal, but the efficient execution plan is hardware-specific. When that plan underperforms, the communication backend also becomes the diagnostic starting point.

Systems Perspective 6.2: Debugging communication bottlenecks

When a distributed training job runs slower than expected, the communication library provides the first diagnostic signals because it exposes the selected algorithm, measured bandwidth, rank mapping, and overlap behavior. A systematic workflow isolates whether the bottleneck is in computation, communication, or their interaction:

1. **Profile with NCCL debug logging**: Set `NCCL_DEBUG=INFO` to see which algorithm (Ring, Tree) and protocol (Simple for bulk bandwidth, LL/LL128 as lower-latency protocols for small messages) NCCL selects for each collective. Unexpected algorithm choices often indicate topology mis-detection.
2. **Measure bare collective performance**: Run NCCL's built-in benchmarks (`nccl-tests`) with the cluster's exact topology to establish the achievable bandwidth baseline. If `nccl-tests` achieves 90 percent of theoretical bandwidth but the training job achieves only 50 percent, the bottleneck is in how the training framework invokes collectives, not in the communication library itself.
3. **Check for stragglers**: Use `torch.cuda.synchronize()` before and after each collective to measure per-operation time. A collective that takes $2\times$ longer than expected often indicates one GPU is delayed (thermal throttling, ECC error recovery, or unbalanced data loading), which stalls the entire barrier.
4. **Verify overlap effectiveness**: Use NVIDIA Nsight Systems to visualize the timeline of compute kernels and communication operations on the same axis. Effective overlap

shows communication operations running in parallel with backward pass kernels. Poor overlap shows gaps where the GPU is idle during communication.

5. **Isolate network vs. host overhead**: If `nccl-tests` shows low bandwidth, the issue is network-level (bad cables, congestion, misconfigured routing). If `nccl-tests` shows full bandwidth but training is slow, the issue is host-level (insufficient overlap, small bucket sizes, CPU bottlenecks in data loading).

Once the backend is correctly matched to the hardware and measured against a bare collective baseline, the remaining exposed communication time must be hidden behind computation.

6.8 Communication-Computation Overlap

A 20-millisecond network delay can effectively disappear—not by upgrading the physical network, but by hiding the communication behind arithmetic. The preceding sections reduced communication cost through algorithm choice, topology awareness, and payload compression. Overlap attacks the residual by changing when communication happens.

6.8.1 Layer-by-layer overlap

Gradient computation during the backward pass proceeds layer by layer, from the output layer back to the input layer. The gradient for layer ℓ is available as soon as that layer's backward pass completes, even while layers $\ell-1, \ell-2, \ldots$ are still computing. This creates an opportunity: begin communicating the gradient for layer ℓ immediately, while the GPU computes the gradient for layer $\ell-1$.

In PyTorch's `DistributedDataParallel` (DDP), this overlap is implemented through **gradient hooks**. Each parameter registers a hook that fires when its gradient is ready. The hook triggers an asynchronous AllReduce for that parameter's gradient bucket. Meanwhile, the backward pass continues computing gradients for earlier layers. If the backward computation for the remaining layers takes longer than the AllReduce (the common case for large models), the communication is completely hidden.

The effectiveness of this overlap depends on the relative sizes of computation and communication at each layer. For transformer models, the largest gradient tensors belong to the attention and feed-forward weight matrices, which are also the most computationally expensive layers. This favorable correlation means that the layers with the most data to communicate are also the layers with the most computation behind which to hide that communication.

The overlap strategy interacts with the algorithm choice from the previous sections. Ring AllReduce, with its higher latency but optimal bandwidth, benefits more from overlap than Tree AllReduce, because Ring's latency penalty (which would otherwise dominate for medium-sized messages) can be hidden behind computation. This is one reason why NCCL may select Ring even below the theoretical crossover point when it detects that the framework supports asynchronous operation: the latency disadvantage of Ring is neutralized by overlap, while its bandwidth advantage remains.

A critical synchronization barrier limits this overlap: the optimizer step *cannot* begin until all gradients are reduced. The gradients for the final layers processed (closest to the input) expose their communication latency because no subsequent backward computation remains to mask them.

6.8.2 Bucket fusion

The layer-by-layer overlap described earlier assumes that each layer's gradient is communicated as a single message. In reality, a transformer layer contains dozens of individual parameter tensors (query, key, value projection matrices, output projection, layer norm parameters, feed-forward weights, and biases). Launching a separate AllReduce for every individual parameter would create thousands of small-message collectives, each paying the full α overhead. To avoid this, DDP uses **bucket fusion** to group parameters into buckets (often configured around tens of megabytes, with 25 MB a common PyTorch reference point) and launches one AllReduce per bucket. The bucket size represents a trade-off: larger buckets amortize α overhead but delay the start of communication (because the bucket cannot be sent until all its parameters have computed gradients). Smaller buckets start communication earlier but pay more α overhead.

The optimal bucket size depends on the model architecture and network characteristics. For models with many small layers (such as deep residual networks), smaller buckets (5–10 MB) improve overlap by starting communication sooner. For models with a few large layers (such as large language models with multi-gigabyte embedding tables), larger buckets (50–100 MB) are preferable because the large layers dominate both computation and communication time. PyTorch's `bucket_cap_mb` parameter in DDP allows tuning this trade-off.

The order in which parameters are added to buckets determines overlap effectiveness. DDP assigns parameters to buckets in reverse order of their use in the forward pass, which corresponds to the order in which gradients become available during the backward pass. This reverse-order assignment ensures that the first bucket to fill (and thus the first AllReduce to launch) contains the parameters from the last layers of the forward pass, which are the first layers of the backward pass. This ordering maximizes the overlap window: the earliest buckets have the most subsequent computation behind which to hide their communication.

6.8.3 Overlap limits: When hiding fails

Communication-computation overlap has fundamental limits. The LogP model's overhead parameter o represents the irreducible GPU time consumed by initiating and completing transfers. Even with perfect overlap, the GPU must spend at least $2o$ per AllReduce operation on initiation and completion. If the model has many layers but each layer's backward pass is short (less than o), the GPU spends more time on communication overhead than on computation, and overlap provides no benefit.

Additionally, the first and last layers of the backward pass cannot overlap. The first layer to complete (the output layer) has no prior communication to overlap with. The last layer to communicate (the input layer) has no subsequent computation to overlap with. For shallow models (2–3 layers), these boundary effects consume a significant fraction of the total time, limiting the achievable overlap to 50–60 percent. For deep models (dozens of layers or more), the boundary effects are negligible, and overlap can hide 90–95 percent of communication time.

A third limitation arises from memory pressure. Overlapping communication with computation requires maintaining additional GPU memory buffers: the gradient being communicated must remain in memory while the backward pass continues producing new gradients for earlier layers. For FSDP workloads, where memory optimization is the primary motivation, this additional buffer requirement can conflict with the memory savings that FSDP provides. The engineering challenge is to find the sweet spot where overlap is aggressive enough to hide communication but not so aggressive that it pushes the GPU into out-of-memory territory. PyTorch FSDP settings such as `limit_all_gathers`, `backward_prefetch`, `forward_prefetch`, and resharding or prefetch choices provide knobs for this trade-off.

A realistic training configuration shows how much communication remains exposed after overlap.



Napkin Math 6.8: Overlap budget for a 7B transformer

Problem: A 32-layer transformer model (7B parameters) is trained on 64 GPUs. Each layer's backward pass takes 15 ms. For this overlap budget, the communication bucket uses conservative FP32-gradient accounting, so the hierarchical AllReduce for each layer's gradients (~880 MB per layer) takes 26.4 ms using 100 MB buckets. What is the step time with and without overlap?

Without overlap (sequential):

$T_{\text{sequential}} = T_{\text{backward}} + T_{\text{comm}}(N)$ = 480 ms + 32 × 26.4 ms = 1324.8 ms.

With overlap (pipelined):

Each layer's AllReduce (26.4 ms) runs in parallel with the next layer's backward pass (15 ms). Since 26.4 ms > 15 ms, there is 11.4 ms of exposed communication per layer that cannot be hidden.

$T_{\text{pipelined}} = T_{\text{backward}} + T_{\text{first layer comm}} + (N_L - 1) \times T_{\text{exposed}}$ = 859.8 ms.

Systems insight: Overlap reduces total step time by 35.1 percent. The remaining exposed communication comes from the AllReduce being slower than the per-layer backward pass. To eliminate this residual, either increase the backward pass computation (larger batch size) or reduce AllReduce time (more aggressive compression, better topology). The overlap is most effective when $T_{\text{backward per layer}} > T_{\text{AllReduce per layer}}$.

The combination of hierarchical AllReduce (reducing the volume of data to communicate), rail-optimized routing (maximizing the throughput of each communication operation), and layer-by-layer overlap (hiding the remaining communication behind computation) forms a common communication optimization stack for distributed training. Each technique addresses a different term in the performance equation: hierarchical algorithms reduce M, topology-aware routing maximizes effective β, and overlap amortizes α by running communication concurrently with computation. When all three are applied together, the communication overhead for a well-configured system can fall to 5–15 percent of total step time, down from the 50–80 percent that a naive flat, synchronous approach would impose.

Even the intra-node base case shows this stack succeeding before any of the multi-node machinery is needed: a 70B-parameter model on a single node of 8 NVLink-connected GPUs must AllReduce 140 GB of FP16 gradients per step, which Ring AllReduce on NVLink completes in roughly 0.3 seconds against a 2.1-second per-step compute budget, comfortably overlapped at about 15 percent of compute time. That quantitative success sets up the pitfalls below, because each one begins by optimizing the wrong term, choosing the wrong primitive, or trusting a topology assumption that no longer matches the workload.

6.9 Fallacies and Pitfalls

Communication optimization attracts persistent misconceptions because the interaction between latency, bandwidth, topology, and compression creates nonobvious failure modes that simple mental models miss.

Fallacy: *Bandwidth is the only metric that matters.*

For small messages (pipeline parallelism activations, MoE tokens), latency (α) dominates. Buying 400G networking will not help if the message takes 5 µs to serialize in software. The critical message size $n^* = \alpha \cdot \beta$ determines which metric to optimize: below n^*, reduce latency; above it, increase bandwidth.

Pitfall: *Assuming AllReduce works for everything.*

AllReduce creates a global barrier and assumes all participants contribute identical data shapes. In Expert Parallelism (MoE) and Recommendation Systems, where each worker needs to send distinct data to every other worker, AllReduce is fundamentally wrong. These workloads require AllToAll, which has $\mathcal{O}(N^2)$ logical connections and hits network contention limits at much smaller cluster sizes.

Fallacy: *Pipeline parallelism eliminates communication overhead.*

Pipeline parallelism replaces AllReduce with point-to-point activation transfers between adjacent stages, and the per-message cost is genuinely cheaper. Pipeline introduces a different overhead: bubble idleness. With p pipeline stages, at least $(p-1)/(p-1+m)$ of GPU-time is idle while the pipeline fills and drains, where m is the number of microbatches. For deep pipelines ($p \geq 8$), this requires $m \geq 32$ microbatches to keep the bubble below 20 percent, which constrains batch size, activation memory, and convergence. The choice between data parallelism and pipeline parallelism is not "communication vs. no communication" but a trade between bandwidth-bound AllReduce and time-bound bubble overhead.

Pitfall: *Using Ring AllReduce by default without checking message size and topology.*

Ring achieves bandwidth-optimal $2\frac{N-1}{N}\frac{M}{\beta}$ but pays $\mathcal{O}(N)$ latency. For a 1 MB gradient across 64 GPUs with $\alpha = 10$ μs, Tree AllReduce wins because Ring's 1,260 µs latency penalty exceeds Tree's 1,000 µs bandwidth penalty. The log-aware crossover estimate $M_{\text{crossover}} \approx N\alpha\beta/\log_2 N$ determines when to switch algorithms; $N\alpha\beta$ is only a coarse upper-scale heuristic.

Fallacy: *Gradient compression is safe without error feedback.*

Top-k sparsification can achieve 99 percent compression, but naively discarding small gradients causes divergence. Without error feedback ($e_{t+1} = (g_t + e_t) - v_t$), gradients below the threshold are lost forever, accumulating systematic bias that eventually destabilizes training.

Pitfall: *Evaluating gradient compression by communication speedup alone.*

A compression scheme that achieves 100× communication speedup is worthless if it degrades model quality enough to require 2× more training iterations to reach the target accuracy. The right metric is *time-to-accuracy*, not *time-per-iteration*. Communication microbenchmarks measure time-per-iteration, while the engineering decision depends on time-to-accuracy, and the two diverge whenever compression introduces gradient bias or variance that slows convergence. Validate compression on the target convergence benchmark with the deployment training schedule, not on a synthetic AllReduce microbenchmark alone.

Fallacy: *Async collectives always hide latency.*

Python's `dist.all_reduce(..., async_op=True)` only returns control to the CPU. The LogP model distinguishes network latency L_{lat} (overlappable) from processor overhead o (nonoverlappable). If the GPU compute kernel is shorter than the communication overhead, the GPU still stalls. Communication can only be hidden when $T_{\text{compute}} > o$.

Pitfall: *Silent data corruption in the network.*

Networks are not perfect. A bad cable, faulty NIC, or firmware bug can corrupt data below the level where the training framework sees an explicit error. At 10,000 nodes running 24/7, even very rare bit errors can appear often enough to matter. The communication-specific lesson is that bandwidth tuning is incomplete without end-to-end validation: checksums, collective result checks, gradient-norm sanity tests, and fault-tolerance mechanisms must catch corrupted gradients before they become model updates.

Fallacy: *Flat AllReduce is good enough for multi-node training.*

A flat Ring AllReduce treats all links as equal, routing data across 50 GB/s InfiniBand when 900 GB/s NVLink is available within the node. For an 8-node cluster with 8 GPUs per node, hierarchical AllReduce reduces inter-node traffic by 8× compared to flat Ring, achieving about 4.8× in the worked example above. Ignoring the bandwidth hierarchy leaves the majority of intra-node bandwidth unused while overloading the scarce inter-node bandwidth.

Pitfall: *Using the ring AllReduce formula for intra-node communication.*

The ring AllReduce cost model assumes a logical ring topology, which matches the inter-node communication pattern across InfiniBand. It does not match intra-node communication. Within a DGX-class node, GPUs talk through NVLink and NVSwitch, which provides all-to-all connectivity rather than a ring. The actual intra-node AllReduce often uses a single-step reduce through the NVSwitch crossbar, with latency closer to a single hop than to the $\mathcal{O}(N)$ ring cost. Capacity planning that applies the ring formula uniformly overstates intra-node communication time by an order of magnitude and pushes the parallelism boundary outward unnecessarily. Use the crossbar model inside the node, the ring or tree model across nodes, and the hierarchical sum for global AllReduce.

Fallacy: *Rank-to-GPU mapping does not affect collective performance.*

If the job scheduler assigns ranks to GPUs without considering the physical topology, hierarchical AllReduce and rail-optimized routing may route traffic suboptimally. A common symptom is that `nccl-tests` achieves full bandwidth on the cluster, but the actual training job achieves only 50–60 percent because ranks are assigned across nodes in a way that prevents rail alignment. Verify that rank assignment matches the expected topology (e.g., ranks 0–7 on Node 0, ranks 8–15 on Node 1) before benchmarking communication performance.

Pitfall: *Benchmarking collectives without recording placement context.*

A collective benchmark is not reusable evidence unless it records rank order, NIC rail binding, NCCL topology configuration, and node allocation. Without that context, teams compare results from different physical layouts and mistake placement differences for library or hardware regressions.

6.10 Summary

Communication is the friction of scale. Computation is local, but learning is global, and the α-β and LogP models transform communication bottleneck analysis from guesswork into quantitative engineering. The critical message size $n^* = \alpha \cdot \beta$ separates latency-bound from bandwidth-bound regimes, and the gap between theoretical predictions and measured NCCL performance reveals

that software stack overhead inflates small-message latency by 5–10×, shifting algorithm crossover points in practice.

The Collective Primitives (AllReduce, AllGather/ReduceScatter, AllToAll) map directly to parallelism strategies, and each primitive determines a distinct scaling ceiling. AllReduce workloads are bandwidth-bound and scale gracefully, while AllToAll workloads are contention-bound and hit practical limits at smaller cluster sizes.

The Ring and Tree AllReduce algorithms occupy opposite ends of the latency-bandwidth trade-off, with the log-aware crossover estimate $M_{\text{crossover}} \approx N\alpha\beta/\log_2 N$ determining which is optimal for a given configuration. Hierarchical AllReduce and Rail-Optimized Routing exploit the bandwidth hierarchy of GPU clusters to multiply effective inter-node bandwidth, while in-network reduction (SHARP) reduces host-visible communication by performing part of the aggregation inside network switches.

When even the fastest wires are not enough, gradient compression provides the final squeeze. Quantization, sparsification, and compression-aware optimizers like 1-bit Adam reduce communication volume by 4–1000×. Error feedback ensures that while the journey may be compressed or delayed, no vital information is permanently lost, preserving the mathematical truth of the model's convergence. Finally, communication-computation overlap through layer-by-layer pipelining and bucket fusion hides the remaining communication time behind useful computation, reducing the effective communication overhead to 5–15 percent of total step time in well-configured systems.

The communication traffic patterns differ fundamentally across the Lighthouse Archetypes.



Lighthouse 6.1: Communication archetype patterns

The "Travel Manifest" for a gradient depends on the system's objective function and constraint regime. Each lighthouse archetype faces a distinct communication challenge, and the techniques developed in this chapter map to those challenges differently, as table 6.16 summarizes.

Table 6.16: Archetype-to-collective mapping: Each lighthouse archetype's communication pattern is determined by its dominant bottleneck.

Archetype	Primary Collective	Dominant Friction	Optimization Strategy
Archetype A (GPT-4/Llama-3)	AllReduce	Bandwidth (β)	Hierarchical AllReduce; Rail-optimization
Archetype B (DLRM at Scale)	AllToAll	Latency (α) & Contention	Topology-aware routing; token load-balancing
Archetype C (Federated MobileNet)	P2P/Async	Connectivity & Latency	Aggressive quantization; Error Feedback

The key insight is that the archetypes face different communication bottlenecks even when they all train neural networks. Archetype A (GPT-4/Llama-3) is bandwidth-bound and benefits from hierarchical AllReduce combined with communication-computation overlap. Archetype B (DLRM at Scale) is latency-bound and benefits from low-latency switches and topology-aware AllToAll routing. Archetype C (Federated MobileNet) is constrained by intermittent connectivity and small devices, so aggressive quantization and error feedback matter more than data-center topology. Applying the wrong optimization to the wrong archetype wastes engineering effort without improving performance.

The compression and error-feedback mechanisms developed in this chapter also matter outside data centers: for Archetype C and similar intermittent or bandwidth-poor settings, the dominant cost may be the ability to send any useful update at all rather than saturating a high-speed fabric.

The communication patterns established in this chapter reveal that distributed training is fundamentally a network engineering problem disguised as a machine learning problem. The α-β model provides the analytical framework for reasoning about this problem: every communication design decision, from algorithm selection to compression to overlap, can be evaluated in terms

of its effect on the latency and bandwidth terms. Understanding these models and techniques transforms practitioners from users of distributed frameworks into engineers who can diagnose bottlenecks, optimize topology configurations, and achieve the scaling efficiency that determines whether trillion-parameter training runs complete in weeks or months.

Key Takeaways: Every byte has a travel cost

- **The α-β model reveals the bottleneck**: The critical message size $n^* = \alpha \cdot \beta$ determines whether to optimize software latency (small messages) or hardware bandwidth (large payloads). In practice, NCCL's effective α is 5–10$\times$ higher than wire-level latency.
- **Algorithm choice is scale-dependent**: Ring AllReduce is bandwidth-optimal but pays $\mathcal{O}(N)$ latency; Tree AllReduce is latency-optimal ($\mathcal{O}(\log N)$) but bandwidth-inefficient. Use the log-aware crossover estimate $M_{\text{crossover}} \approx N\alpha\beta / \log_2 N$ to reason about the switch point, and treat $N\alpha\beta$ only as a coarse upper-scale heuristic.
- **Hierarchical algorithms multiply bandwidth**: By performing local reductions over fast NVLink before crossing the slow InfiniBand bridge, hierarchical collectives effectively multiply inter-node bandwidth by the number of GPUs per node.
- **AllToAll is the contention king**: Unlike AllReduce, AllToAll creates $\mathcal{O}(N^2)$ logical connections. This makes Expert Parallelism (MoE) and Recommendation Systems fundamentally harder to scale than dense LLMs.
- **Error feedback makes lossy compression safe**: Sparsification can discard 99 percent of gradients, provided the "error" is accumulated locally and added to the next step. This turns biased estimators into unbiased ones over time.
- **Overlap is the final multiplier**: Communication-computation pipelining through gradient hooks and bucket fusion can hide 90–95 percent of communication time behind backward pass computation for deep models.
- **Topology discovery is not optional**: High-performance libraries like NCCL dynamically map logical rings to physical wires to avoid "hot spots" and maximize bisection bandwidth. Misaligned rank-to-GPU mapping can degrade performance by 2–4$\times$.

If parallelism decides what must be communicated, this chapter is about what that communication actually costs. Every collective (AllReduce, AllGather, AllToAll) is an instruction in the fleet's real instruction set, and its price is set by the α-β model: a fixed latency to begin and a per-byte bandwidth to finish. The engineering is in lowering both, choosing a ring where bandwidth binds and a tree where latency does, collapsing the slow inter-node hop into a fast local one, and above all overlapping the transfer with computation so the fleet pays for it in time it would have spent anyway. Communication is the penalty compute pays to act as one machine, and this chapter is how that penalty is kept small enough to be worth paying.

What's Next: From logic to resilience

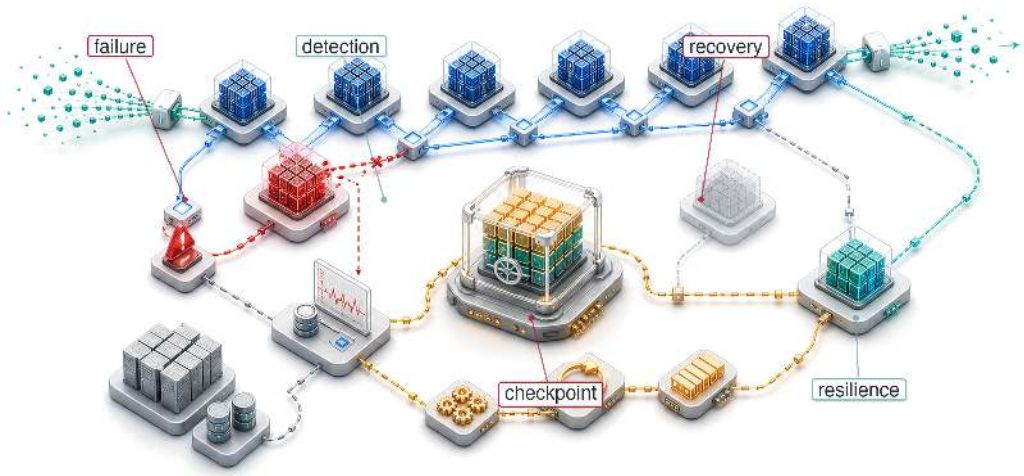
The fleet now has its *logic* (Parallelism) and its *traffic patterns* (Communication). The map is drawn and the vehicles are built. In production, however, hardware fails as a statistical certainty: GPUs overheat, networks drop packets, and nodes fail mid-calculation. In fault tolerance (Chapter 7), we examine how to maintain the illusion of a perfect supercomputer on imperfect hardware. We move from the logic of movement to the mechanics of survival, exploring checkpointing, recovery, and elastic training.

Research Questions: For further inquiry

- How should measured α-β or LogP parameters determine the choice among ring, tree, butterfly, double-tree, and hierarchical collectives?
- How do data, tensor, pipeline, and expert parallelism impose different collective patterns and scaling ceilings?
- When does topology-aware communication outperform a theoretically optimal flat algorithm?
- When is gradient compression justified, and what convergence evidence is sufficient to trust it?

7

Fault Tolerance

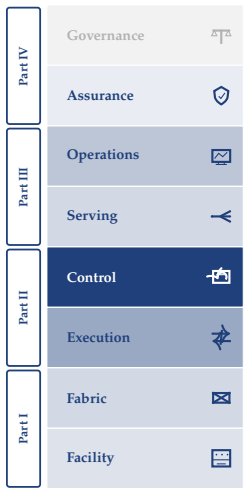


7.1 Failure Analysis at Scale
7.2 Hardware Fault Taxonomy
7.3 Software Faults
7.4 Check-and-Verify: Defending Against Silent Data Corruption
7.5 Fault Injection Tools and Frameworks
7.6 Checkpointing: Preserving Progress
7.7 Failure Detection and Recovery
7.8 Elastic Recovery
7.9 Serving Fault Tolerance
7.10 Graceful Degradation
7.11 Distributed Debugging and Observability
7.12 Case Studies
7.13 Fallacies and Pitfalls
7.14 Summary

Purpose

Why does scale transform hardware failure from rare exception to routine condition that systems must absorb continuously?

A single accelerator can run for years between hardware failures. A thousand accelerators turn that same component risk into failures every couple of days. A ten-thousand-device cluster sees failures every few hours once host, power, and network domains are included. This arithmetic is inescapable: individual component reliability does not change, but aggregate system reliability degrades multiplicatively as components are added. At large fleet scale, the question is not whether failures occur during a training run but how many, and systems that cannot absorb failures without losing progress cannot operate at all. The same logic applies to serving: a globally distributed inference system experiences regional outages, network partitions, and capacity fluctuations as continuous background conditions rather than exceptional events. Fault tolerance at scale is not about preventing failures, because prevention is impossible. It is about designing systems where failures are expected, detected, isolated, and recovered from automatically, allowing useful work to continue despite the constant churn of components entering and leaving operational status. In C^3 terms, fault tolerance spends compute and communication on coordination: preserving state and recovering work because at fleet scale component failure is a statistical certainty, not an anomaly.



Learning Objectives

- Calculate fleet-level MTBF from component failure rates and estimate failure frequency for large training clusters
- Classify hardware, software, and silent-data-corruption failures by detectability, blast radius, and recovery path
- Derive checkpoint intervals that balance write overhead, lost work, and cluster failure rates
- Design distributed checkpointing and elastic recovery plans for multi-terabyte model state across GPU fleets
- Evaluate fault injection and observability evidence to validate detection, isolation, and recovery behavior
- Implement serving redundancy, failover, and state replication under millisecond latency and partial-failure constraints
- Select graceful degradation strategies that preserve useful service when models, features, regions, or capacity fail

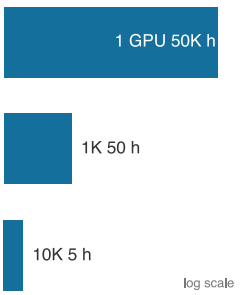
7.1 Failure Analysis at Scale

Imagine a 10,000-GPU cluster midway through a three-month training run for a new foundation model. The communication layer has done its job: thousands of devices exchange gradients through AllReduce, AllGather, and AllToAll as if they were one machine. Then the arithmetic of scale catches up. A GPU fails every few hours, and if the system cannot absorb that ordinary physical event, synchronized training halts and millions of dollars of compute sit idle. **Fault tolerance** is the system property that lets distributed execution continue making useful progress when components fail, degrade, or disappear. In the fleet stack shown in figure 1.13, fault tolerance acts as the resilience layer for distributed execution. The challenges span the full failure spectrum, from transient bit flips through intermittent aging-related errors to permanent component failures, as figure 7.1 illustrates. The per-category failure rates and the mean time between failures (MTBF) scaling, $\text{MTBF}_{\text{system}} = \text{MTBF}_{\text{component}}/N$, annotated in that figure are previews; later analysis derives the inverse scaling and tabulates the cluster rates. Gray failures and silent data corruption (SDC) add a second axis of difficulty: low detection rate, high blast radius.

That fragility is the direct consequence of successful synchronization. Distributed training systems achieve massive throughput by coordinating thousands of devices, and collective communication keeps that coordination synchronized through rigid exchanges. The same synchronization means one stalled device can stall the fleet. Fault-tolerant ML systems preserve progress by detecting failure, checkpointing recoverable state, restarting or elastically reshaping jobs, and treating hardware churn as normal execution rather than an exceptional path.

The transition from small-scale experimentation to large-scale production changes the relationship between systems and failures. A researcher training a model on a single GPU can go years without a hardware failure. That same workload on a 1,000-GPU cluster sees GPU-only failures every couple of days, and a production cluster fails more often once PCIe, power, storage, and network domains enter the failure budget. This shift from *rare exception* to *routine occurrence* demands different engineering approaches. The mathematical analysis that follows makes this transition precise and quantitative.

Because failures *cannot* be eliminated at this scale, the engineering challenge is not to prevent them but to keep making progress despite them: fault-tolerant systems verify completion rather than assume it, treat failure as a normal code path, exercise recovery continuously rather than occasionally, and account for partial failures that leave the system in states naive error handling never anticipates. The techniques that achieve this draw on decades of distributed systems research, but ML workloads change the economics. Training exhibits properties that enable fault tolerance strategies unavailable to general distributed systems: the mathematical properties of stochastic gradient descent tolerate certain errors that would corrupt other computations, checkpoint sizes are large but predictable, and recovery targets need only be *approximate* rather than *exact*. Exploiting



System MTBF collapses as the fleet grows: 50,000 h to 5 h.

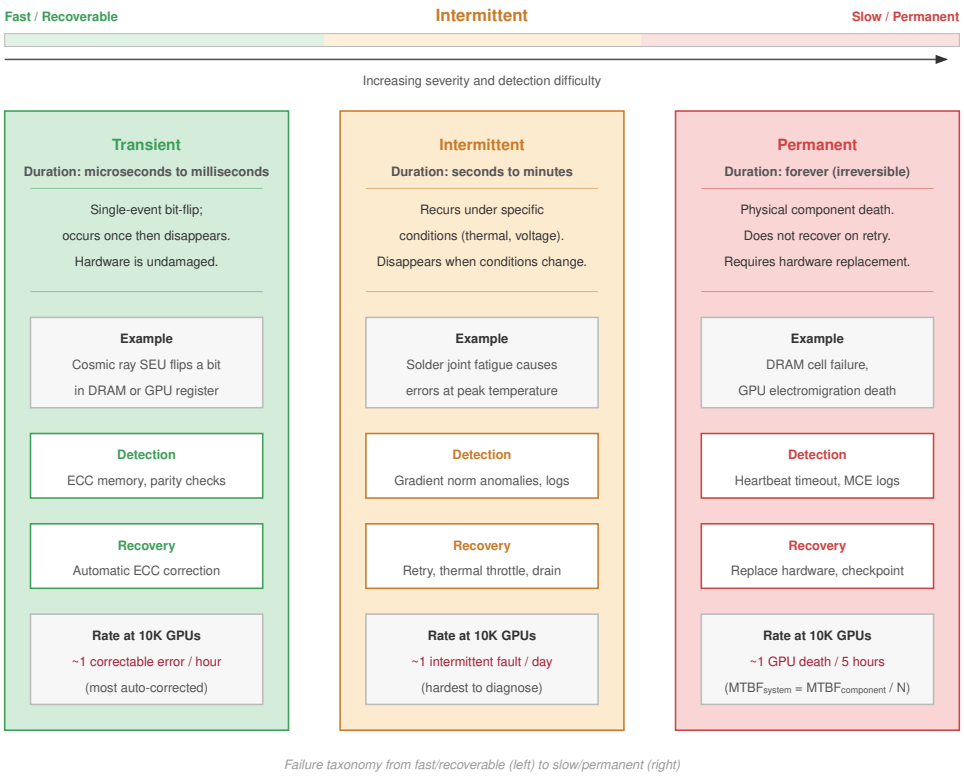


Figure 7.1: The ML Failure Spectrum: Reliability challenges at scale span a temporal taxonomy from transient errors (bit flips, single-event upsets) through intermittent errors (aging, marginal components) to permanent errors (stuck-at faults, dead components). Each category demands different detection latency and recovery strategy, and fault tolerance engineering must address all three to maintain fleet-scale throughput.

these properties makes ML-specific fault tolerance cheaper than the general-purpose approaches it descends from.

7.1.1 The mathematics of inevitable failure

System reliability engineering provides the foundational framework for understanding failure at scale (Birolini 2017). Individual components exhibit failure rates characterized by the failure rate parameter λ,[1] measured in failures per unit time. For a single component with constant failure rate λ, the probability of surviving without failure until time t follows an exponential distribution,[2] as in equation 7.1:

$$R_{\text{single}}(t) = e^{-\lambda t} \tag{7.1}$$

MTBF for this component equals $1/\lambda$. By the book's convention, mean time to failure (MTTF) names a single component's expected time to its first failure while MTBF names the repairable system's expected time between successive failures; under the constant-rate exponential model used here the two coincide numerically, and the fault-tolerance model uses MTTF for per-component constants and MTBF for composed system rates. Data center GPUs often use planning MTBF values in the tens of thousands of hours, with field behavior depending on operating conditions, cooling effectiveness, manufacturing variation, and workload stress.[3] section E.1.1 catalogs the canonical per-component MTTF values (H100, A100, TPU, PCIe, power, network) that anchor these λ and FIT figures, so a reader can substitute the constant for any device and reproduce the rates used throughout the fault-tolerance analysis.

[1] **Failure Rate (λ)**: Expressed in FITs (Failures In Time), where 1 FIT equals one failure per billion device-hours. A data center GPU with 50000 hours MTBF has FIT $= 20{,}000$, corresponding to $\lambda_{\text{hour}} = 2.0 \times 10^{-5}$ failures/hour. That seems negligible for one device but becomes dominant at fleet scale: a cluster with 10,000 GPUs accumulates 200,000,000 FITs from GPUs alone, translating to an expected GPU failure every 5 hours.

When multiple independent components operate in a system where any single failure causes system failure, equation 7.2 formalizes how system reliability becomes the product of individual component reliabilities:

$$R_{\text{system}}(t) = \prod_{i=1}^{N} R_i(t) = \prod_{i=1}^{N} e^{-\lambda_i t} \tag{7.2}$$

For N identical components with individual failure rate λ, equation 7.3 gives the identical-component simplification:

$$R_{\text{system}}(t) = e^{-N\lambda t} \tag{7.3}$$

The system failure rate becomes $N\lambda$, and equation 7.4 expresses how the system MTBF scales inversely with component count. This inverse scaling reveals the counterintuitive reality of the 9s of reliability at cluster scale:

$$\text{MTBF}_{\text{system}} = \frac{1}{N\lambda} = \frac{\text{MTBF}_{\text{component}}}{N} \tag{7.4}$$

Figure 7.2 visualizes the fundamental tension: as clusters grow, the expected time between failures shrinks below common training durations, making fault tolerance necessary for long-running fleet jobs.

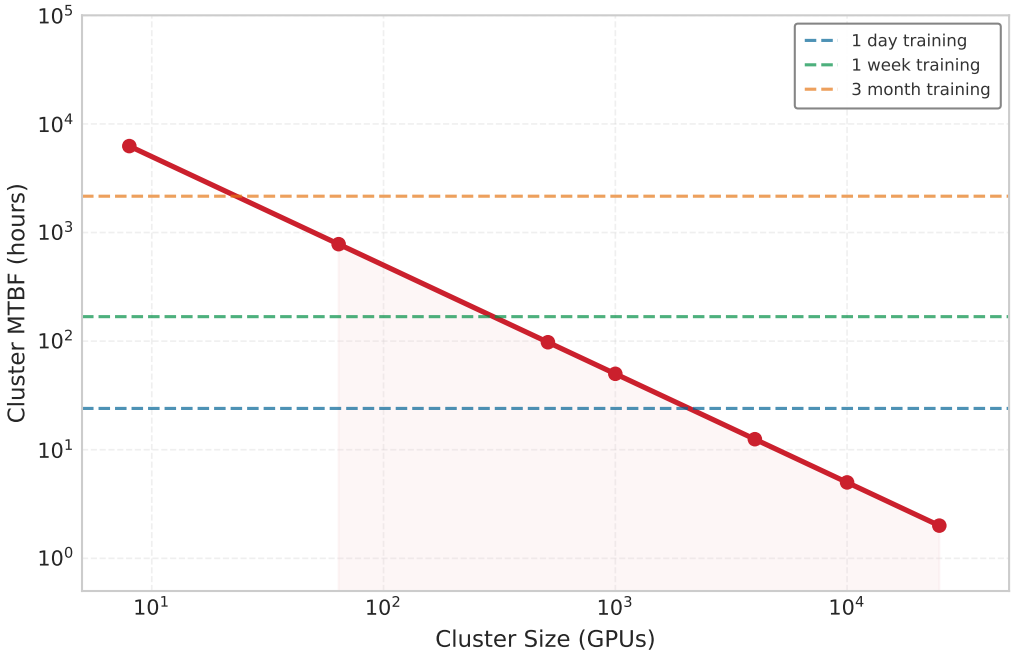


Figure 7.2: The Reliability Gap: As GPU count increases, cluster MTBF drops below common training durations. A 10,000-GPU cluster experiences a failure roughly every 5 hours, making a 3-month training run impossible without fault tolerance mechanisms.

7.1.2 The Young-Daly law: Optimal checkpointing

When failure is inevitable, the key engineering decision is how often to save progress. Checkpointing too frequently wastes time on I/O; checkpointing too rarely wastes time re-computing work after a failure. The **Young-Daly formula**[4] (principle 12) resolves that tension with a single square-root law:

$$\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$$

2 | **Poisson Process**: A statistical model for events occurring independently at a constant average rate. Reliability models assume hardware failures follow a Poisson distribution, leading to the exponential survival function $R(t) = e^{-\lambda t}$. This assumption simplifies fleet planning: the probability of at least one failure in a 10,000-component cluster is $1 - e^{-N\lambda t}$, making failure risk scale linearly with N but exponentially with t.

3 | **GPU MTBF Variation**: This chapter uses 50,000 hours as an A100-class planning constant, not as a guarantee for every deployed GPU. Published fleet studies from Google TPUv4 pods and Meta GPU clusters report that field reliability depends on topology, cooling, workload, and operational practice (Zu et al. 2024; Kokolis et al. 2025; Dubey et al. 2024). The planning constant therefore anchors the arithmetic, while operators should substitute measured fleet rates when they have them.

4 | **Young-Daly Formula**: J. W. Young derived the first-order optimal checkpoint interval in a 1974 *Communications of the ACM* paper; John Daly independently refined it in 2006 with tighter second-order bounds (Young 1974; Daly 2006). The formula's square-root relationship ($\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$) means that halving system MTBF only increases optimal checkpoint frequency by $\sqrt{2} \approx 1.4\times$, explaining why doubling cluster size does not demand doubling checkpoint I/O bandwidth.

T_{write} in distributed ML training is not the time to flush a small process state—it is the time required to transfer hundreds of gigabytes of FP32 optimizer state (momentum and variance tensors for every trainable parameter) plus model weights to durable storage. For a 70B-parameter model with full Adam state, a single checkpoint can reach several hundred gigabytes; for the later 175B-parameter running example, that figure exceeds a terabyte. Compressing T_{write} therefore requires purpose-built high-bandwidth storage, and T_{write} itself is the central constraint that determines how tightly the checkpoint interval can be set before I/O overhead displaces productive training compute.

The optimal interval balances the cost of writing checkpoints against the expected cost of reworking lost progress, the U-shaped trade-off figure 7.3 plots. Its scaling consequence is the one to carry forward: as clusters grow, system MTBF drops, so the optimal interval shrinks, demanding higher-bandwidth storage (Chapter 4) to keep T_{write} small before the "checkpoint tax" consumes the cluster's compute capacity. Because the relationship is a square root, halving MTBF tightens the interval by only $\sqrt{2} \approx 1.4\times$, so doubling cluster size does not demand doubling checkpoint I/O bandwidth. Section E.2.1 carries out the full derivation and the tighter second-order bounds; the later Checkpointing section (section 7.6.1) applies the formula to the local cluster once its system MTBF is in hand.

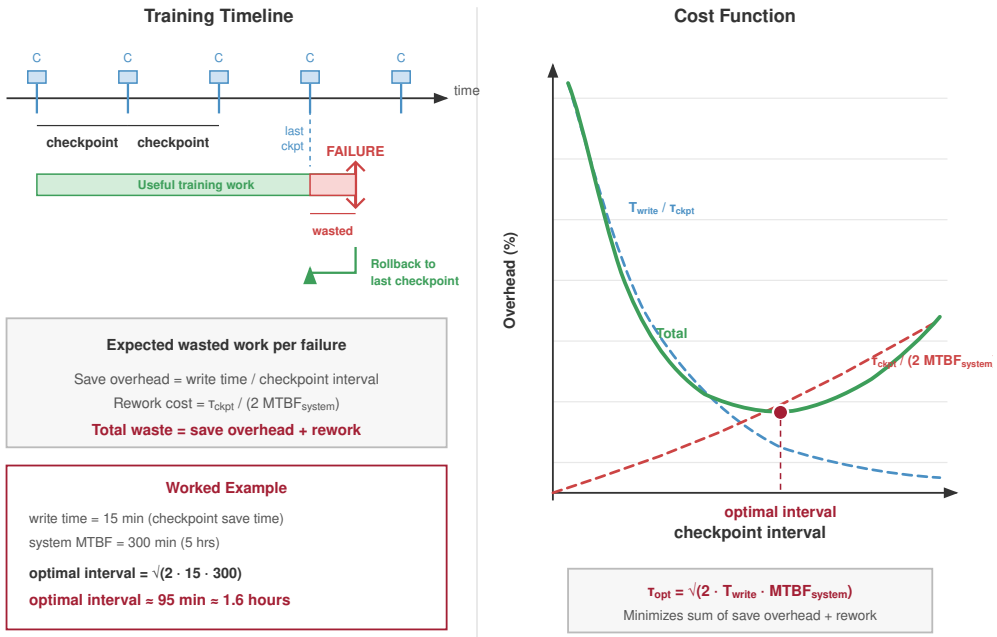


Figure 7.3: The Young-Daly Optimal Checkpoint: Total wasted work is the sum of *checkpointing overhead* (which decreases with interval τ_{ckpt}) and *rework cost* (which increases with τ_{ckpt}). The minimum point defines the optimal interval $\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$. For an illustrative cluster with a five-hour system MTBF and 15-minute write time, the optimal interval is approximately 1.6 hours.

A quick availability calculation shows the same scaling pressure from another angle.

Napkin Math 7.1: The 9s of reliability

Problem: A cluster of 10,000 GPUs runs with each GPU at 99.99 percent availability (only 52.6 minutes of downtime per year). What is the probability that the *entire cluster* is up at the same instant?

Math:

1. **Single GPU availability probability** (R_{GPU}): A GPU with 99.99 percent availability has an all-up probability $R_{\text{GPU}} = 0.9999$ at a randomly chosen instant.
2. **Cluster availability probability** (R_{cluster}): $R_{\text{cluster}} = (R_{\text{GPU}})^N$.
3. **Result**: $(R_{\text{GPU}})^N \approx 0.37$.

Systems insight: Even with 99.99 percent reliable hardware, a 10,000-GPU cluster has only a 37 percent chance that every GPU is simultaneously available. Hardware reliability alone is insufficient; software must handle failures automatically.

The linear relationship between component count and failure rate has a direct implication: adding GPUs turns rare component failures into a continuous system condition.

Systems Perspective 7.1: Scale transforms failure

A single GPU with an MTBF of 50,000 hours (5.7 years) absorbs the occasional failure with manual intervention, but a 10,000-GPU cluster with that same per-GPU reliability has a GPU-only system MTBF of just 5 hours: failures arrive multiple times per day. Systems must be designed expecting failure, not hoping to avoid it.

7.1.3 Quantitative reliability analysis

The GPU-only calculation in the opening reliability example supplies the scale intuition; a real training system then composes GPUs, HBM, links, power supplies, storage paths, and redundancy into one failure budget. With MTBF and the FIT rate already defined (FIT $= 10^9/\text{MTBF}$, so a 50,000-hour GPU contributes 20,000 FIT), the new question is how component rates *compose*. The composition rule depends on the redundancy structure. For **series systems** (for example, a node where all 8 GPUs must work), failure of any component fails the whole, so reliabilities multiply and rates add:

$$R_{\text{system}}(t) = \prod_{i=1}^{N} R_i(t), \qquad \text{MTBF}_{\text{system}} = \frac{1}{\sum \lambda_i}$$

which is why adding components reduces system MTBF linearly. For **parallel systems** providing redundancy (for example, active-active model replicas), failure occurs only when all redundant components fail simultaneously:

$$R_{\text{system}}(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t))$$

Composition also exposes which subsystem dominates the budget, and memory is the consequential case. Consider training an Archetype A (GPT-4/Llama-3) 70B model (section 1.6.1) on 1,024 A100 GPUs. Unprotected HBM is the binding term: at roughly 250 FIT per megabit, the cluster's $1,024 \times 80$ GB of memory accumulates so many failure sites that uncorrected corruption would strike roughly every 20 seconds, making large-scale training impossible. Error-correcting-code (ECC) memory protection is therefore not optional. ECC detects and corrects common single-bit errors, cutting the effective soft-error rate by roughly two orders of magnitude and pushing memory back below the logic and interconnect terms in the budget, though residual multi-bit or escaped corruptions remain as the silent-data-corruption problem.

These per-component FIT figures describe the silicon's intrinsic soft-error budget, which is why they differ from the 50,000-hour whole-device MTTF used in the worked cascade that follows: the device MTTF folds in every field failure mode (thermal stress, power events, mechanical wear), not just the logic and memory soft-error rates isolated here. The cascade composes those whole-device rates into the cluster figure that actually sets the checkpoint interval; this estimate only establishes why memory protection is the precondition for everything that follows.

7.1.4 Worked example: Cluster MTBF calculation

Consider a training cluster designed for large language model development with the following specifications:

- 10,000 NVIDIA H100 GPUs
- Individual GPU MTBF: 50,000 hours
- Each GPU connected to host via PCIe (MTBF: 200,000 hours)
- Each node contains 8 GPUs with shared power supply (MTBF: 100,000 hours)
- Network infrastructure per node (NIC, cables): MTBF 150,000 hours

Together, these components define the failure domain used in the rate calculation below.

Step 1: Calculate failure rate per GPU subsystem

Each GPU operates within a failure domain that includes the GPU itself, its PCIe connection, and proportional shares of the power supply and network infrastructure.

$$\lambda_{\text{GPU}} = \frac{1}{50{,}000} = 2.0 \times 10^{-5} \text{ failures/hour}$$

$$\lambda_{\text{PCIe}} = \frac{1}{200{,}000} = 0.5 \times 10^{-5} \text{ failures/hour}$$

$$\lambda_{\text{power/GPU}} = \frac{1}{8} \times \frac{1}{100{,}000} = 0.125 \times 10^{-5} \text{ failures/hour}$$

$$\lambda_{\text{network/GPU}} = \frac{1}{8} \times \frac{1}{150{,}000} = 0.083 \times 10^{-5} \text{ failures/hour}$$

Step 2: Calculate total per-GPU failure rate

$$\lambda_{\text{total/GPU}} = (2.0 + 0.5 + 0.125 + 0.083) \times 10^{-5} = 2.708 \times 10^{-5} \text{ failures/hour}$$

Step 3: Calculate system failure rate and MTBF

$$\lambda_{\text{system}} = 10{,}000 \times 2.708 \times 10^{-5} = 0.2708 \text{ failures/hour}$$

$$\text{MTBF}_{\text{system}} = \frac{1}{0.2708} = 3.69 \text{ hours}$$

This result means the cluster experiences a failure approximately every 3.7 hours on average. The expected failure cadence is 6.5 failures/day; a training run lasting one week will experience approximately 45 failures. Any training system operating at this scale must treat failure as a continuous condition, not an exceptional event. Section E.1.2 works this same cascade through step by step, composing per-component rates into a fleet-wide system MTBF, so a reader can rerun the calculation for a different node design or cluster size.

The cluster MTBF scaling in table 7.1 isolates the GPU-only baseline across cluster sizes. The full-system calculation is lower once PCIe, power, and network failure domains are included.

Table 7.1: Cluster MTBF Scaling: GPU-only mean time between failures decreases linearly with cluster size, transforming failures from rare events to continuous operating conditions at scale. Additional host, power, storage, and network failure domains reduce system-level MTBF further.

Cluster Size (GPUs)	Individual GPU MTBF	Cluster MTBF	Expected Failures per Day
8	50,000 hours	6250 hours (260 days)	0.004
64	50,000 hours	781.2 hours (33 days)	0.03
512	50,000 hours	97.7 hours (4 days)	0.25
1,000	50,000 hours	50 hours (2 days)	0.48
4,000	50,000 hours	12.5 hours	1.9

Cluster Size (GPUs)	Individual GPU MTBF	Cluster MTBF	Expected Failures per Day
10,000	50,000 hours	5 hours	4.8
25,000	50,000 hours	2 hours	12

The theoretical $1/N$ scaling in table 7.1 is not merely a textbook exercise. Figure 7.4 overlays published measurements and fitted projections from Meta's production clusters against the theoretical curve. The Kokolis et al. (2025) study of Meta's Research SuperCluster reported measured MTTF values for observed RSC job sizes and projected a 1.8-hour MTTF for 16,384-GPU jobs; Meta's Llama 3 training report independently documented 419 failures across 54 days on 16,384 H100 GPUs, an observed MTTF of about 3.1 hours. Together, the measured Llama run and Kokolis projection put 16,384-GPU training on a 2-to-3-hour failure cadence. This evidence transforms the theoretical argument from an abstract scaling law into an engineering constraint that determines checkpoint frequency, recovery architecture, and infrastructure investment.

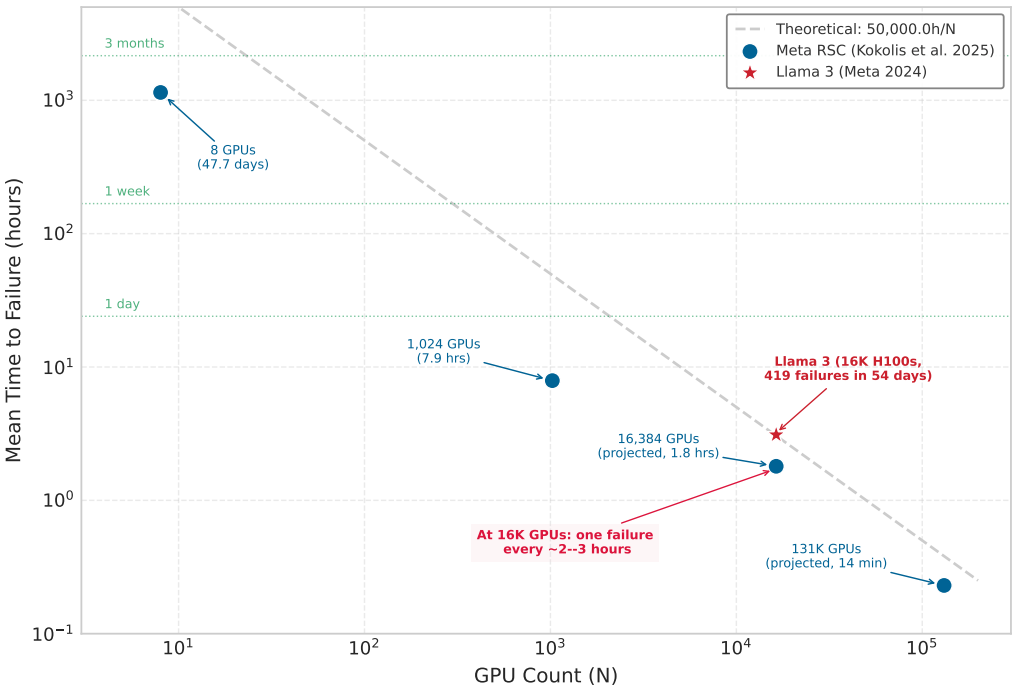


Figure 7.4: Published Failure Rates from Production Clusters: Measured MTTF values and fitted projections from Meta's RSC clusters (Kokolis et al. 2025) plotted with Llama 3 training measurements (Dubey et al. 2024) against the theoretical 1/N curve. At 16,384 GPUs, the Kokolis projection and Llama 3 empirical measurement both fall in the 2-to-3-hour failure range, confirming that fault tolerance is a mathematical necessity, not an edge case. Data sources: Kokolis et al., arXiv:2410.21680; Meta, arXiv:2407.21783.

7.1.5 Failure taxonomy

The MTBF calculations in the reliability analysis quantify *how often* failures occur, which is critical for setting checkpoint intervals and sizing recovery infrastructure. Designing effective fault tolerance also requires understanding *what kind* of failures occur. Use the vocabulary carefully: a fault is the underlying defect or event, an error is the corrupted state it produces, and a failure is the externally visible behavior the training system must handle (Avizienis et al. 2004). A network partition that resolves in seconds demands different handling than a permanent GPU failure. A silent memory corruption that produces incorrect gradients requires different detection mechanisms than a node crash that stops responding entirely. Failure characteristics guide the selection of

appropriate recovery mechanisms. The taxonomy presented here classifies failures along two primary dimensions: temporal behavior (transient vs. persistent) and failure manifestation (fail-stop vs. Byzantine) (Constantinescu 2008; Lamport et al. 1982).

7.1.5.1 Transient failures

Transient failures occur temporarily and resolve without intervention, so the recovery question is whether the system can retry or validate the affected work before corrupted state propagates. Four common transient patterns matter for ML systems:

- **Network packet loss**: Drops packets while retransmission succeeds.
- **Memory bit flips**: Corrupt individual bits through cosmic ray induced single-event upsets.[5]
- **Thermal throttling**: Temporarily reduces performance during temperature spikes.
- **Software timeouts**: Arise from temporary resource contention.

Transient failures are particularly insidious in ML training because they may not trigger explicit errors. A transient memory bit flip during gradient computation produces incorrect gradients that propagate through subsequent training steps. The model continues training but produces subtly degraded results. Large-scale SDC studies show escaped corruption, while DRAM field studies show memory errors are common enough that fleet-level monitoring matters (Dixit et al. 2021; Sridharan et al. 2015; Schroeder et al. 2009). In ML training, that same undetected-corruption pattern can surface as degraded gradients, anomalous loss curves, or rollback-worthy checkpoints rather than an explicit crash[6].

The appropriate response to transient failures depends on detection capability. Errors that trigger explicit exceptions can be handled through retry logic. Silent corruption requires validation mechanisms such as gradient checksums, periodic model evaluation, and statistical monitoring of training dynamics.

7.1.5.2 Fail-stop failures

Fail-stop failures cause components to cease operation entirely and detectably. The failed component stops responding to requests and can be identified through timeout mechanisms:

- **GPU hardware failure**: Makes a device unresponsive.
- **Node crash**: Terminates all processes on a machine.
- **Network partition**: Isolates the node from the cluster.
- **Storage failure**: Prevents checkpoint reads or writes.

In synchronous distributed training, fail-stop failures carry a particularly severe consequence: they stall the entire job. AllReduce collectives require every participating rank to contribute its gradient shard before the reduction can complete. A single GPU that goes silent—whether from a hardware fault, a process crash, or a network partition—blocks every other GPU in the ring until the timeout expires. On a 10,000-GPU job, that means 9,999 accelerators sitting idle, accumulating billable hours against a run that has made zero forward progress, until the coordinator finally declares the missing rank failed and triggers recovery.

Fail-stop failures are the easiest class to handle because detection is straightforward: the component stops responding. Recovery involves replacing the failed component and restoring state from the most recent checkpoint. The primary challenge is minimizing detection time and recovery latency.

Detection time T_{detect} typically involves heartbeat mechanisms where each GPU rank periodically signals liveness to a coordinator. If no heartbeat arrives within the timeout period T_{timeout}, the coordinator declares failure. Setting T_{timeout} requires balancing false positive rate against detection latency. False positives declare healthy workers failed due to transient delays, while slow detection wastes compute during the detection window—a cost that scales linearly with the number of GPUs blocked in the collective.

For a heartbeat interval of H seconds and network-delay standard deviation σ_d in seconds, equation 7.5 defines the timeout heuristic:

$$T_{\text{timeout}} = H + k\sigma_d \tag{7.5}$$

[5] **Single-Event Upset (SEU)**: From particle physics, where "upset" denotes a nondestructive state change. Cosmic rays and alpha particles from chip packaging can flip bits in memory and logic; the observed rate depends on altitude, process technology, packaging materials, shielding, and workload (Ziegler et al. 1996; Baumann 2005). ECC memory corrects single-bit errors but cannot handle every multi-bit upset in the same word, leaving a residual silent-corruption risk that compounds across the terabytes of state in large-scale ML training.

[6] **Silent Data Corruption (SDC)**: At fleet scale, SDC can bypass normal exception paths: infrastructure studies have documented defective hardware producing valid-looking but wrong results, while DRAM field studies show memory errors are common enough that fleet-level monitoring matters. For ML training, the practical debugging consequence is that the absence of an error message does not imply correct computation; systems need statistical checks on loss, gradients, and model state.

Here, k is a dimensionless safety multiplier that typically ranges from three to five to achieve low false positive rates while maintaining reasonable detection speed. ML training clusters built on dedicated high-bandwidth fabrics (InfiniBand, NVLink) exhibit very low baseline jitter, so σ_d is functionally small under normal conditions. The practical difficulty is distinguishing a genuinely dead rank from one that is temporarily slow—for example, a GPU lagging because it is writing a large checkpoint to attached storage while the collective is already waiting. Tuning T_{timeout} too tightly causes checkpoint writes to trigger false failure declarations; tuning it too loosely extends the window during which every other rank in the AllReduce collective is blocked idle.

7.1.5.3 Byzantine failures

Where a fail-stop component simply goes silent and is detected and replaced, a far more insidious class keeps running but produces incorrect results. A GPU that returns wrong gradients without throwing errors, a network that delivers corrupted packets that pass CRC checks, or a worker that computes different results for identical inputs all exemplify **Byzantine failures**, the most challenging class in distributed systems (Lamport et al. 1982). In ML systems, this category includes silent data corruption, numerical instability, determinism violations, and adversarial corruption; the common property is that the worker still participates while the value it contributes can poison shared state.

The physics of silent corruption At the nanometer scale of advanced transistors, hardware is not *deterministic*; it is *probabilistic*. Silent data corruption (SDC) is driven by two primary mechanisms. **Single-event upsets (SEUs)** occur when high-energy particles (cosmic rays at sea level, alpha particles from packaging materials) strike memory cells or logic gates, flipping a bit from 0 to 1; at 10,000+ GPUs, this is a statistical certainty. **Manufacturing variances** appear when "marginal" chips that pass initial QA exhibit bit flips only under specific voltage/temperature conditions, such as the intense di/dt swings of a backward pass.

Facebook documented a pervasive SDC issue where a hardware fault caused a valid file to be reported as "size zero" during decompression (Dixit et al. 2021). As figure 7.5 illustrates, the system "worked" (no crash), but data was silently deleted. In ML, this manifests as valid-looking but numerically corrupted gradients.

Real-world evidence of SDC in production systems confirms these risks. Figure 7.6 shows corrupted data blocks accumulating in a shuffle and merge database at Google, where even a small fraction of corrupted blocks can cascade into significant data quality degradation.

Google reported that SDC in TPU pods can manifest as sudden, inexplicable spikes in gradient norm (figure 7.7) (Dean 2024). A single bit flip in an exponent can turn a small gradient into a numerically enormous value, corrupting the training trajectory if the anomaly is not detected.

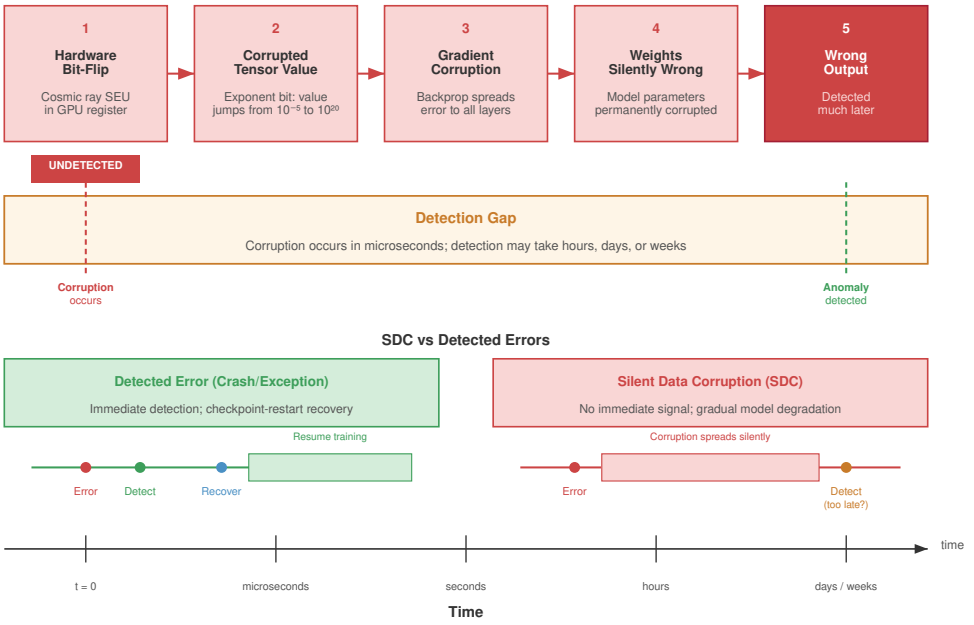
Google addresses this class of failure with system-level mitigation, including spare capacity and sanity checks that can drain suspect chips when loss or gradient monitors flag anomalies (figure 7.8) (Dean 2024). This moves reliability from the component, which we cannot trust, to the system, which verifies the result.

The hot spare pattern in figure 7.8 illustrates one approach to SDC mitigation, but recognizing silent corruption in the first place requires understanding what distinguishes it from benign training noise.

Checkpoint 7.1: Detecting silent corruption

Verify your understanding of Byzantine failures and SDC:

- ☐ **SDC danger**: Explain why silent data corruption (SDC) is more dangerous for ML training than a simple node crash.
- ☐ **Hot spare vs. restart**: Compare a **hot spare** redundancy strategy with traditional **checkpoint-restart** in terms of recovery latency.
- ☐ **Sanity checker**: Identify the metric used as a "sanity checker" to identify potential hardware faults: GPU temperature or gradient norm.
- ☐ **Byzantine scope**: Assess the True or False claim that Byzantine failures only occur when a malicious actor intentionally sabotages the training cluster.



SDC propagation chain: hardware fault to model-level corruption with extended detection latency

Figure 7.5: Silent Data Corruption Propagation: Unexpected faults can return incorrect file sizes, leading to data loss during decompression and propagating errors through distributed querying systems. This example from Facebook emphasizes how silent errors bypass standard exception handlers. Source: (Dixit et al. 2021).

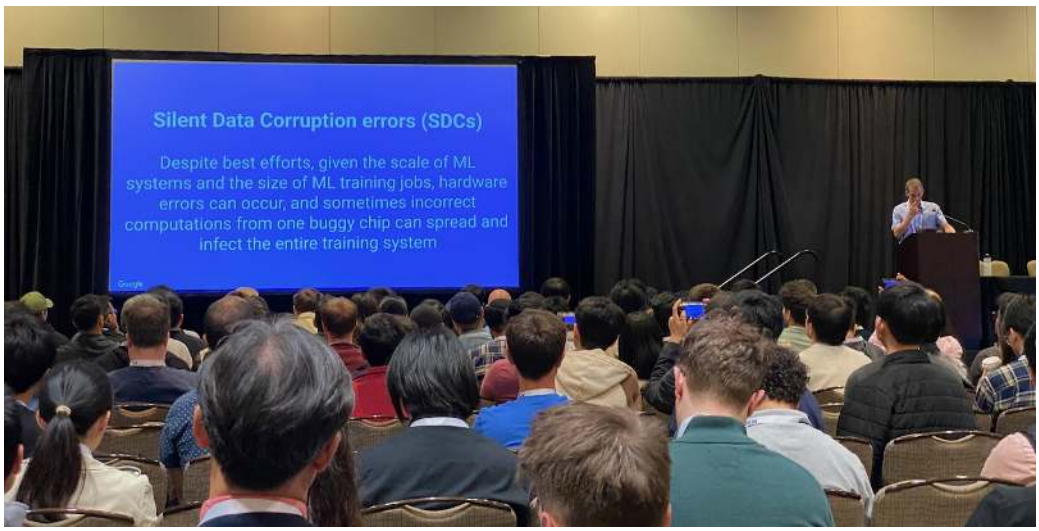


Figure 7.6: Silent Data Corruption in Spark: AI systems that use large-scale data processing frameworks such as Spark are vulnerable to silent data corruption (SDC) accumulating during data transfer and storage. SDC manifests in a shuffle and merge database, highlighting corrupted data blocks (red) amidst healthy data (blue/gray). Source: (Dean 2024).

Silent-corruption detection is the first defense, but the same logic extends to any Byzantine worker whose output remains syntactically valid. These failures are particularly dangerous in distributed training because the standard assumption that workers compute identical gradients for identical data no longer holds. A single Byzantine worker can corrupt the averaged gradient, potentially

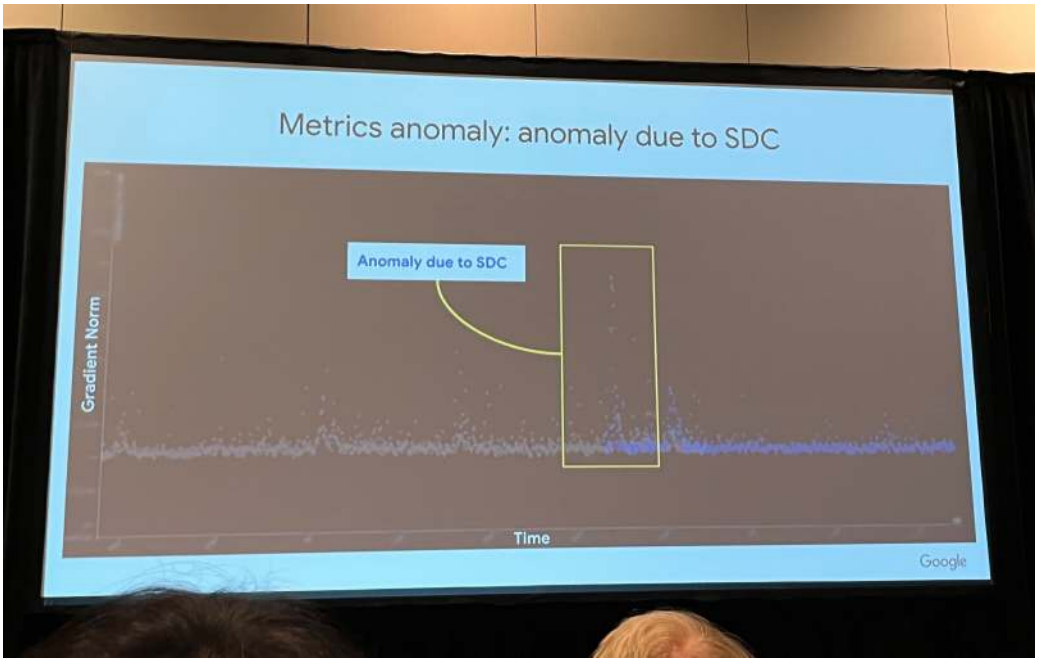


Figure 7.7: Gradient Norm Deviation: Transient hardware faults, such as silent data corruption (SDC), can disrupt optimization by causing abrupt changes in gradient norms. Google's production examples show SDC anomalies surfacing as visible gradient-norm spikes. Source: (Dean 2024).

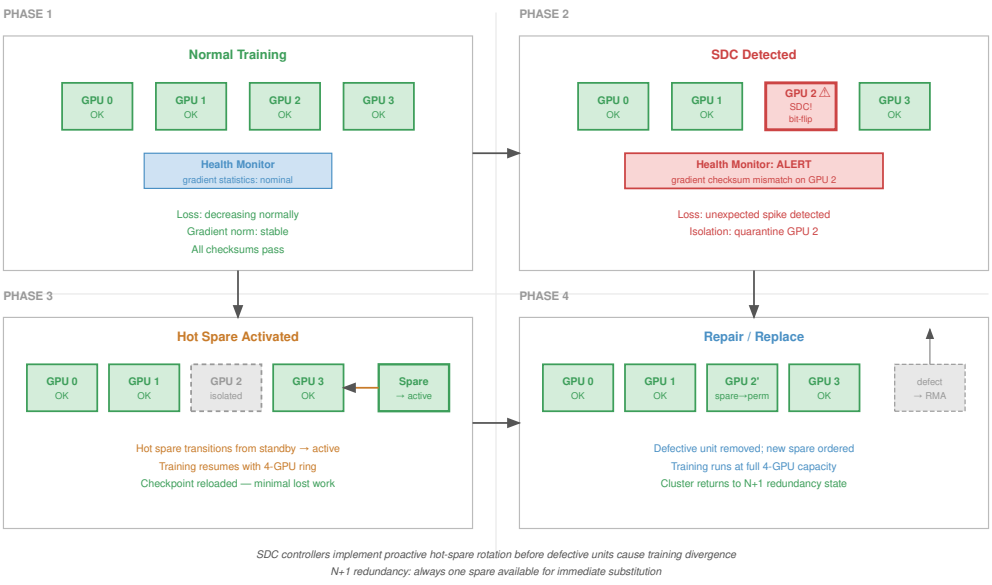


Figure 7.8: Hot Spare Redundancy: Google's data-center examples use spare capacity and sanity checking to keep ML training moving despite hardware faults, transitioning work away from suspect resources when monitors flag anomalies. This approach contrasts with parallel redundancy techniques such as dual modular redundancy and triple modular redundancy by providing a reactive fault-tolerance mechanism. Source: (Dean 2024).

causing training to diverge or converge to a poor solution. Figure 7.9 contrasts the straightforward detection of fail-stop failures with the insidious nature of Byzantine corruption.

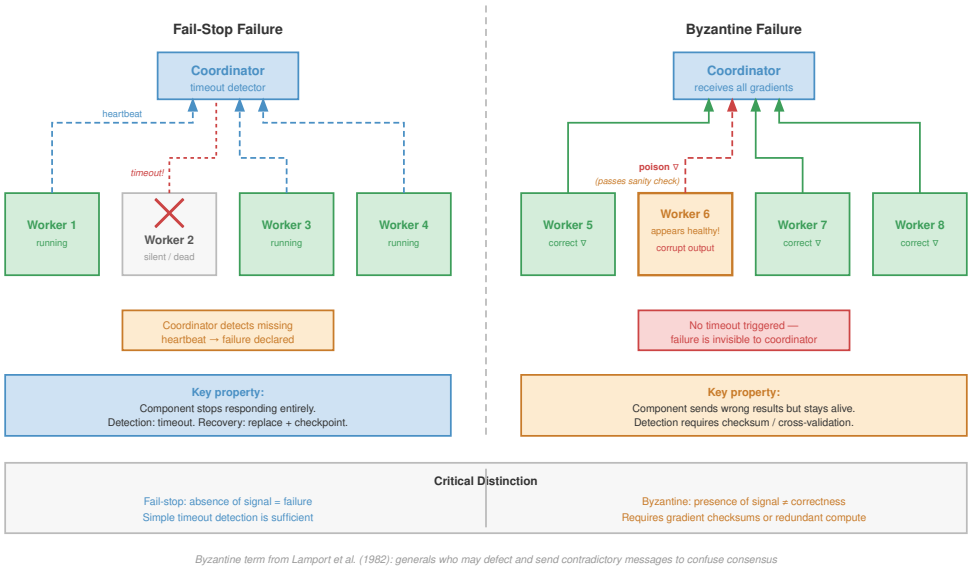


Figure 7.9: Fail-Stop vs. Byzantine Failures: In the fail-stop model (left), a failed worker simply ceases to send messages, which is easily detected by timeouts. In the Byzantine model (right), a failed worker continues to participate but sends incorrect data (for example, corrupted gradients reported as valid), which can poison the global model state if not detected by redundant validation.

Detection of Byzantine failures requires redundant computation. Multiple workers computing gradients for the same data enable comparison of results. Statistical outlier detection can identify workers consistently producing anomalous gradients. These detection mechanisms add computational overhead and may not catch subtle corruption.

Byzantine-resilient distributed training algorithms exist but impose significant overhead. Algorithms such as Krum (Blanchard et al. 2017) and coordinate-wise trimmed mean (Yin et al. 2018) compute aggregates that are robust to a bounded number of Byzantine workers, but they require more communication and computation than simple averaging. The systems consequence is visible here: corrupted gradients can push the optimizer toward an unreliable model state while the training job appears healthy. Reliability therefore has to protect semantic correctness, not only process liveness.[7]

7.1.5.4 Correlated failures

The reliability calculations in section 7.1.1 assume independent failures. Real systems exhibit correlated failures when shared dependencies fail many components at once:

- **Power supply failure**: Takes down every GPU in a node.
- **Network switch failure**: Isolates all attached nodes.
- **Cooling system failure**: Forces thermal shutdown across racks.
- **Software bug**: Crashes every process using a faulty CUDA driver version.
- **Operator error**: Misconfigures an entire cluster.

Correlated failures violate the independence assumption underlying equation 7.3. When failures are correlated, the actual system reliability is lower than the formula predicts. Correlated failures can also defeat redundancy strategies. Three replicas of a model provide no availability benefit if all three run on the same power domain and a power failure takes out all three simultaneously.

Defending against correlated failures requires understanding failure domains and ensuring redundancy spans independent failure domains. Table 7.2 catalogs common failure domains in ML infrastructure, from single GPUs to entire data center regions, each requiring distinct mitigation

7 **Byzantine-Resilient ML**: Named after Lamport's 1982 "Byzantine Generals Problem," these algorithms (Krum, trimmed mean, signSGD) tolerate a bounded fraction of corrupted workers, with the exact bound depending on the algorithm and assumptions. The trade-off is concrete: for N workers, Krum requires pairwise distance computations over worker gradients, so overhead grows quadratically with worker count and directly competes with the throughput gains of data parallelism (Blanchard et al. 2017).

strategies. Figure 7.10 illustrates how these domains nest hierarchically, with failures propagating downward through the containment structure.

Table 7.2: Failure Domains in ML Infrastructure: Understanding failure domain boundaries enables placement of redundant components across independent domains, preventing correlated failures from defeating redundancy strategies.

Failure Domain	Impact Scope	Typical Recovery Time	Mitigation Strategy
Single GPU	1 GPU	Seconds (spare)	Hot spare, elastic training
Node (power/OS)	8 GPUs	Minutes	Checkpoint, node replacement
Rack (ToR switch)	32–64 GPUs	Minutes to hours	Cross-rack redundancy
Power domain	100–500 GPUs	Hours	Multiple power feeds
Data Center region	All GPUs	Hours to days	Geographic distribution
Software version	All GPUs running version	Minutes (rollback)	Staged rollouts

The nesting of failure domains in table 7.2 has direct consequences for redundancy placement: placing replicas within the same domain provides zero protection against that domain's failure mode.



Figure 7.10: Hierarchy of Failure Domains: Failure domains are often nested or overlapping. A GPU failure affects one device. A node failure affects 8 GPUs. A rack switch failure affects 32-64 GPUs. A power distribution unit (PDU) failure may affect multiple racks. Effective fault tolerance requires placing replicas across independent domains (for example, different racks or rows) to survive correlated failures.

The key insight from figure 7.10 is that redundancy is effective only when replicas span independent failure domains: placing three model replicas on the same rack provides no protection against the rack switch or PDU that all three share.

Checkpoint 7.2: Mapping failure domains

Verify your understanding of how failure domains nest and their operational impact:

- ☐ **Blast radius**: Calculate how many 8-GPU nodes a rack switch failure typically affects in a standard data center configuration.
- ☐ **Zone correlation**: Explain why placing all 3 model replicas in the same Zone (for example, `us-east-1a`) fails to provide protection against a regional power outage.

- ☐ **Staged rollouts**: Identify the failure domain addressed by **Staged Rollouts** rather than hardware redundancy.
- ☐ **Nested domains**: Explain **Hierarchical Containment** and decide whether the corresponding Rack failure domain still matters when a Zone fails.

War Story 7.1: The supercomputer that rerouted around faults (2024)

Context: Google's TPUv4 machine-learning supercomputer used 4,096 TPU chips connected by a custom 3D torus Inter-Chip Interconnect (ICI) fabric, and Google engineers published their operating experience with automatic fault resiliency and hardware recovery (Zu et al. 2024).

Failure mode: At that scale, a failed machine, chip, ICI cable, or optical circuit switch was not merely a lost component. It could disrupt the topology that synchronous training jobs relied on for collective communication.

Consequence: Google built a control plane that detected faults, reconfigured the ICI fabric, and routed jobs around failures or onto spare healthy cubes. The paper reports 99.98 percent system availability while gracefully handling hardware outages experienced by about 1 percent of training jobs.

Systems lesson: Fault tolerance is a topology-management problem. In tightly coupled accelerator fleets, recovery must preserve the communication graph, not merely replace an individual component; otherwise one local hardware fault can become a cluster-wide training interruption.

7.1.6 The bathtub curve and hardware lifecycle

The failure taxonomy classifies failure types and domains, answering what kind of failures occur. Equally important for designing fault tolerance is understanding when in a component's lifetime failures are most likely to occur. Hardware failure rates are not constant over component lifetime. Figure 7.11 illustrates the bathtub curve, a well-established model in reliability engineering that describes how failure rates vary across three distinct phases:

The first phase, infant mortality, exhibits elevated failure rates from manufacturing defects, improper installation, and early-life wear-out of marginal components. This phase typically lasts days to weeks for electronic components. Burn-in testing[8] operates components under stress conditions before deployment to precipitate infant mortality failures before production use.

After surviving infant mortality, components enter the useful life phase, where they exhibit relatively constant failure rates under the standard reliability-engineering model (Birolini 2017; Klutke et al. 2003). This phase represents the longest portion of component lifetime and is the period where the exponential reliability model in equation 7.1 applies most accurately. For data center GPUs, the useful-life window is an operational planning concept shaped by refresh cycles, cooling, utilization, and observed fleet health rather than a fixed physical duration.

As components age, they enter the wear-out phase, where failure rates increase due to accumulated wear. For GPUs, wear mechanisms include electromigration[9] in circuits, thermal cycling stress on solder joints, and degradation of thermal interface materials. The onset of wear-out depends heavily on operating conditions; components operated at high temperatures or with frequent thermal cycling enter wear-out earlier.

The practical implication for ML systems is that fleet-wide failure rates depend on age distribution. A cluster populated entirely with new GPUs will experience elevated failure rates during the first few weeks, followed by a stable period, then increasing failures as the fleet ages. Mixed-age fleets exhibit more consistent aggregate failure rates because different cohorts are in different lifecycle phases.

The three phases in figure 7.11 have direct operational consequences: burn-in testing filters infant mortality before deployment, while predictive analytics using GPU telemetry (temperature trends, error counts, performance degradation) targets the wear-out phase, enabling scheduled component

8 **Burn-in Testing**: Components operate at elevated temperature (85–125 degrees C) and voltage for 24–168 hours to precipitate infant mortality failures before production. Large operators may burn in accelerators before deployment, reducing the infant-mortality failures that fresh bare-metal hardware can otherwise expose to early jobs.

9 **Electromigration**: Gradual displacement of metal atoms in conductors by electron momentum transfer, first characterized in early electromigration studies and later modeled in Black's reliability equation (Black 1969). Mean time to failure decreases with higher current density and higher temperature, so sustained high-power accelerator workloads make thermal management a direct determinant of fleet lifespan.

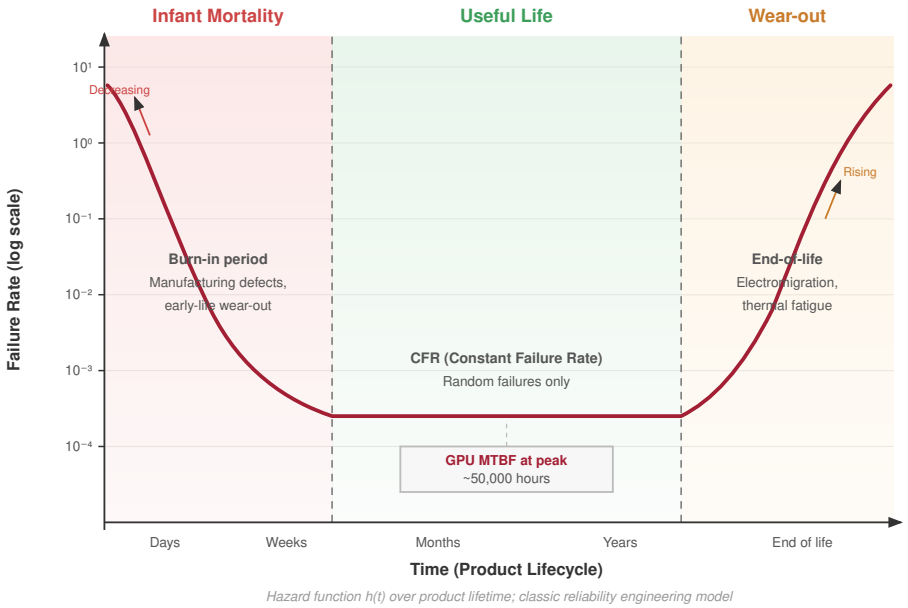


Figure 7.11: The Bathtub Curve: Hardware failure rates $\lambda(t)$ vary over time. (1) Infant Mortality: high failure rate initially due to manufacturing defects. (2) Useful Life: constant, low failure rate where random failures dominate. (3) Wear-Out: increasing failure rate as components age. Burn-in testing aims to filter out infant mortality failures before deployment.

replacement during maintenance windows rather than unplanned outages during training runs. ML infrastructure teams apply this model directly to cluster scheduling and fleet admission. Rather than treating fresh accelerator pods as immediately equivalent to proven production nodes, operators often run stress workloads before assigning the hardware to high-value training. Sustained matrix-multiply loops, memory tests, and communication tests expose marginal devices before they touch a multi-week pretraining run. Operators repair or replace an accelerator that fails during this screening phase before it ever touches a high-value training run; an accelerator that survives enters the useful-life period where the constant-rate exponential model applies. This discipline matters because the cost asymmetry is stark: catching an infant-mortality failure during screening costs a few hours of idle accelerator time, while the same failure during a multi-week foundation model pretraining run can invalidate days of gradient accumulation and force a checkpoint rollback. Scheduling policy reflects this asymmetry: cluster operators typically assign fresh or recently repaired nodes to short exploratory jobs or inference serving first, not to months-long pretraining jobs, until they have accumulated enough operating hours to leave the infant-mortality risk window.

7.1.7 Model-type diversity in failure impact

While the mathematics of failure rates apply universally, the failure impact by model type differs dramatically. The impact of losing an hour of training depends on what that training costs, how much state must be recovered, and how long recovery takes. Table 7.3 quantifies these factors across model architectures, revealing orders-of-magnitude variation from LLMs incurring millions of dollars in wasted compute to vision models losing modest amounts of progress.

Table 7.3: Failure Impact by Model Type: The cost of training failures varies dramatically by model type, driven by training duration, checkpoint overhead, and the value of accumulated training state. These differences demand model-specific fault tolerance strategies.

Model Type	Typical Training Duration	Checkpoint Size	State Sensitivity	Failure Cost
Archetype A (70B-class dense LLM)	2–4 weeks	350–700 GB	High (position in curriculum)	$2–5M compute per 24hr loss
Vision (ViT-Large)	1–3 days	1–2 GB	Medium (augmentation state)	$10–50K per day loss
Archetype B (DLRM at Scale)	Continuous	2–4 TB (embeddings)	Very High (embedding freshness)	Revenue impact per hour
Speech (Whisper-scale)	3–7 days	5–10 GB	Medium	$50–200K per day loss
Scientific (AlphaFold)	Days to weeks	10–50 GB	High (exploration state)	Research delay

Large Language Models experience the highest absolute failure costs due to their extended training durations and the computational expense of each training hour. For example, a large training run consuming 25,000 GPUs at approximately $2 per GPU-hour incurs $1.2M in compute costs per day. A failure that loses 24 hours of training progress costs $1.2M in wasted compute plus schedule delay. The checkpoint overhead spans a wide range: 70B-class dense LLM checkpoints can reach hundreds of gigabytes, and the 175B-parameter running example reaches 3.7 TB once FP32 Adam optimizer state is included.

Recommendation Systems present unique challenges because their training is often continuous rather than episodic. The value of a RecSys model derives partly from its freshness. Embeddings that capture recent user behavior outperform stale embeddings. A failure that loses hours of embedding updates may degrade recommendation quality in ways that directly impact revenue. Meta has documented that recommendation model freshness directly correlates with engagement metrics, making recovery time a business-critical metric.[10]

Vision Models occupy a middle ground with moderate training durations and manageable checkpoint sizes. The relatively small checkpoints enable frequent checkpointing with minimal overhead. A ViT-Large checkpoint in the 1–2 GB range imposes little overhead compared with large language or embedding-heavy recommendation workloads. Data augmentation state represents the primary state beyond model weights that must be preserved for reproducible recovery. The augmentation parameters and data shuffling seed must be captured.

Scientific Models such as those used in protein structure prediction or climate simulation often have unique state requirements beyond model parameters. AlphaFold-style training may maintain exploration state tracking which protein families have been sampled, preventing repetition during recovery. Drug discovery models may track which molecular configurations have been evaluated. This domain-specific state complicates checkpoint and recovery design.

10 **RecSys Freshness**: Meta's DLRM infrastructure documents that embedding staleness measured in hours produces measurable degradation in recommendation relevance and engagement metrics. This inverts the typical fault tolerance priority: for recommendation systems, minimizing recovery time matters more than minimizing checkpoint overhead, because stale embeddings directly reduce revenue.

7.1.8 Economic framework for fault tolerance investment

Fault tolerance mechanisms consume resources: storage for checkpoints, bandwidth for checkpoint writes, compute cycles for redundant calculations, and engineering time for implementation and maintenance. Rational investment in fault tolerance requires quantifying both the cost of failures and the cost of prevention.

Failure costs include wasted compute, schedule delay, opportunity cost, and engineering time. Wasted compute measures GPU-hours expended on training steps that must be repeated. Schedule delay captures how extended time to a trained model impacts business timelines. Opportunity cost recognizes that compute consumed by recovery cannot be used for other training. Engineering cost accounts for time spent debugging failures and manually recovering.

Prevention costs include storage, throughput overhead, recovery infrastructure, and complexity. Storage cost scales with model size and checkpoint frequency. Checkpoint writes consume memory bandwidth and may stall training. Recovery infrastructure requires spare capacity and automated recovery systems. Fault tolerant systems are harder to develop and debug.

Optimal investment in fault tolerance balances these costs. For small-scale training on a few GPUs where failures are rare, minimal fault tolerance may be cost-optimal. Infrequent checkpoints and manual recovery suffice. For large-scale training on thousands of GPUs where failures occur multiple times daily, extensive fault tolerance provides positive return on investment. Frequent checkpoints, automatic recovery, and elastic training become essential. Figure 7.3 visualizes how the trade-off between checkpoint overhead and recovery cost reaches an optimum that depends on both system MTBF and checkpoint write time.

Equation 7.6 presents a simplified economic model for expected cost per training run:

$$C_{\text{total}} = C_{\text{compute}} + E[N_{\text{failures}}] \times C_{\text{per-failure}} + C_{\text{ft}} \tag{7.6}$$

where C_{compute} is the base compute cost, $E[N_{\text{failures}}]$ is the expected number of failures during training, $C_{\text{per-failure}}$ is the cost per failure, and C_{ft} is the cost of fault tolerance mechanisms. The cost per failure includes wasted compute plus overhead.

Equation 7.7 formalizes when fault tolerance investment is justified:

$$\frac{\partial C_{\text{ft}}}{\partial x} < \frac{\partial (E[N_{\text{failures}}] \times C_{\text{per-failure}})}{\partial x} \tag{7.7}$$

where x represents investment in fault tolerance mechanisms. In practice, this means investing in fault tolerance until the marginal cost of additional protection exceeds the marginal reduction in failure costs.

Three foundational principles guide every design decision in this domain.

> Systems Perspective 7.2: Three rules of failure at scale
>
> 1. **At scale, failures are continuous, not exceptional.** A 10,000-GPU cluster experiences failures every few hours. Systems must be designed expecting failure as normal operation.
> 2. **Checkpoint intervals have an optimum.** The Young-Daly formula, $\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}}$, provides quantitative guidance for checkpoint frequency. This formula is derived in section 7.1.2.
> 3. **Training and serving have fundamentally different fault tolerance requirements.** Training tolerates minutes of recovery time; serving requires milliseconds. This difference demands entirely different approaches.

Rule 3—the training/serving divergence—sets the sequence: training recovery comes first, serving resilience later. Before either, the physical realities of what breaks must be understood, starting with the hardware faults that trigger these failures.

7.2 Hardware Fault Taxonomy

Consider what happens when a cosmic ray flips a single bit in a GPU's High Bandwidth Memory, or when thermal expansion causes a microscopic fracture in an NVLink connector. These physical events cascade into software errors that can corrupt a multi-week training run, but the recovery system does not observe "cosmic ray" or "fracture" directly. It observes symptoms: a gradient spike with no process crash, a rank that disappears from the collective, or a node that fails only when it heats under sustained load. Hardware taxonomy is useful only when it turns those symptoms into a recovery decision.

7.2.1 Hardware fault impact on ML systems

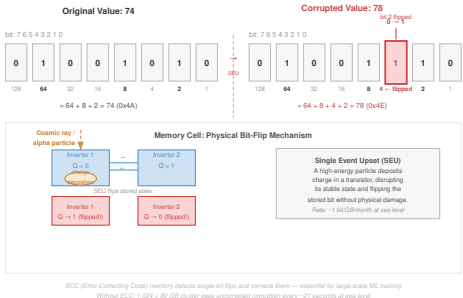
ML systems amplify the consequences of hardware faults beyond what traditional applications experience. Computational intensity creates millions of opportunities per second for faults to corrupt results. Training runs lasting days or weeks increase the probability of encountering faults. Small corruptions in model weights can cause large changes in output predictions, and distributed dependencies mean that a single-point failure can disrupt entire workflows.

A single bit-flip in a weight matrix illustrates the severity. If a critical weight in a ResNet-50 model flips from `0.5` to `-0.5` due to a transient fault affecting the sign bit in the IEEE 754[11] floating-point representation, the sign of a feature map reverses, causing a cascade of errors through subsequent layers. Fault-injection studies show that DNN resilience depends sharply on model, layer, structure, and bit position, so a small number of faults in vulnerable locations can cause disproportionate accuracy loss (Reagen et al. 2018). Unlike traditional software where a single bit error might cause a crash, in neural networks it can silently corrupt the learned representations that determine system behavior.

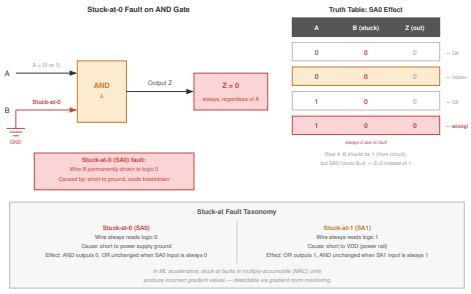
Reliability remains a design pressure as accelerator devices scale. Smaller geometries and lower operating voltages can reduce the charge needed to disturb a stored value, increasing sensitivity to some soft-error mechanisms (Baumann 2005), while large ML fleets expose rare hardware faults often enough that software-level detection and recovery become part of the training system design (He et al. 2023). ML system architects must treat hardware as an **unreliable substrate**, where algorithmic fault tolerance (gradient checksums, weight replication, periodic production consistency checks) is a mandatory requirement rather than an HPC specialty.

The temporal signature of a hardware fault determines that response. A one-time corruption asks for detection and rollback, a persistent defect asks for quarantine and replacement, and a recurring load-sensitive defect asks for evidence collection before the node poisons more jobs. Figure 7.12 summarizes the three categories that matter operationally.

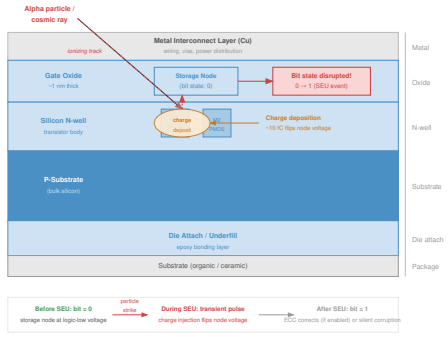
11 **IEEE 754**: The IEEE 754 floating-point standard defines the binary32 format with 1 sign bit, 8 exponent bits, and 23 fraction bits (IEEE Standards Association 2019). The bit layout creates an asymmetric vulnerability for ML: a sign-bit flip inverts a weight entirely ($0.5 \rightarrow -0.5$), while an exponent-bit flip can shift magnitude by orders of magnitude, so resilience mechanisms often prioritize the most sensitive bit positions rather than treating all bits as equally important.



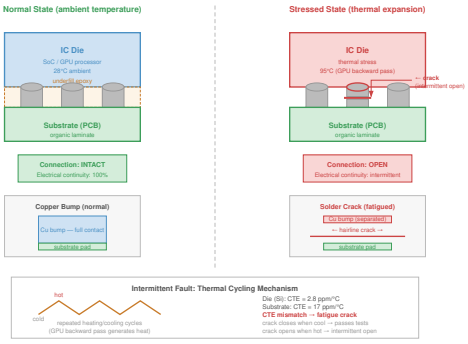
(a) Transient Fault (Bit Flip)



(b) Permanent Fault (Stuck-at)



(c) Transient Timing



(d) Intermittent Timing

Figure 7.12: Temporal Taxonomy of Hardware Faults: Four panels show how the three temporal categories manifest in practice. Transient panels (bit flip and timing glitch) show temporary disruptions that do not cause permanent damage. The Intermittent panel shows sporadic errors driven by unstable conditions or aging. The Permanent panel (stuck-at-0/1) shows irreversible physical damage that requires replacement.

The three categories differ by what the recovery system should infer:

- Transient faults are temporary disruptions caused by external factors such as cosmic rays or electromagnetic interference (Ziegler et al. 1996). Their danger is silent corruption: a rank may keep participating while sending a wrong gradient or activation.
- Permanent faults represent irreversible damage from physical defects or component wear-out, such as stuck-at faults or device failures that require hardware replacement. Their danger is repeatability: retrying the same device reproduces the same bad computation or hard failure.
- Intermittent faults appear and disappear sporadically due to unstable conditions like loose connections, aging components, or thermal stress. Their danger is ambiguity: the job may pass validation during one run and fail under a slightly different load.

The recovery strategy changes because each category fails on a different time scale and leaves different evidence behind.

7.2.2 Transient faults

The failure taxonomy classified transients by recovery posture, asking whether the system could retry or validate the affected work. This pass asks a different question: the physical mechanism that produces them and the detection it demands. Transient faults are the most common category, and figure 7.13 illustrates the basic mechanism: a bit-flip error occurs when a single bit in memory unexpectedly changes state, potentially altering critical data or computations in ways that cascade through neural network layers.

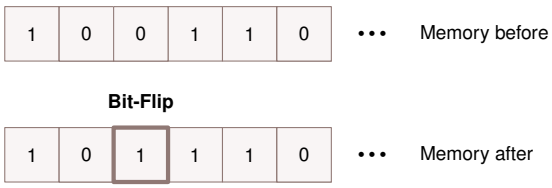


Figure 7.13: Bit-Flip Error: Transient faults can alter individual bits in memory, corrupting data or program instructions and potentially causing system malfunctions. These single-bit errors exemplify the vulnerability of hardware to transient faults like those induced by radiation or electromagnetic interference.

Transient faults matter because they can leave no damaged component to find after the incident. A Single Event Upset from radiation, a voltage fluctuation from an unstable power path (Reddi and Gupta 2013), electromagnetic interference, electrostatic discharge, crosstalk, ground bounce, a timing violation, or a soft error in combinational logic (Mukherjee et al. 2005) can all collapse to the same ML symptom: one tensor, packet, or instruction differs from what the collective expected. The recovery design therefore emphasizes online detection, correction where possible, and rollback from a known-good state rather than manual hardware replacement.

7.2.2.1 Quantitative fault rates

Advanced semiconductor processes can increase soft-error sensitivity as node capacitance, supply voltage, and stored charge shrink (Baumann 2005). For GPUs[12], the practical risk is amplified by massive parallelism: thousands of execution lanes and high-bandwidth memories create many sites where transient faults can affect weights, activations, or gradients. Operational MTBF[13] values are therefore workload-, component-, and environment-dependent rather than universal. For the checkpointing analysis below, the important systems rule is compounding: a cluster of 1,000 accelerators with an illustrative per-accelerator MTBF of 50,000 hours experiences an expected failure every 50 hours, necessitating robust checkpointing.

Memory subsystems are the most vulnerability-prone components, and fault tolerance mechanisms impose a direct bandwidth tax. Table 7.4 quantifies this cost across memory technologies:

The memory bandwidth protection analysis shows the throughput tax that error protection can impose on different memory technologies.

12 | **GPU Fault Rates**: GPUs concentrate thousands of execution lanes and high-bandwidth memory stacks in a single package. ML accelerators also run sustained high-power kernels with large volumes of weight, activation, and gradient data in flight, so transient faults can affect both computation and communication state.

13 | **MTBF (Mean Time Between Failures)**: Formalized by the U.S. military in MIL-HDBK-217 (1965), MTBF assumes exponential failure distributions during the useful-life period. For ML training, MTBF feeds directly into the Young-Daly formula: a cluster with 50,000-hour per-device MTBF and 1,000 devices has a system MTBF of 50 hours, demanding checkpoints every 1–2 hours to keep wasted compute below 1 percent of total training time.

Table 7.4: Memory Bandwidth Protection Analysis: Impact of ECC protection on effective memory bandwidth across different memory technologies used in ML accelerators. The bandwidth overhead directly affects training throughput for memory-bound workloads.

Memory Technology	Base Bandwidth	ECC Overhead	Effective Bandwidth
DDR4-3200	51.2 GB/s	12.5%	44.8 GB/s
HBM2	900 GB/s	12.5%	787.5 GB/s
HBM3	1600 GB/s	12.5%	1400 GB/s
GDDR6X	760 GB/s	Typically none	760 GB/s

The bandwidth table should not be read as a universal ranking of memory error rates. HBM, GDDR, and DDR reliability depend on device generation, operating temperature, protection scheme, and how errors are counted. The durable systems lesson is narrower: error protection consumes bandwidth and capacity, while fleet studies show that memory errors occur often enough to justify ECC, scrubbing, and monitoring in large deployments (Schroeder et al. 2009; Sridharan et al. 2015; Dixit et al. 2021). Background memory scrubbing (periodic reads and rewrites to detect accumulating soft errors) is usually engineered so that the bandwidth tax is small compared with foreground training traffic.

7.2.2.2 Transient fault impact on ML

Figure 7.14 shows the same charge-disturbance mechanism the failure taxonomy introduced (section 7.1.5.3), now at the device level: a cosmic ray strikes a memory cell or transistor and the induced charge alters stored or transmitted data. What this pass adds is the downstream effect on the model rather than the physics.

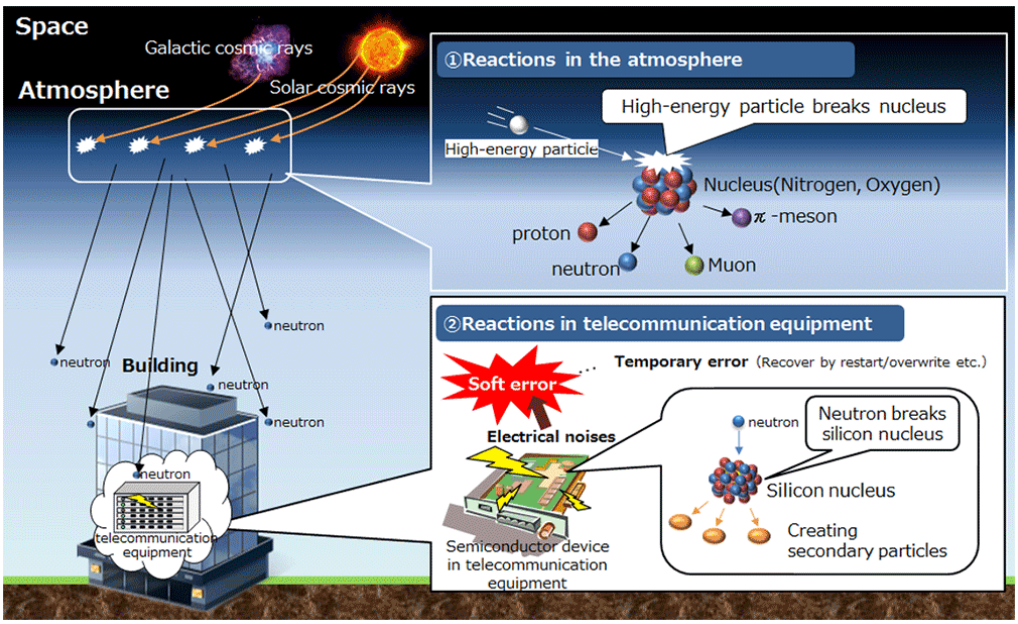


Figure 7.14: Transient Fault Mechanism: Cosmic rays and electromagnetic interference induce bit flips within hardware by altering electrical charges in memory cells and transistors, potentially corrupting data and causing system errors. Understanding these fault sources is critical for building robust AI systems that can tolerate unpredictable hardware behavior. Source: NTT.

During training, transient faults in the memory storing model weights or gradients can lead to incorrect updates that compromise convergence and accuracy (He et al. 2023). During inference, a bit flip in the activation values of a neural network can alter the final classification or regression output (Mahmoud et al. 2020). In safety-critical applications, these faults can result in incorrect decisions

that compromise safety (Li et al. 2017; Jha et al. 2019; Wan et al. 2021). Resource-constrained environments amplify these vulnerabilities: Binarized Neural Networks (BNNs) (Courbariaux et al. 2016), which represent weights in single-bit precision, suffer performance degradation from 98 percent to 70 percent test accuracy when random bit-flipping soft errors are inserted with 10 percent probability (Aygun et al. 2021). In distributed training, network partitions[14,15] can isolate ranks, and in synchronous training, even a single partitioned rank blocks the entire AllReduce collective.

7.2.3 Permanent faults

Permanent faults are irreversible hardware defects that persist until the faulty component is repaired or replaced. The operational clue is repeatability: the same accelerator, memory cell, link, or storage device fails again after retry, often at the same address, path, or workload phase. Manufacturing defects (improper etching, incorrect doping, contamination) and wear-out mechanisms (electromigration[16], oxide breakdown[17], thermal stress[18]) can all produce this signature. The most common abstract model is the stuck-at fault (Seong et al. 2010), where a signal or memory cell becomes permanently fixed at 0 or 1 regardless of input.

The most consequential permanent faults in ML accelerators are those that corrupt arithmetic silently and repeatably. A stuck-at fault in a Tensor Core's multiply-accumulate datapath, or a defective cell in an HBM bank that stores weight shards, produces the same wrong value every time the same computation runs. In training, this means every forward pass through the affected matrix produces a deterministically biased result, and every backward pass accumulates a skewed gradient. Because the error is reproducible but numerically small, training loss may decline normally for hundreds of steps before the accumulated bias manifests as gradient divergence or unexpectedly poor validation accuracy. In inference, the same defect deterministically skews specific output logits—a safety-critical property in medical or autonomous-driving deployments, where the fault does not crash the system but systematically tilts every decision involving the affected weight tile.

The Intel Pentium FDIV bug, discovered in 1994, provides the canonical illustration of this failure mode in a general-purpose processor. An error in the lookup table used by the Pentium processor's division unit caused incorrect results for specific operations (figure 7.15). The error was small—a mistake in the fifth digit—but it compounded across operations, corrupting precision-critical applications. The ML accelerator case differs in geometry but not in principle: where the FDIV bug corrupted scalar divisions, a stuck-at fault in a Tensor Core corrupts specific rows or columns of every matrix-multiply output that routes through the defective lane, affecting all feature maps or gradient shards that touch that partition of the computation. In safety-sensitive applications, the persistent arithmetic error becomes a safety hazard because every downstream decision inherits the biased computation.

Figure 7.16 visualizes how stuck-at faults propagate through logic gates and interconnects, causing incorrect computations or persistent data corruption that affects downstream model behavior.

For ML systems, the recovery decision is to stop trusting the component. A permanent accelerator datapath fault can keep producing bad gradients until the device is drained from the cluster (He et al. 2023; J. J. Zhang et al. 2018), while a permanent storage fault can compromise both the training dataset and the checkpoints needed for recovery. Checksum validation, replicated storage, hardware redundancy, error-correcting codes (Kim et al. 2015), and checkpoint-restart recovery[19] (Egwutuoha et al. 2013) work together because each mechanism helps identify the durable copy that can still be trusted. The Young-Daly formula introduced in section 7.1.2 then gives the economic boundary. Hardware hardening increases MTBF, but the square-root relationship means reliability investment must be balanced against faster checkpointing and restart infrastructure.

7.2.4 Intermittent faults

Intermittent faults are the hardest category because they create evidence, then disappear. A node may pass a reboot test, rejoin the fleet, and fail again only when the next training job drives the package into the same thermal, voltage, or communication regime. Physical degradation (cracks in solder joints, aging ball grid arrays, residue-induced electrical connections) creates those load-dependent conditions (figure 7.17) (Constantinescu 2008; Rashid et al. 2015). Voltage-underscaling studies such as ThUnderVolt show a separate timing-error route: reduced voltage margins can make

14 **Network Partition**: A network partition leaves one subset of workers unable to communicate with another. In synchronous training, even a single partitioned rank blocks the entire AllReduce collective, making partition tolerance a prerequisite for training jobs that run long enough to encounter routine fabric or control-plane disruptions.

15 **InfiniBand Subnet Manager (SM)**: A centralized software entity that discovers all nodes and switches, assigns Local Identifiers (LIDs), and calculates routing tables. In a network partition, the SM's role is critical: it must rediscover the new topology and update routing before training can safely resume. If the SM itself is partitioned, the fabric can enter a "zombie" state where nodes are physically connected but cannot route messages, a common cause of T_{detect} delays in large training runs.

16 **Electromigration**: Gradual displacement of metal atoms in conductors by electron momentum transfer, first characterized in early electromigration studies and later modeled in Black's reliability equation (Black 1969). Mean time to failure decreases with higher current density and higher temperature, so sustained high-power accelerator workloads make thermal management a direct determinant of fleet lifespan.

17 **Oxide Breakdown**: Irreversible gate oxide failure creating conductive paths through the transistor insulator. Gate oxide thickness shrank from roughly 100 nm in 1980s processes to nanometer-scale dimensions in FinFET-era devices, increasing susceptibility. Time-dependent dielectric breakdown (TDDB) constrains chip reliability projections, making oxide integrity a fleet-planning concern for ML accelerator deployments spanning 3–5 year hardware refresh cycles.

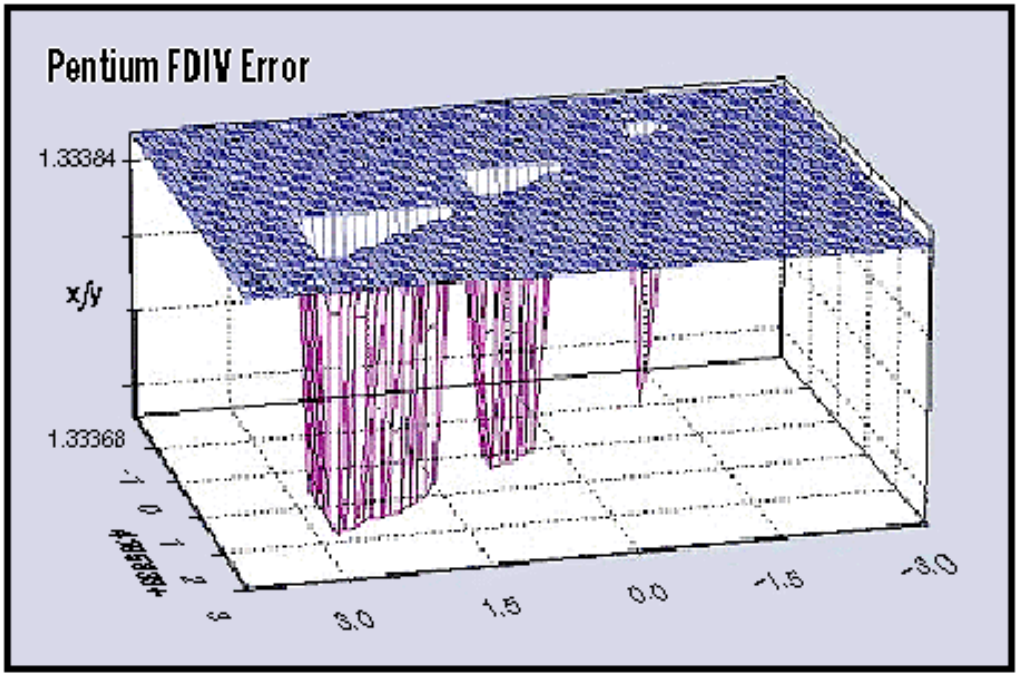


Figure 7.15: FDIV Error Regions: The downward spikes piercing the surface mark where the Pentium processor's faulty division unit produced incorrect results when calculating 4195835/3145727. Ideally, all values should round to 1.3338, but the bug dropped affected results from 1.33384 to 1.33368, an error in the fifth digit. Source: *Byte Magazine*.

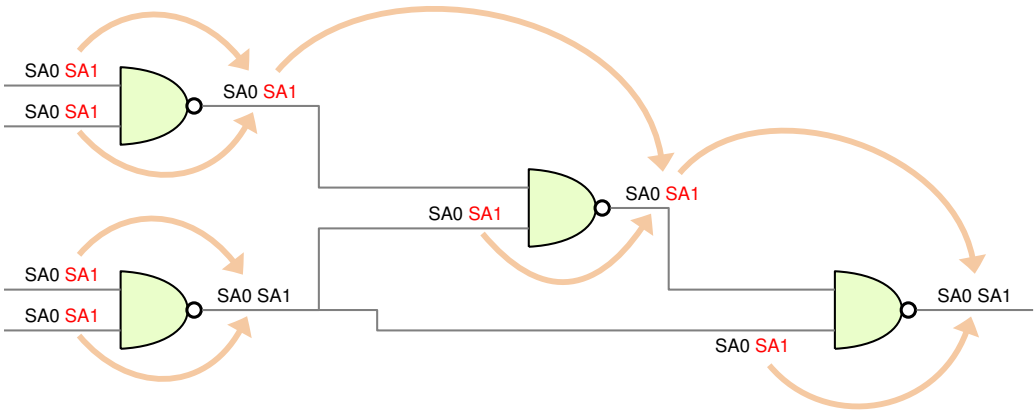


Figure 7.16: Stuck-at Fault Model: Digital circuits can experience permanent faults where a signal line becomes fixed at a logical 0 or 1, regardless of input; this figure represents a simplified depiction of a stuck-at-0 fault, where a signal is persistently low, potentially leading to incorrect computations or system failures. Source: accendo reliability.

signal propagation unreliable and cause incorrect computations that are difficult to reproduce (J. Zhang et al. 2018).

Figure 7.18 reveals how residue-induced intermittent faults in DRAM chips create unreliable electrical connections that lead to sporadic failures.

For ML systems, intermittent faults should be treated as suspect until enough evidence proves otherwise. Sporadic processing or memory errors can accumulate across iterations, degrading convergence without triggering explicit failures (He et al. 2023; Rashid et al. 2015). Runtime monitoring and anomaly detection provide the first hint, environmental controls reduce thermal and voltage triggers, and adaptive resource management can drain, downclock, or avoid a suspect

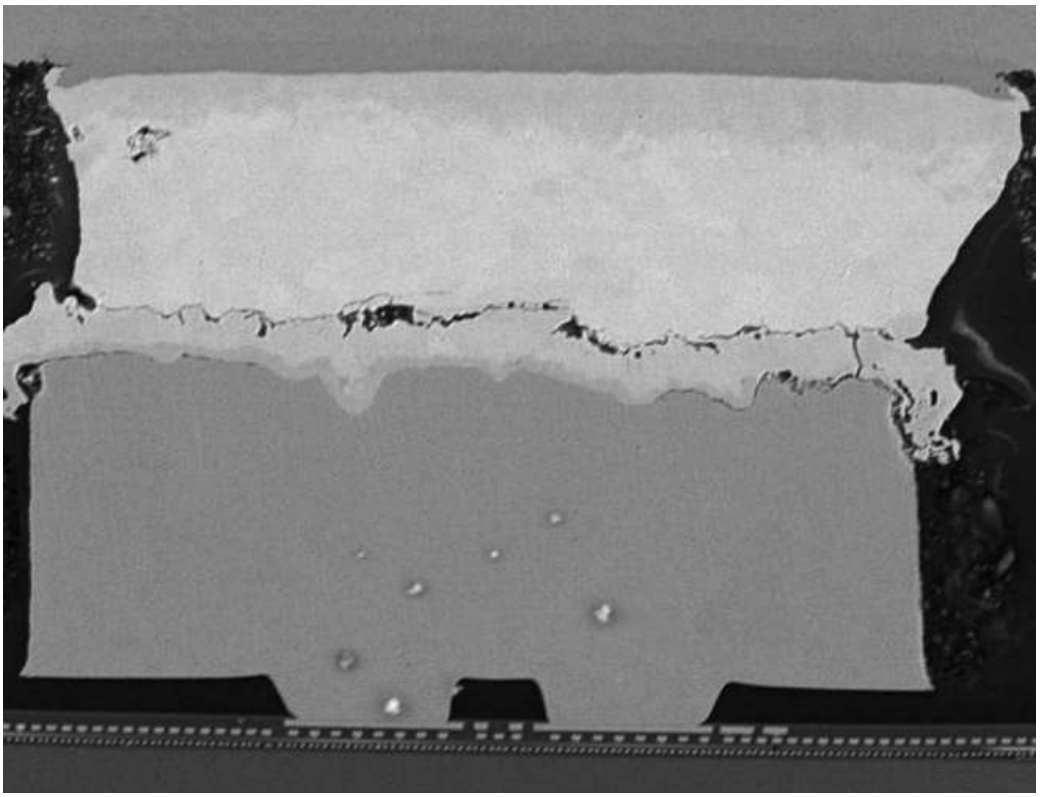


Figure 7.17: Intermittent Fault Mechanism: Increased resistance from cracks between copper bumps and package solder represents a common source of intermittent faults, disrupting signal transmission and potentially causing unpredictable system behavior. Microscopic material defects like these highlight the vulnerability of hardware to latent failures that are difficult to detect during testing but can manifest during operation. Source: constantinescu.

component while preserving job progress (Rashid et al. 2012). The goal is not merely to keep the node alive; it is to prevent a nondeterministic component from making validation and recovery untrustworthy.

7.2.5 Hardware fault detection and mitigation

Hardware fault mitigation works only when the detection mechanism matches the fault signature. Permanent defects are best exposed before deployment, transient bit flips need online correction, and intermittent or silent errors require runtime evidence. At the hardware level, two foundational mechanisms protect against the fault classes described earlier.

Built-in self-test (BIST) (Bushnell and Agrawal 2002) incorporates additional circuitry for self-testing using scan chains[20] that apply predefined test patterns to internal logic during system startup. BIST catches manufacturing defects and permanent faults before they corrupt production workloads.

Error detection and correction codes[21] (Hamming 1950) add redundant bits to detect and correct bit errors. Figure 7.19 illustrates the simplest form: parity checks append an extra bit to each data word, enabling immediate detection when a bit flip occurs. More advanced codes such as cyclic redundancy checks (CRC)[22] compute checksums that detect over 99.9 percent of transmission errors, a capability critical for validating gradient payloads during the distributed AllReduce operations covered in Chapter 6.

Hardware redundancy uses component duplication and voting to detect and mask faults (Sheaffer et al. 2007). Double modular redundancy (DMR)[23] duplicates computation and compares outputs at 100 percent silicon overhead; triple modular redundancy (TMR)[24] performs computation three times and takes a majority vote at 200 percent overhead, enabling automatic single-fault correction (Arifeen et al. 2020). Figure 7.21 shows how a TMR voter circuit masks a single faulty unit by selecting the

18 **Thermal Stress**: Degradation from repeated temperature cycling that cracks solder joints and degrades thermal interface materials. ML accelerators under sustained training loads can experience thermal throttling as clock speeds drop to prevent damage. The trade-off is direct: aggressive cooling (liquid, immersion) extends component lifespan and maintains training throughput but increases data center infrastructure cost, so cooling must be evaluated against both reliability and facility cost.

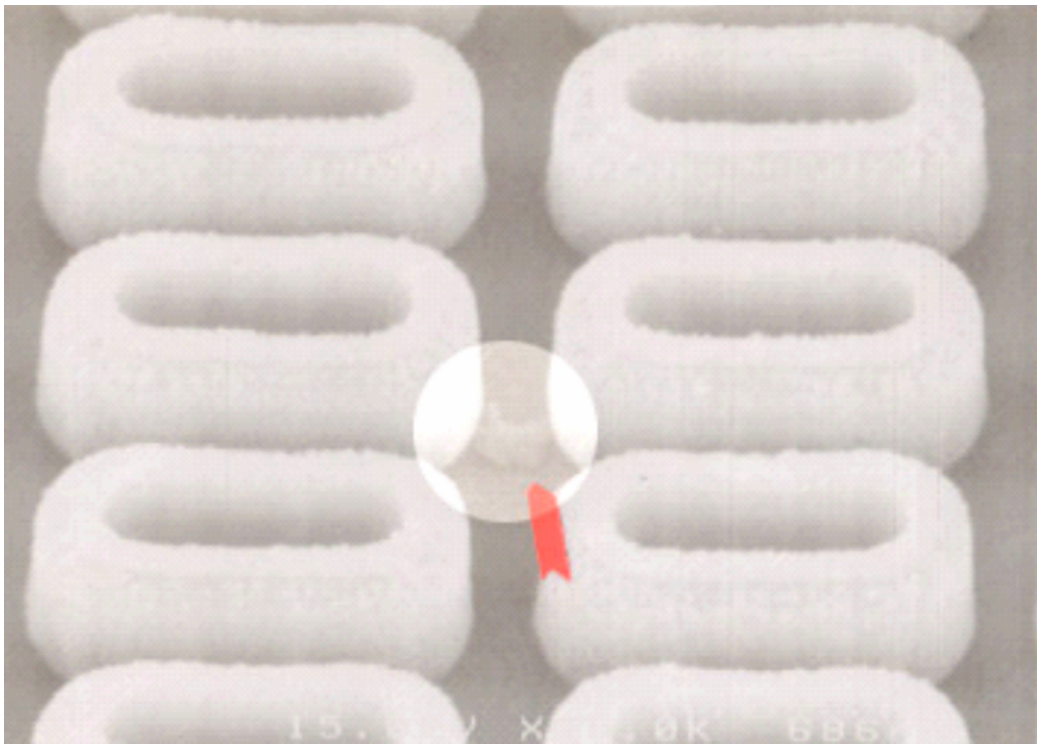


Figure 7.18: DRAM Residue Fault: Intermittent failures in DRAM chips commonly arise from microscopic residue accumulation and create unreliable electrical connections. Physical defects can induce sporadic errors and highlight the need for fault-tolerant system design and hardware testing. Source: Constantinescu.

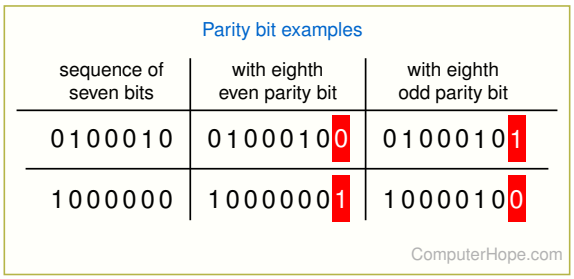


Figure 7.19: Parity Bit Error Detection: This figure provides a simple error detection scheme where an extra bit (the parity bit) ensures the total number of 1s in a data sequence is either even or odd. The second sequence includes a flipped bit, triggering the parity check and indicating a data corruption event during transmission or storage. Source: computer hope.

majority output. Tesla's Full Self-Driving computer uses DMR across two independent system on chip (SoC) units (figure 7.20), while the Boeing 777 uses TMR in its primary flight computer for safety-critical aviation control (Yeh 1996; Bannon et al. 2019).

At the software level, the same decision tree continues, with each fault signature routed to the one evidence stream that exposes it most cheaply. Silent statistical drift appears when gradients, losses, activations, or latencies no longer match the expected distribution, and runtime monitoring plus anomaly detection (statistical outlier tests, One-Class SVM) catches it at the cheapest layer because those checks ride on metrics the training loop already emits (Francalanza et al. 2017; Chandola et al. 2009). A corrupted-but-live state or checkpoint is caught by data-consistency checks that confirm it still corresponds to trusted data (Lindholm et al. 2019). A fail-stop node is separated from a merely slow one by heartbeat mechanisms (Kawazoe Aguilera et al. 1997), and a task that stalls without crashing is caught by watchdog timers (Pont and Ong 2002) that trigger recovery on lost progress.

19 **Checkpoint-Restart**: Originated in 1960s mainframe batch processing, where restarting multi-hour jobs from scratch was prohibitively expensive. Large distributed training jobs can checkpoint 100+ GB model states every 10–30 minutes; Google's TPUv4 resiliency study reports coordinated checkpointing and reconfiguration that kept wasted computation from node failures below 1 percent of total training time.

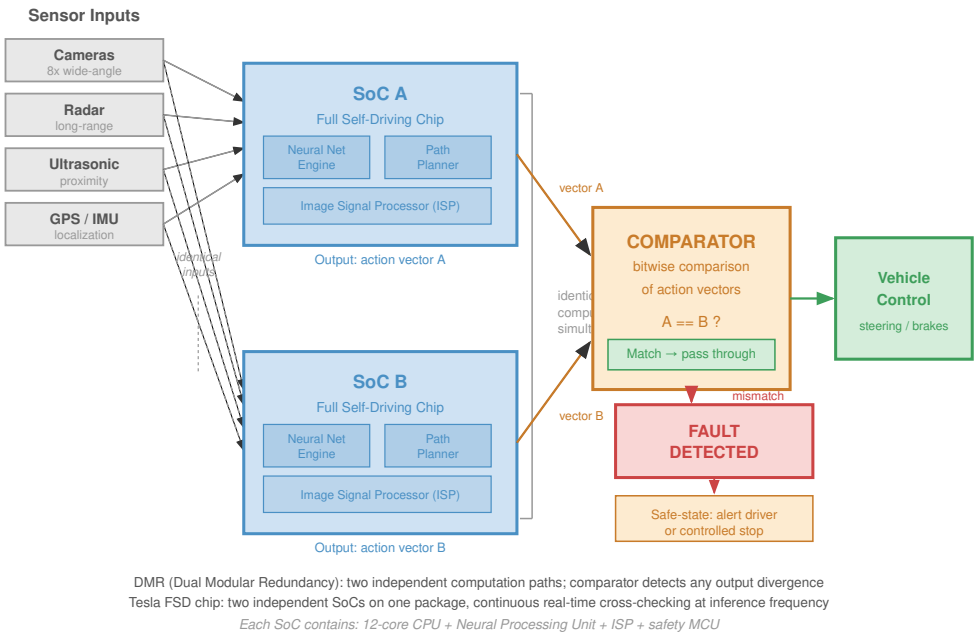


Figure 7.20: Dual Modular Redundancy (DMR): Tesla's full self-driving computer employs a DMR architecture, replicating critical computations across two independent system-on-chips (SoCs). A hardware comparator unit validates that both chips produce matching outputs before allowing a control command to reach the vehicle's actuators, ensuring that a single-chip hardware failure or bit flip cannot trigger dangerous driving behavior.

Only when none of these passive signals suffice, because the computation itself may be wrong with no statistical tell, does the system pay for software-implemented redundancy (SWIFT-style instruction duplication, N-version programming, Reed-Solomon reconstruction) (Reis et al. 2005; Avizienis et al. 2004; Plank 1997; Reed and Solomon 1960), whose duplicated or reconstructed computation is the most expensive layer and so is reserved for the highest-value state.

These mechanisms reduce the unreliable substrate risk, but they do not protect against faults introduced by the software stack itself. The next layer must reason about bugs that look like valid computation while corrupting the ML pipeline.

20 **Scan Chains**: Test infrastructure linking internal flip-flops into shift registers, developed in the 1970s for IC design-for-testability. The trade-off is concrete: 5–15 percent silicon area overhead buys 95 percent+ manufacturing fault coverage. For ML accelerators with billions of transistors in matrix-multiply units, scan-based testing during burn-in catches the stuck-at faults that would otherwise silently corrupt weight and gradient computations in production.

Checkpoint 7.3: Classifying a hardware fault

Verify your understanding of the transient, permanent, and intermittent fault categories and the fault, error, failure progression:

- ☐ **Transient fault**: Classify a GPU that produces a wrong result once after a cosmic-ray strike and then computes correctly for the rest of the run, and justify why error correction codes, not node replacement, match that fault.
- ☐ **Intermittent vs. permanent**: Distinguish an **intermittent** fault from a **permanent** one when both can produce a stuck-at value, including the diagnostic signal that separates them.
- ☐ **Fault-error-failure chain**: Trace a single bit flip through the fault, error, failure chain, identifying where error correction codes and checkpoint-restart each intervene.
- ☐ **Silent corruption**: Explain why silent corruption sits apart from the other fault manifestations even when its underlying fault is an ordinary transient bit flip.

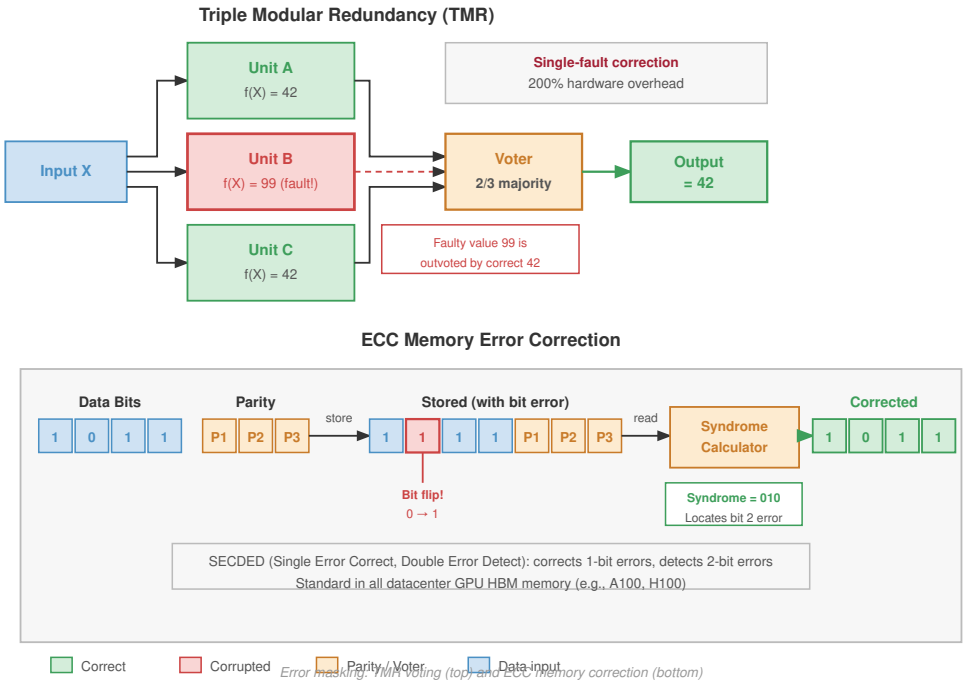


Figure 7.21: Error Masking via Voting: In triple modular redundancy (TMR), the same computation runs on three independent units. A voter circuit takes a majority vote of the results. Even if one unit suffers a fault (for example, bit flip), the system 'masks' the error by selecting the matching outputs from the other two units, allowing continuous operation without interruption.

7.3 Software Faults

A team spends three months testing an LLM against complex prompt-level failure cases, only to realize that its preprocessing script accidentally truncated all inputs at 512 tokens, silently discarding the system prompt entirely. In the pursuit of complex algorithmic robustness, engineers often overlook a common source of ML failure: mundane software bugs. In ML systems, a logic error in a data loader does not crash the pipeline; it subtly degrades the gradient, making software faults uniquely damaging.

Software faults require a different recovery lens from hardware failures. Hardware faults enter from silicon, electrons, and physical wear; software faults enter through design choices, dependency versions, and pipeline glue. The fault-tolerance problem shifts from replacing a failed component to identifying which valid-looking transformation corrupted data, gradients, or predictions.

The practical challenge is that software faults interact with every other system threat. A bug in data preprocessing can create distribution shifts, an implementation error in numerical computation can corrupt model behavior while preserving valid tensor shapes, and a race condition in distributed training can make different workers update from inconsistent state. These interactions arise because AI software stacks span frameworks, libraries, runtime environments, distributed systems, and deployment infrastructure. Each layer boundary creates an opportunity for faults to emerge and propagate, making software-level mitigation essential for production-scale reliability.

7.3.1 Software fault properties and propagation

Software faults in ML frameworks range from syntactic and logical errors to memory leaks,[25] concurrency bugs, and integration failures. They propagate across system boundaries: an error in a tensor allocation routine can cascade to disrupt training, inference, or evaluation in seemingly unrelated modules. Some faults are intermittent, manifesting only under specific conditions such as high system load, particular hardware configurations, or rare data inputs.

[21] **Hamming Codes (1950)**: Richard Hamming invented error-correcting codes at Bell Labs after repeated frustration with relay computer failures corrupting weekend batch jobs. His SECDED (single-error-correcting, double-error-detecting) scheme uses parity bits at power-of-2 positions to locate errors with $\mathcal{O}(\log n)$ overhead. ECC memory modules descend from this design, protecting the terabytes of model weights and optimizer state in ML training from the soft errors that would otherwise accumulate silently.

Resource mismanagement is a prominent failure class. GPU memory allocations accumulate across training iterations as intermediate activations, optimizer states, and gradient buffers are allocated but not released, until the allocator exhausts available capacity and raises an out-of-memory error mid-batch. A 70B parameter model in BF16 with AdamW optimizer states requires roughly 840 GB of GPU memory across a node, leaving virtually no headroom for unexpected buffer growth. Memory pressure builds gradually over hundreds of iterations, making root-cause attribution difficult without per-layer memory profiling.

Concurrency and synchronization errors constitute another recurring fault class. In distributed or multi-threaded environments, incorrect coordination among parallel processes leads to race conditions,[26] or inconsistent states.

Deadlocks[27] create a related synchronization failure, where workers wait indefinitely on resources or messages that never arrive. A bug in a high-level library might only manifest when paired with a specific version of a low-level numerical library such as cuDNN or MKL, requiring a holistic view of the system to identify root causes.

7.3.2 Software fault detection and prevention

Because many software faults leave the pipeline running while corrupting gradients, software fault mitigation strategies must assign each lifecycle phase a detection job. The prompt-truncation bug from the opening example should not survive until production monitoring: a unit test should catch the tokenizer boundary, an integration test should catch the batch shape and metadata path, a regression test should protect representative prompts, and runtime monitoring should detect any unexpected truncation distribution that escapes development. Table 7.5 organizes these gates by the kind of corruption they are designed to catch.

Table 7.5: Fault Mitigation Strategies: Software faults in ML systems require layered gates that catch different corruption modes: tokenizer boundary errors, batch-shape drift, unsafe slicing, dependency incompatibility, and runtime anomalies.

Category	Technique	Corruption caught	When to Apply
Testing and Validation	Unit testing, integration testing, regression testing	Tokenizer boundaries, batch shapes, golden-example regressions	During development
Static Analysis and Linting	Static analyzers, linters, code reviews	Unsafe slicing, implicit casts, unguarded shape assumptions	Before integration
Runtime Monitoring & Logging	Metric collection, error logging, profiling	Length-distribution shifts, NaN gradients, memory-growth anomalies	During training and deployment
Fault-Tolerant Design	Exception handling, modular architecture, checkpointing	Bad batches fail closed and recovery resumes from known-good state	Design and implementation phase
Update Management	Dependency auditing, test staging, version tracking	Tokenizer, data-collator, framework, and CUDA compatibility drift	Before system upgrades or deployment
Environment Isolation	Containerization (for example, Docker, Kubernetes), virtual environments	Environment-specific behavior and irreproducible dependency stacks	Development, testing, deployment
CI/CD and Automation	Automated test pipelines, monitoring hooks, deployment gates	Untested model, data, or preprocessing changes reaching production	Continuously throughout development

These safeguards matter only when each gate targets a concrete corruption path. A unit test that checks only whether the tokenizer runs is too weak; it must assert that system prompts, labels, masks, and sequence-length metadata survive preprocessing with the intended semantics. Integration tests then follow one batch through loading, sharding, collation, and model input construction, while regression tests preserve representative prompts and edge cases before they enter distributed execution (figure 7.22). The continuous integration/continuous deployment (CI/CD) structure in figure 7.23 is useful only when the release pipeline treats model, data, and preprocessing changes as ML artifacts that can corrupt the training objective; otherwise, it is just a generic software-delivery diagram. Automated gates make those checks release requirements instead of optional habits.

22 **CRC (Cyclic Redundancy Check)**: Polynomial checksum family introduced by Peterson and Brown (1961). CRC coverage depends on the polynomial, frame length, and error model; it is best understood as a low-cost detection layer rather than a universal guarantee. In distributed ML training, checksums or hashes can validate gradient payloads exchanged during collectives; without a verification layer, a corrupted gradient packet can silently poison the parameter update for every worker in the collective.

23 **DMR (Double Modular Redundancy)**: Duplicates computation and compares outputs to detect disagreements, at 100 percent silicon overhead vs. TMR's 200 percent. DMR detects mismatch but cannot choose the correct output by itself, so its coverage depends on the comparator, fault model, and safe fallback policy. Tesla's Full Self-Driving computer uses DMR across two independent SoCs, reflecting the design trade-off: DMR halves the hardware cost of TMR while requiring a safe fallback policy when outputs disagree (Bannon et al. 2019).

24 **TMR (Triple Modular Redundancy)**: Performs computation three times and takes a majority vote, enabling automatic single-fault correction at 200 percent hardware overhead. First applied in early fault-tolerant mainframes in the 1950s, TMR remains a longstanding pattern for radiation-exposed and safety-critical inference where single-fault correction matters more than hardware efficiency.

25 **Memory Leak**: A programming error where allocated memory is never released. In ML systems, GPU memory leaks are uniquely destructive because accelerator memory is scarce (40–80 GB per device) and shared across the entire training pipeline. A single leaked tensor per batch—perhaps from a debugging hook left in production—can exhaust GPU memory within hours, causing silent OOM failures that kill long-running training jobs without checkpointing.

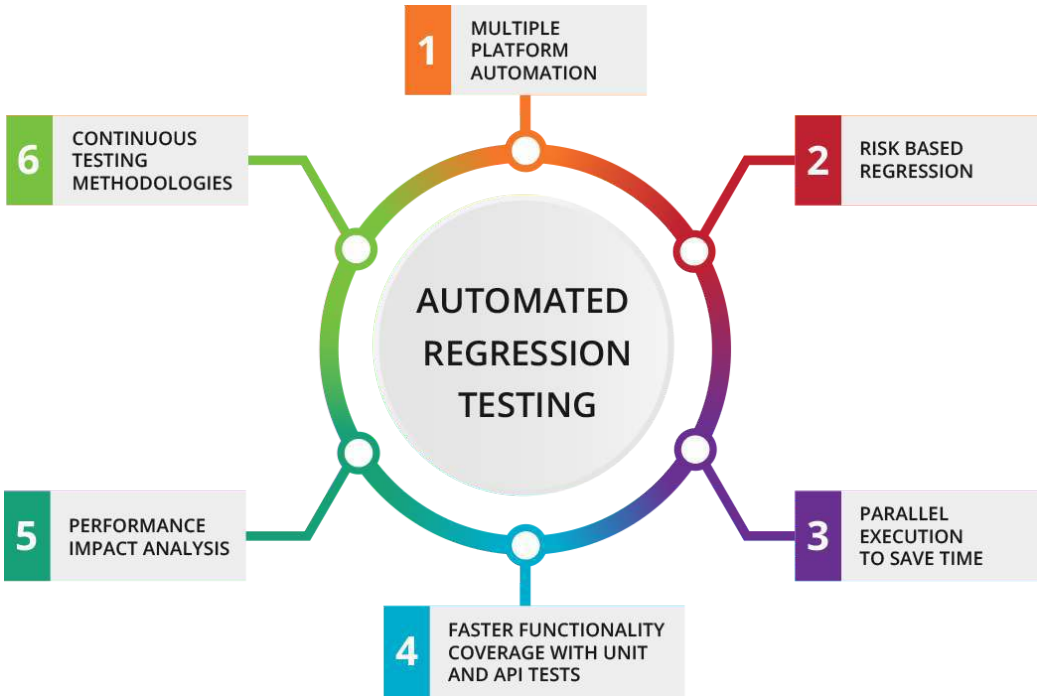


Figure 7.22: Automated Regression Testing: The recurring concerns of regression testing, spanning platform coverage, risk-based selection, parallel execution, unit and API coverage, performance impact, and continuous methodologies. For ML pipelines, these same gates must preserve tokenizer behavior, feature semantics, and batch construction across code changes. Source: UTOR.

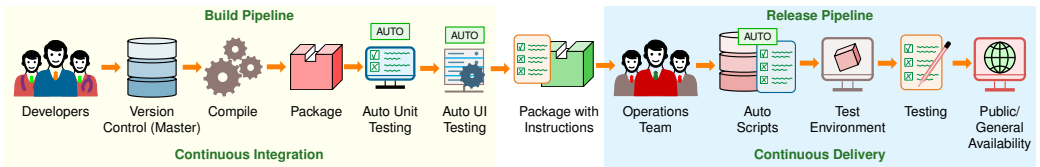


Figure 7.23: A CI/CD Pipeline: The standard software-delivery flow, from developer commit through version control, build, automated unit and UI testing, packaging, and release to general availability, grouped into a continuous-integration build pipeline and a continuous-delivery release pipeline. The accompanying text explains the additional gates, on model artifacts, data contracts, dependency versions, and preprocessing semantics, that this structure must add to become fault-tolerant for ML. Source: geeksforgeeks.

These software practices catch implementation faults that surface through tests or deployment gates. Silent corruption is harder because the system may continue running while producing wrong values, so the next layer is explicit verification.

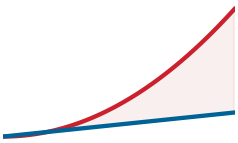
7.4 Check-and-Verify: Defending Against Silent Data Corruption

As clusters scale to 100,000+ GPUs, the probability of a "Silent Data Corruption" (SDC) event (in which an ALU or HBM bit flip occurs without triggering ECC or hardware alerts) approaches certainty during large collective operations. Standard AllReduce algorithms assume that if a node is *alive*, its data is *correct*. In the machine learning fleet, we must transition to a Byzantine fault tolerance mindset: "Trust, but verify."

Check-and-verify methods turn that mindset into a data-path invariant. The system computes a checksum, hash, or redundant reduction over each gradient shard or reduced buffer, compares the result across ranks or against an independently computed digest, and blocks the optimizer step when the verification disagrees. A quick estimate makes the exposure concrete.

26 **Race Condition**: A timing bug where system behavior depends on the uncontrolled sequence of concurrent events. In asynchronous distributed training, if multiple workers update the same parameter weight simultaneously without proper locking or versioning, updates can be lost or overwritten, leading to divergence or "ghost" weights that ruin convergence without triggering an error.

[27] **Deadlock**: A state where two or more processes are permanently blocked, each waiting for the other to release a resource. In pipelined training, a deadlock can occur if Stage 1 is waiting for a free buffer to send activations to Stage 2, while Stage 2 is waiting for Stage 1 to receive backward gradients, halting the entire fleet.



Silent data corruption accumulates until it becomes routine.

> **Napkin Math 7.2: The SDC certainty**
>
> **Problem**: Calculate the probability that at least one GPU in a 100,000-GPU fleet experiences a silent ALU error during a single 2 s training step.
>
> 1. **Fleet size**: 100,000 accelerators.
> 2. **Individual risk**: 10^{-6} per hour (a conservative estimate for SDC).
> 3. **The exposure**: In a 2-second window, the fleet has $100{,}000 \times (2/3600) \approx 56$ "GPU-hours" of exposure.
> 4. **The probability**: $\Pr(\text{at least one SDC}) \approx 0.0056$ percent.
>
> **Systems insight**: In a 100k-GPU fleet, a silent error occurs every 18,000 steps (roughly every 10 hours). Check-and-verify methods compute a CRC or hash for each gradient shard or reduced buffer, compare the digest across ranks or against a redundant reduction, and block the optimizer step when the digest disagrees. Without checksummed collectives or hash-and-verify gradients on the AllReduce path, model parameters silently accumulate corrupted contributions and drift within half a day. Robustness moves from being a *restart* problem to a *verification* problem: the fleet must perform redundant reductions or use parity-protected gradients to catch silent parameter corruption.

The calculation makes check-and-verify mandatory, but verification only earns trust when tested against realistic failures. The next step is therefore to inject faults deliberately and confirm that the same checks catch the corruption before it reaches model state.

7.5 Fault Injection Tools and Frameworks

Verification only has value when the injected failure resembles the production failure the system must survive. A multi-node training cluster that claims to tolerate network partitions should have that connection severed in a staging environment so engineers can observe what the orchestration layer does. **Fault injection** is the engineering discipline of deliberately perturbing a system to empirically verify that robustness mechanisms work before production traffic discovers the gap. ML/tensor/GPU fault-injection tools cover bit flips, tensor faults, and accelerator-oriented failures (Z. Chen et al. 2020; Tsai et al. 2021; Gräfe et al. 2023), while network partitions, latency injection, dependency termination, and API-response faults belong to chaos and distributed-systems testing practice (Basiri et al. 2016).

7.5.1 Fault and error models

That resemblance is a modeling decision, not an implementation detail. A transient bit flip, a permanent accelerator defect, and a corrupted gradient require different detection logic, so the experiment must specify duration, location, granularity, and propagation path before it begins.

These choices define the **fault model**: how a hardware fault manifests in the system. The corresponding **error model** represents how that fault propagates and affects the system's behavior.

The first modeling choice fixes the physical signature of the fault. Duration determines whether recovery should expect a transient event, a permanent defect, or an intermittent condition that is hard to reproduce. Location determines whether the injected fault enters through memory cells, functional units, or interconnects. Granularity determines whether the experiment studies a single bit, such as a bit flip, or a multi-bit burst error.

The error model carries that physical signature into the ML system. It describes how an initial hardware-level disturbance becomes corrupted weights, miscomputed activations, or higher-level logical errors in an ML framework. The abstraction level matters because each level exposes different propagation paths and masks different effects.

The choice of fault or error model is central to robustness evaluation. For example, a system built to study single-bit transient faults (Sangchoolie et al. 2017) will not offer meaningful insight into the effects of permanent multi-bit faults, since its design and assumptions are grounded in a different fault model entirely.

The implementation context of an error model also matters. A single-bit flip at the architectural register level, modeled using simulators like gem5 (Binkert et al. 2011), differs meaningfully from a similar bit flip in a PyTorch model's weight tensor. While both simulate value-level perturbations, the lower-level model captures microarchitectural effects that are often abstracted away in software frameworks.

Some fault behavior patterns transfer across abstraction levels, while others do not. Studies that compare single-bit and multi-bit fault models show that impact depends on where the fault lands and how it propagates, so robustness results from one injected-fault model should not be treated as universal (Sangchoolie et al. 2017; Papadimitriou and Gizopoulos 2021). Other important behaviors like error masking (Mohanram and Touba 2003) may only be observable at lower abstraction levels. This masking phenomenon can cause faults to be filtered out before they propagate to higher levels (figure 7.24) (Ko 2021), meaning software-based tools may miss these effects entirely.

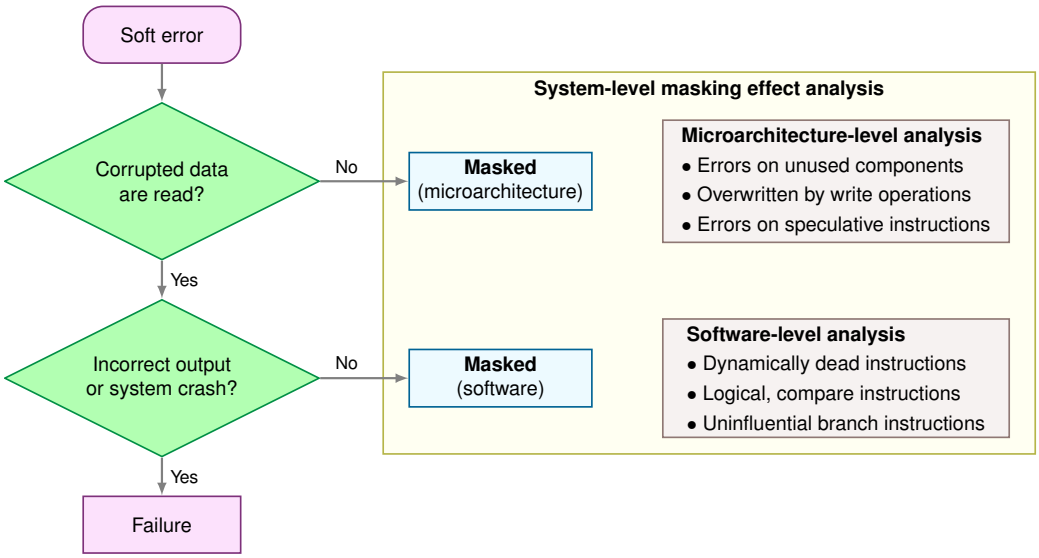


Figure 7.24: Error Masking: Microarchitectural redundancy can absorb single-bit faults before they propagate to observable system errors, highlighting a discrepancy between hardware-level and software-level fault models. This figure details how fault masking occurs within microarchitectural components, demonstrating that software-based error detection tools may underestimate the true resilience of a system to transient errors.

7.5.2 Fault injection methods

Fault injection methods trade realism against experimental scale. Hardware-based injection is the calibration point: it introduces faults into physical systems so software-level assumptions can be checked against real hardware behavior. Software-based injection is the scalable counterpart: it covers many model states quickly but must be calibrated against hardware behavior before its conclusions become production reliability claims.

7.5.2.1 Hardware-based fault injection

The method choice depends on whether the experiment needs bit-level targeting or physical radiation realism. FPGA-based fault injection uses field-programmable gate arrays (FPGAs)[28], reconfigurable integrated circuits that can be programmed to implement various hardware designs. By modifying the FPGA configuration, faults can be introduced at specific locations and times during the execution of an ML model.

Radiation or beam testing (Velazco et al. 2010) exposes hardware running ML models to high-energy particles like protons or neutrons. Specialized test facilities enable controlled radiation exposure to induce bitflips and other hardware-level faults, providing highly realistic fault scenarios that mirror conditions in radiation-rich environments.

28 **FPGA (Field-Programmable Gate Array)**: Reconfigurable hardware containing millions of programmable logic blocks. For fault injection, FPGAs provide bit-level targeting precision that software-based tools cannot match.

7.5.2.2 Software-based fault injection

Software-based fault injection trades physical realism for experimental scale. These tools simulate the effects of hardware faults by modifying a model's underlying computational graph, tensor values, or intermediate computations. They integrate directly with ML development pipelines, require no specialized hardware, and allow researchers to conduct large-scale fault injection experiments quickly and cost-effectively.

PyTorchFI (Mahmoud et al. 2020), a dedicated fault injection library for PyTorch developed in collaboration with Nvidia Research, is useful when the question is how tensor-visible perturbations affect model behavior. It injects faults into weights, activations, and gradients, showing that even simple bit-level faults can cause severe visual and classification errors, including the appearance of "phantom" objects where none exist.

7.5.2.3 Bridging hardware-software gap

The abstraction choice is the central risk in software-based fault injection. These tools offer speed and flexibility, but they do not always capture the full range of effects that hardware faults can impose on a system. The **abstraction gap** arises because software-based tools operate at a higher level and may overlook low-level hardware interactions or nuanced error propagation mechanisms.

Tools like **Fidelity** (He et al. 2020) address this gap by mapping low-level hardware error behavior to software-visible effects. By studying how faults originating in hardware move through architectural registers, memory hierarchies, and numerical operations, Fidelity makes software-injected faults resemble the way faults would manifest in a physical system.

Example 7.1: Software fault injection limits

Scenario: A team validates a training system with software fault injection and concludes that a particular tensor bit flip always corrupts model output.

Mechanism: Physical beam testing can produce a different result because hardware includes **masking effects** below the software abstraction. A bit flip in a hardware register might be corrected by ECC memory or masked by logical gates before it ever reaches the variable that software tools mutate directly.

Systems lesson: Software fault injection is useful for fast coverage of model-visible states, but it can overstate vulnerability when it bypasses circuit-level defenses. Production resilience claims therefore need a hardware-calibrated fault model, not software injection alone.

7.5.3 Hardware fault summary

While hardware and software faults represent distinct failure mechanisms, they ultimately manifest as system-level events that must be managed by the reliability logic established earlier in this chapter. The fault characteristics in table 7.6 close the loop from fault model to recovery choice. The table compares transient, permanent, and intermittent faults across duration, persistence, causes, and ML system impact so the detection and mitigation strategy matches the fault category being tested.

Table 7.6: Fault Characteristics: Transient, permanent, and intermittent faults differ by duration, persistence, and recurrence, impacting system reliability and requiring distinct mitigation strategies for robust AI deployments (Constantinescu 2008). Understanding these distinctions guides the design of fault-tolerant systems capable of handling diverse hardware failures during operation.

Dimension	Transient Faults	Permanent Faults	Intermittent Faults
Duration	Short-lived, temporary	Persistent, remains until repair or replacement	Sporadic, appears and disappears intermittently
Persistence	Disappears after the fault condition passes	Consistently present until addressed	Recurs irregularly, not always present
Causes	External factors (for example, electromagnetic interference or cosmic rays)	Hardware defects, physical damage, wear-out	Unstable hardware conditions, loose connections, aging components

Dimension	Transient Faults	Permanent Faults	Intermittent Faults
Manifestation	Bit flips, glitches, temporary data corruption	Stuck-at faults, broken components, complete device failures	Occasional bit flips, intermittent signal issues, sporadic malfunctions
Impact on ML systems	Introduces temporary errors or noise in computations	Causes consistent errors or failures, affecting reliability	Leads to sporadic and unpredictable errors, challenging to diagnose and mitigate
Detection	Error detection codes, comparison with expected values	Built-in self-tests, error detection codes, consistency checks	Monitoring for anomalies, analyzing error patterns and correlations
Mitigation	Error correction codes, redundancy, checkpoint and restart	Hardware repair or replacement, component redundancy, failover mechanisms	Robust design, environmental control, runtime monitoring, fault-tolerant techniques

7.6 Checkpointing: Preserving Progress

The practical question is how distributed training systems detect these faults and recover from them without losing days of computation. When a top-of-rack switch dies two weeks into a 175-billion parameter model training run, the cluster loses 64 GPUs instantly. The system cannot simply restart from scratch; the sunk cost is too high. The failure analysis in section 7.1 established that large-scale training systems will experience such failures frequently, requiring robust mechanisms to preserve and resume progress.

Training fault tolerance has three obligations: preserve state, detect and classify failures, and resume or resize the job. Checkpointing supplies the first obligation. The sections that follow then build the rest of the recovery model: failure detection decides what happened, recovery procedures restore a consistent process group, and elastic recovery decides whether the job can continue with a changed worker set.

Definition 7.1: Checkpointing

Checkpointing is the periodic serialization of the complete training state (parameters, optimizer state, and data loader position) to persistent storage.

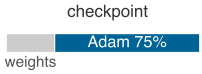
1. **Significance**: It minimizes lost work after a system failure by trading checkpoint I/O against rework. Writing too often wastes accelerator time on storage traffic; writing too rarely risks replaying many completed steps after a failure. The Young-Daly formula ($\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$) captures that local trade-off by choosing the interval that balances write cost against expected lost work.
2. **Distinction**: Unlike incremental backups, checkpointing must capture the exact execution context (including random seeds and learning rate schedules) to ensure deterministic resumption of the optimization loop.
3. **Common pitfall**: A frequent misconception is that checkpointing is "just writing to disk." In reality, for large models, it is a storage-system stress test: the simultaneous write from thousands of GPUs can trigger a checkpoint storm that saturates the entire network fabric (BW).

A training cluster that loses power without state preservation loses millions of dollars worth of gradient updates computed over the preceding weeks. The defense is to periodically write the model state to durable storage. Checkpointing captures sufficient state to resume training from a recorded point: model parameters, optimizer state,[29] training progress indicators, and random state for reproducibility.

7.6.1 Checkpoint interval from failure analysis

The Young-Daly formula stated in section 7.1.2 gives the optimal checkpoint interval, $\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}}$, but it cannot be applied until both inputs are known. The failure analysis

[29] **Adam Optimizer State**: Adaptive Moment Estimation (Adam) maintains per-parameter first-moment (m) and second-moment (v) estimates, tripling memory requirements compared to vanilla SGD (3×: parameters + two state vectors). For a 175B parameter model, optimizer state alone reaches 2.1 TB, which typically dominates checkpoint size and recovery time, making optimizer state the primary bottleneck in checkpoint I/O design.



Optimizer state, not weights, dominates checkpoint size.

earlier in this chapter supplies the system MTBF term, and the checkpoint payload established above supplies T_{write}. With both in hand, the formula stops being an abstract law and becomes a concrete schedule for the chapter's own cluster.

Figure 7.25 plots that trade-off for an illustrative 175B-parameter cluster: save overhead falls as intervals lengthen while expected rework rises, and the Young-Daly interval sits at the minimum of their sum. Its round inputs convey the curve's shape; the worked calculation below pins the exact interval from this chapter's own computed MTBF and write time.

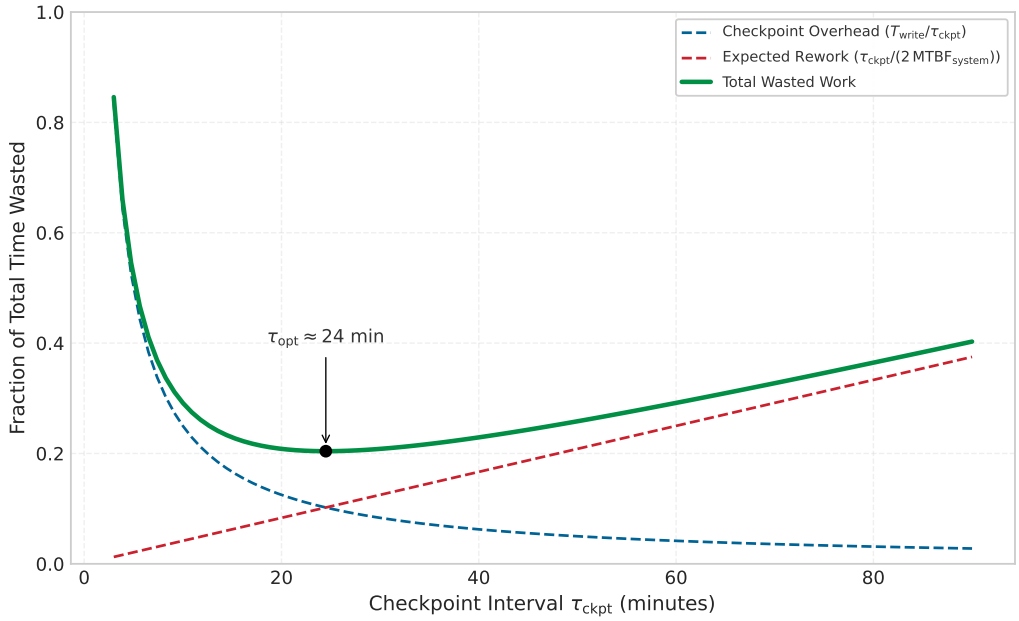


Figure 7.25: The Checkpoint Trade-Off: Checkpoint save overhead decreases with longer intervals, but wasted computation from failures increases. The Young-Daly optimal interval minimizes total overhead. For an illustrative 175B-model cluster (~2.5-minute checkpoint writes, 2-hour MTBF), the optimal interval is approximately 24 minutes; the worked calculation below derives the exact interval for this chapter's 10,000-GPU cluster.

The chapter's own 10,000-GPU cluster supplies both inputs, replacing the round figures in figure 7.25 with computed values. The worked cascade (section 7.1.4) put system MTBF at 3.69 hours, and the checkpoint payload above writes in 37 s (a 3.7 TB checkpoint at 100 GB/s). Substituting these into the Young-Daly formula gives the canonical interval for this cluster.

Napkin Math 7.3: The Young-Daly optimal interval

Problem: A 10,000-GPU cluster has an MTBF of 3.69 hours. A full model checkpoint takes 37 s to write. What is the optimal checkpoint frequency?

Math: Apply the Young-Daly formula: $\tau_{\text{opt}} = \sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$

1. **Convert to common units**: $\text{MTBF}_{\text{system}} = 3.69 \text{ hours} \times 3600 \text{ s/hour} = 13284 \text{ s}$.
2. **Substitute and solve**: With $T_{\text{write}} = 37$ s and MTBF $= 13284$ s, the formula gives $\tau_{\text{opt}} \approx 991.4$ s, or about 16.5 min.

Systems insight: At this interval, the "checkpoint tax" (time spent saving + time spent recomputing) is minimized to approximately 7.5 percent. Checkpointing every hour instead would increase failure risk, wasting about 14.6 percent of the cluster's capacity. As clusters scale, the optimal interval must shrink to keep up with the falling MTBF.

This result demonstrates why failure analysis matters: without knowing the system MTBF, we cannot set checkpoint intervals rationally. With this interval, checkpoint overhead consumes approximately 7.5 percent of training time, but the estimate depends on assumptions that often fail in production.

Systems Perspective 7.3: Young-Daly formula assumptions

The Young-Daly formula provides valuable intuition but rests on assumptions that may not hold in practice:

- **Exponentially distributed failures**: Assumes a constant failure rate. Real systems exhibit "bathtub curve" behavior with higher rates during burn-in and wear-out phases.
- **Deterministic checkpoint time**: Assumes T_{write} is constant. In practice, checkpoint time varies 2–3× due to storage contention, network congestion, and memory pressure.
- **Recovery time equals checkpoint time**: Assumes recovery reads the same data written during checkpoint. Often recovery takes 3–5× longer due to job scheduling delays, topology reconstruction, and warmup.
- **Single failure mode**: Assumes one failure at a time. Correlated failures (power, cooling, shared switch) violate this assumption.
- **Infinite timeline**: Optimizes for long training runs. Short runs, where total time is comparable to MTBF, require different analysis.

When assumptions are violated, the optimal interval may shift significantly, but restart overhead should not be treated as if it were paid at every checkpoint. In the first-order lost-time model, restart time adds a per-failure recovery term to total expected waste; it does not replace T_{write} inside $\sqrt{2 \cdot T_{\text{write}} \cdot \text{MTBF}_{\text{system}}}$ unless a recovery-aware Daly-style model is explicitly derived and calibrated.

Figure 7.26 makes the temporal cost of failures concrete by showing the sequence of events during a training run: productive computation, periodic checkpoints, a failure event, and the recovery process with its associated wasted work.

The timeline in figure 7.26 reveals why T_{restart} matters as much as T_{write}: the total failure cost is the sum of lost work (bounded by τ_{opt}) and recovery time, which includes job scheduling, checkpoint loading, and pipeline warmup. Production systems where T_{restart} exceeds T_{write} by 3–5× should include recovery time in their total-waste and SLO budgets, and should use a cited recovery-aware checkpoint model rather than folding restart time into the per-checkpoint write term.

7.6.1.1 Checkpoint overhead analysis

Beyond the time consumed by checkpoint writes, checkpointing imposes additional overhead through memory consumption and training disruption. Checkpoint serialization requires memory buffers for gathering distributed state and preparing data for write. For synchronous checkpointing, all workers must hold their checkpoint data in memory until the checkpoint completes. This potentially requires significant additional memory allocation.

Synchronous checkpointing pauses training while the checkpoint writes. Even with fast storage, the pause disrupts the training pipeline and may cause GPU idle time. Data loading and forward passes cannot proceed during checkpoint operations.

Equation 7.8 quantifies the wasted time due to checkpointing:

$$f_{\text{ckpt}} = \frac{T_{\text{write}}}{\tau_{\text{ckpt}}} + \frac{T_{\text{pause}}}{\tau_{\text{ckpt}}} \tag{7.8}$$

Here, f_{ckpt} is the dimensionless fraction of time lost to checkpointing, T_{write} is checkpoint write time, τ_{ckpt} is the checkpoint interval, and T_{pause} is any training pause beyond the checkpoint write itself. This includes memory allocation, coordination, and serialization.

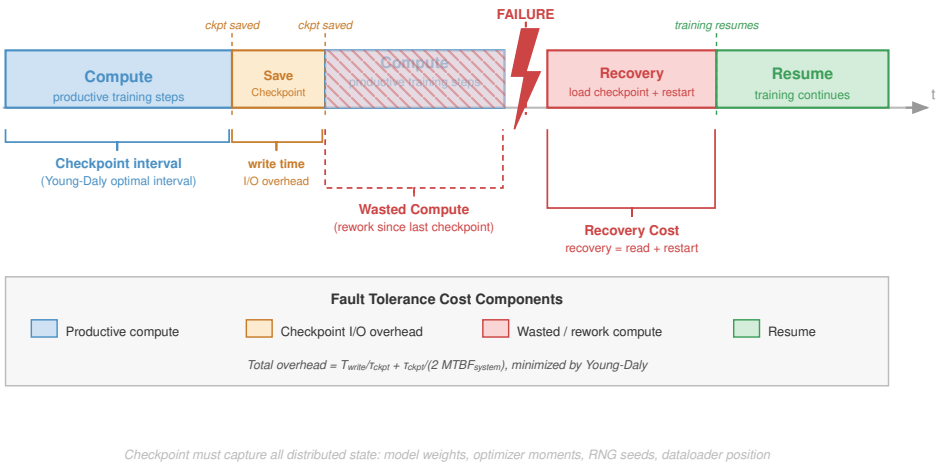


Figure 7.26: Checkpoint-Recovery Timeline: A training run proceeds through alternating phases of computation (green) and checkpoint writes (blue). When a failure occurs (red lightning bolt), all work since the last completed checkpoint is lost (hatched gray). Recovery involves job restart overhead, checkpoint loading, and pipeline warmup before productive training resumes. The total cost of a failure includes both the lost work and the recovery latency.

7.6.1.2 The "stop-the-world" cost

The financial impact of synchronous checkpointing at scale is severe. When a 10,000-GPU cluster pauses for the 2 minutes a multi-TB checkpoint typically takes on shared storage, it burns $10,000 \times \frac{2}{60} \approx 333$ idle GPU-hours, which at roughly $3 per GPU-hour costs about $1,000 in wasted compute for a single checkpoint. If checkpoints occur hourly, that idle time compounds to $24,000 per day spent waiting for storage I/O. This economic reality drives the aggressive adoption of asynchronous checkpointing strategies that move data movement off the critical path. Table 7.7 reveals how checkpoint characteristics vary dramatically by model architecture: larger models require longer write times but also benefit from correspondingly longer optimal intervals.

The Archetype A row below uses the mixed-precision optimizer footprint (2.1 TB), not the full FP32 training-state total (3.7 TB) cited in the introduction.

Table 7.7: Checkpoint Overhead by Model Type: Checkpoint sizes use the mixed-precision optimizer footprint for large LLMs (see Archetype A note above), not the 3.7 TB full FP32 training-state total from the introduction. Larger models require longer save times but also benefit from longer optimal checkpoint intervals. The percentage column reports save-only checkpoint write tax, not total Young-Daly waste; at the optimum, expected rework adds a comparable term.

Model Type	Mixed-Precision Checkpoint Size	Write Time (100 GB/s)	Optimal Interval (5 h GPU-only MTBF)	Save Overhead
Archetype A (175B dense LLM)	2.1 TB	21 s	14.5 min	2.4%
20B dense trans-former class	240 GB	2.4 s	4.9 min	0.8%

Model Type	Mixed-Precision Checkpoint Size	Write Time (100 GB/s)	Optimal Interval (5 h GPU-only MTBF)	Save Overhead
BERT-Large	1.4 GB	0.014 s	22 s	0.06%
Archetype B (DLRM at Scale)	4 TB	40 s	20 min	3.3%
ResNet-50	102.4 MB	0.001 s	6 s	0.02%
Vision trans-former class	1.2 GB	0.012 s	21 s	0.06%

While the theoretical overhead of checkpointing appears manageable for individual models, writing these massive state files introduces a new bottleneck. When thousands of GPUs simultaneously attempt to write gigabytes of data to a shared file system, the resulting I/O congestion threatens to bring the entire cluster to a halt. The scale of that I/O burst is easiest to see by expanding the example from one model to thousands of simultaneous writers.

Example 7.2: The checkpoint storm

Scenario: A training job saves a distributed checkpoint from thousands of accelerators at the same instant.

Mechanism: For the 2.1 TB mixed-precision optimizer footprint in the Archetype A row above—not the 3.7 TB full FP32 total—1,000 workers would each write roughly 2.1 GB of state. The same mechanism becomes catastrophic in embedding-heavy recommendation or trillion-parameter mixture-of-experts jobs: 10,000 GPUs, each holding a 10 GB shard of model or embedding state, can flood the network fabric with 100 TB of simultaneous writes.

Systems lesson: A checkpoint storm occurs when massive parallel writes overwhelm the cluster's I/O infrastructure, causing switch buffers to overflow and storage controllers to lock up.

While table 7.7 suggests modest overhead percentages, real deployments often encounter checkpoint times far exceeding these theoretical estimates. Diagnosing such discrepancies requires examining the full system stack.

Example 7.3: Debugging checkpoint overhead

Problem: A team training a 70B parameter model observes that checkpointing takes 10 minutes per checkpoint, far exceeding their expected 2 minutes target. Training throughput has dropped 30 percent because the cluster sits idle during checkpoints. Diagnose the bottleneck and identify the recovery path.

Diagnosis: The fleet stack exposes where the checkpoint overhead comes from.

The fleet stack framework in figure 1.13 structures our investigation across three layers.

Analysis: The infrastructure layer reveals the hardware constraints.

The diagnosis begins at the bottom of the stack:

- **Model state size**: 70B parameters $\times$ 2 bytes (BF16) = 140 GB for weights, plus 560 GB for FP32 Adam optimizer states (first and second moments), totaling approximately 700 GB per checkpoint before gradients, master weights, metadata, or framework overhead
- **Storage system**: Shared NFS filer with 10 Gbps network attachment
- **Theoretical bandwidth**: 10 Gbps = 1.25 GB/s maximum throughput
- **Minimum write time**: 700 GB / 1.25 GB/s = 560 s (9.3 min)

The infrastructure layer reveals our first insight: even with perfect efficiency, the NFS bandwidth cannot achieve a 2-minute checkpoint for this model size.

Analysis: The distribution layer shifts the diagnosis from raw bandwidth to checkpoint coordination.

- **Checkpoint mode**: Synchronous, all 512 GPUs pause and wait
- **Write pattern**: All 64 nodes writing simultaneously to shared NFS
- **Observed behavior**: writes that take 10 minutes instead of the 9.3 min theoretical minimum

The Distribution Layer reveals the shared-storage bottleneck. With 64 nodes competing for 10 Gbps, each node effectively receives only 20 MB/s, the 10 Gbps link split evenly across all 64 nodes. If the checkpoint is sharded evenly, each node writes about 10.9 GB, so the write still lands near 9.3 min. The observed 10 minutes is therefore close to the bandwidth floor, not something more simultaneous NFS writers can solve.

Analysis: The governance layer connects the bottleneck to the completion target.

- **SLA requirement**: Training must complete within 2 weeks
- **Current trajectory**: 30 percent overhead extends training to 2.6 weeks
- **Budget impact**: Extended training incurs additional $500K in compute costs

Fix: A layer-aware checkpoint path removes the critical-path storage bottleneck.

1. **Infrastructure layer**: Install local Non-Volatile Memory Express (NVMe) staging drives (3.5 GB/s per node).
2. **Distribution Layer**: Implement asynchronous checkpointing:
   - Phase 1: GPU $\rightarrow$ CPU memory copy (fast, 32 GB/s PCIe)
   - Phase 2: CPU $\rightarrow$ local NVMe (3.5 GB/s, training resumes)
   - Phase 3: Background NVMe $\rightarrow$ NFS copy (overlapped with training)
3. **Validation**: The critical-path data copy drops to approximately 0.34 s if GPU-to-CPU staging proceeds in parallel across 64 nodes (or 21.9 s if serialized through one 32 GB/s path), reducing training impact from 30 percent to under 1 percent once background NVMe and NFS copies are overlapped.

Systems lesson: Diagnosing distributed systems failures requires examining all layers. A purely algorithmic fix (Distribution Layer) would fail because the physical bandwidth was insufficient. A purely hardware fix (infrastructure layer) would be wasteful without understanding the coordination pattern. The solution required changes at multiple layers working in concert.

The interval calculation chooses *when* the system should preserve state. The next implementation question is *how* that state is written and recovered without turning the checkpoint itself into the critical path.

7.6.1.3 Synchronous vs. asynchronous checkpointing

The synchronous and asynchronous checkpointing approaches create different failure recovery trade-offs. **Synchronous checkpointing** guarantees a globally consistent state, with all workers at the same training step, simplifying recovery logic. All workers coordinate to reach a consistent state, write their portions, and resume training only after all writes complete.

Asynchronous checkpointing reduces training disruption but requires tracking which workers have completed which checkpoints, adding complexity to recovery coordination. Workers snapshot their state to CPU memory or staging storage, then continue training while a background process writes the snapshot to persistent storage.[30]

[30] **Asynchronous Checkpointing**: Pipelines the checkpoint write behind training computation by staging GPU state to CPU memory via CUDA streams, then writing to storage on a background thread. Implementations such as DeepSpeed can approach very low checkpoint overhead when sufficient CPU staging memory is available, decoupling the checkpoint I/O latency from the training critical path at the cost of higher peak host memory consumption.

7.6.1.4 Checkpoint storage and recovery

The tiered checkpoint storage architecture described in section 4.7, with local NVMe for speed, distributed filesystem for durability, and object storage for long-term retention, provides the storage foundation on which recovery mechanisms operate. The fault-tolerance question is how recovery mechanisms use that infrastructure rather than how the storage hierarchy is designed.

For fault tolerance, the critical concern is not where checkpoints are stored but how quickly they can be read during recovery. Recovery time depends on storage tier bandwidth: local NVMe enables fastest recovery (5–10 GB/s per node), distributed filesystems provide moderate speed with durability (50–200 GB/s aggregate), while object storage offers slowest recovery but highest durability for disaster recovery scenarios.

7.6.2 Checkpoint coordination patterns

Recovery from distributed checkpoints for sharded models requires understanding the coordination protocols that ensure checkpoint consistency. When training spans multiple workers, two primary approaches exist: centralized checkpointing where a coordinator gathers all state and writes a single checkpoint, and distributed checkpointing where each worker writes its own portion of the checkpoint.

7.6.2.1 Centralized checkpointing

In centralized checkpointing, workers send their state to a coordinator process that assembles and writes the complete checkpoint. This approach simplifies checkpoint management and produces self-contained checkpoint files, but every benefit is paid for at the coordinator: all state crosses the coordinator's network links, the coordinator needs memory for the entire checkpoint, and coordinator failure loses the checkpoint operation. The pattern works acceptably for tens of workers, where the management simplicity may outweigh the bottleneck, but it becomes impractical for hundreds or thousands of workers because the coordinator becomes both the throughput limiter and the single point of failure.

The decision boundary is the size of the state and the number of writers. Centralized checkpointing is attractive when operational simplicity matters more than aggregate bandwidth, but distributed and sharded approaches become necessary once the checkpoint itself is a fleet-scale object.

7.6.2.2 Distributed checkpointing

In distributed checkpointing, each worker writes its portion of the checkpoint to a shared filesystem or object storage, as figure 7.27 contrasts with the centralized approach. A coordinator signals when to checkpoint and confirms completion, but state flows directly from workers to storage without aggregation.

The coordination protocol proceeds in six steps:

1. Coordinator broadcasts checkpoint request with checkpoint ID
2. Each worker reaches a consistent state (barrier synchronization)
3. Each worker writes its shard to `checkpoint_<id>/worker_<rank>.pt`
4. Each worker confirms write completion to coordinator
5. Coordinator writes checkpoint metadata after all confirmations
6. Coordinator broadcasts checkpoint complete, training resumes

This protocol ensures that either all workers complete their writes or the checkpoint is incomplete. Valid checkpoints have complete metadata. Incomplete checkpoints have missing metadata and can be detected. Partial checkpoints can be garbage collected. In practice, that strictness determines whether the system can claim strict, bounded-asynchronous, or eventual consistency.

> Systems Perspective 7.4: Checkpoint consistency models
>
> The idealized protocol assumes step 2 completes quickly. At scale, this barrier synchronization becomes the dominant checkpoint cost because there is almost always at least one slow worker in a 10,000+ GPU cluster.

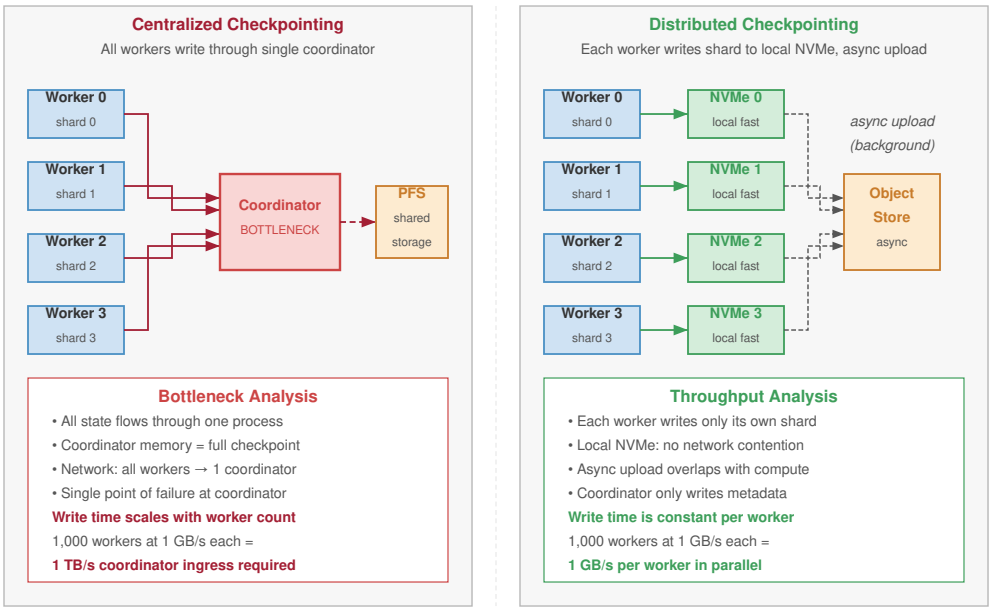


Figure 7.27: Distributed Checkpoint Architecture: Comparison of centralized vs. distributed patterns. (Top) Centralized aggregation creates bottlenecks at the coordinator. (Bottom) Distributed sharding enables every worker to write directly to the Parallel File System (PFS) in parallel, aggregating bandwidth across the storage fabric and minimizing the training pause.

Strict Synchronous: All workers checkpoint at exactly the same training step. Provides strongest consistency but highest overhead from barrier synchronization.

Bounded Asynchronous: Workers may be within k steps of each other (typically $1 \leq k \leq 3$). The checkpoint manager tracks the "checkpoint wavefront" across workers. Recovery uses the earliest consistent cut across all shards. This trades perfect consistency for dramatically reduced synchronization overhead and is what production systems actually use.

Eventual Consistency: Workers checkpoint when convenient, reconcile during recovery. Lowest overhead but requires complex recovery logic to reconstruct consistent state.

The basic protocol has a subtle correctness bug: if the coordinator crashes after some workers confirm but before writing metadata, those workers believe the checkpoint succeeded while the system has no valid checkpoint. Production systems use two-phase commit[31], a classic distributed systems protocol (Gray 1978), to make the checkpoint decision atomic:

In the prepare phase, workers write to a staging location and report success to the coordinator. In the commit phase, the coordinator atomically renames or commits all shards together, moving files from `staging/` to `checkpoints/`, for example. If the coordinator fails between phases, workers detect the failure during the next heartbeat and roll back staged writes, so either all shards commit together or none do. Consistency remains a separate requirement: synchronous gradient updates naturally create step boundaries where all workers have applied the same updates, but asynchronous training and pipeline parallelism require more careful coordination to define a consistent cut.

31 **Two-Phase Commit (2PC)**: Formalized by Gray (1978), 2PC ensures all participants either commit or abort atomically. The protocol's known weakness is blocking: if the coordinator fails between prepare and commit, participants wait indefinitely. For distributed checkpointing this blocking is acceptable because a stuck checkpoint can be timed out and retried, whereas the alternative (partial checkpoint writes) would leave the system with an unrecoverable inconsistent state.

7.6.2.3 Sharded checkpointing

Modern distributed training frameworks partition model state across workers using techniques like ZeRO (Zero Redundancy Optimizer) and FSDP (Fully Sharded Data Parallel). In these configurations, no single worker holds complete model state. Each worker holds only its assigned parameter shard plus corresponding optimizer state.

Sharded checkpointing[32] (Rajbhandari et al. 2020) uses this distribution: each worker writes only its shard, dramatically reducing per-worker write volume. Recovery loads shards and redistributes state to workers based on the recovery configuration.

This approach enables efficient checkpointing even for massive models. A 175B parameter model with a 3.7 TB checkpoint distributed across 1,024 workers requires each worker to write only 3.6 GB, achievable in seconds with local NVMe storage.

When recovering with a different number of workers than the checkpoint, shard redistribution must remap state to the new worker configuration. This occurs due to elastic scaling or hardware changes. Framework support for flexible resharding enables recovery even when the worker count changes. However, possessing a valid checkpoint is not sufficient on its own. Sharded checkpointing mitigates the I/O storm, but the system must still identify the failure, trigger restoration, and coordinate the distributed shards before training can resume. The speed of detection and recovery determines the true cost of the interruption.

32 **Sharded Checkpointing**: Each worker saves only its local partition rather than gathering state to a single writer. For ZeRO-3 or FSDP with 1,024 workers training a 175B model, each worker writes approximately 3.6 GB instead of one writer handling 3.7 TB, parallelizing I/O across all nodes. The trade-off: recovery requires all shards to be present and consistent, making the checkpoint protocol more complex and the failure of any single shard's storage fatal to the entire checkpoint.

Checkpoint 7.4: Reasoning about the checkpoint tax

Verify your understanding of how failure rate and checkpoint cost set the optimal checkpoint interval:

- ☐ **Young-Daly shift**: Use the Young-Daly formula to estimate how the optimal interval τ_{opt} shifts when a cluster doubles in size and its system MTBF halves, with T_{write} held fixed. State whether the interval lengthens or shortens and by about what factor.
- ☐ **Two-sided tax**: Explain why writing checkpoints too frequently and writing them too rarely both raise the **checkpoint tax**, including the two costs τ_{opt} balances.
- ☐ **Faster writes**: Determine how an order-of-magnitude reduction in T_{write} from sharded checkpointing changes the optimal interval and expected lost work per failure.
- ☐ **Square-root scaling**: Explain why the Young-Daly interval depends on the square root of MTBF rather than scaling linearly with it.

7.7 Failure Detection and Recovery

A GPU silently hangs, dropping its utilization to zero while its peer GPUs wait indefinitely at an AllReduce barrier. Every minute the system takes to notice this straggler and reboot the node costs thousands of dollars in idle cluster time. Checkpoints preserve state, but the recovery process itself determines how much compute a failure wastes.

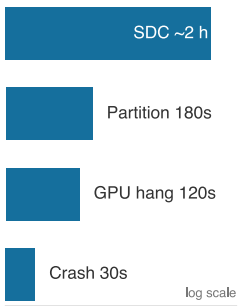
Equation 7.9 decomposes recovery time into four primary components:

$$T_{\text{recovery}} = T_{\text{detect}} + T_{\text{restart}} + T_{\text{load}} + T_{\text{warmup}} \tag{7.9}$$

where:

- T_{detect}: Time between the actual hardware fault and the system classifying it as a failure
- T_{restart}: Time for the job scheduler to allocate new resources and launch replacement processes
- T_{load}: I/O time to read checkpoint state from distributed storage into GPU memory
- T_{warmup}: Time for the system to refill the data pipeline, compile just-in-time (JIT) kernels, and stabilize throughput

Each component presents distinct optimization opportunities, and the dominant term varies by cluster configuration. Understanding this decomposition enables targeted investment in the bottleneck rather than uniform improvement across all components.



Detection latency climbs from seconds (crash) to hours (silent corruption).

7.7.1 Failure detection mechanisms

Detection is the first line of defense, governed by a fundamental trade-off between speed and false positive rate. A timeout that is *too aggressive* mistakes temporary network jitter for a node failure, triggering an unnecessary and expensive restart. A timeout that is *too conservative* allows the entire cluster to sit idle while a dead node holds up synchronization.

Heartbeat monitoring is the standard mechanism: each worker periodically sends "I am alive" signals to a central coordinator or monitoring service. Missing heartbeats trigger failure classification. The heartbeat interval H and timeout T_{timeout} control the trade-off. In high-scale clusters, heartbeat arrival times often follow a heavy-tailed distribution due to network congestion, necessitating adaptive timeouts rather than static thresholds. Production systems typically use $T_{\text{timeout}} = H + k\sigma_d$ where k ranges from 3 to 5 and σ_d is the observed standard deviation of network delay.

Collective communication timeouts provide a second detection layer. During synchronous training, collective operations (AllReduce, Broadcast) are blocking: if a single rank fails silently (a frozen GPU driver, for instance) every other rank in the communicator hangs indefinitely waiting for data that will never arrive. NCCL[33] provides configurable transport and RAS timeout parameters for this purpose, while higher-level frameworks may impose process-group operation timeouts. These settings are often set conservatively to avoid crashing jobs during legitimate periods of slow communication, which unfortunately extends T_{detect}.

Container orchestration health checks provide a third layer. Kubernetes and SLURM offer liveness probes (verifying that processes are running) and readiness probes (verifying that processes are ready to handle requests). These operate independently of the training framework, catching failures that application-level heartbeats might miss—such as a process that is alive but deadlocked.

Loss spike detection catches the most insidious failure mode: silent data corruption. Hardware errors that do not crash the process but corrupt the mathematical result (bit flips in ALU logic, for instance) manifest as sudden, catastrophic spikes in the loss function. The loss jumps 10–100× or collapses to NaN instantly. Unlike gradient explosions caused by high learning rates, these spikes occur without hyperparameter changes. Robust systems instrument the training loop to pause immediately upon detecting such anomalies, pinpoint the rank with the corrupted gradient via checksums or replay, and drain that node before restarting from the last healthy checkpoint.

Training dynamics monitoring extends detection beyond explicit errors. Monitoring loss values, gradient norms, and activation statistics can detect Byzantine failures that produce incorrect results without triggering exceptions. Sudden loss spikes, gradient explosions, or statistical anomalies in per-rank gradient distributions may indicate silent corruption that would otherwise go undetected for hours.

The operational question is therefore how long detection really takes once these layers interact.

33 **NCCL Timeout**: NVIDIA's collective communication library exposes transport and RAS timeout controls such as `NCCL_IB_TIMEOUT` and `NCCL_RAS_TIMEOUT_FACTOR` (NVIDIA 2026b). Higher-level launch and distributed-training frameworks can add their own restart and operation-timeout behavior (PyTorch Contributors 2026b). Aggressive timeout settings accelerate failure detection but risk false positives during legitimate long collective operations on large models, forcing a trade-off between detection latency and training stability that has no universal optimum.



Systems Perspective 7.5: Realistic failure detection latencies

Production experience shows that failure detection takes significantly longer than theoretical heartbeat timeouts suggest. The core challenge is distinguishing failures from stragglers, as table 7.8 records. These latencies exist because aggressive timeouts cause false positives (killing healthy-but-slow workers), while conservative timeouts delay real failure detection. Production systems typically use multi-stage detection: fast initial timeout triggers investigation, slower confirmation timeout triggers recovery.

Table 7.8: Production Failure Detection Latencies: Realistic detection times for process crashes, GPU hangs, network partitions, and silent data corruption, with the dominant cause of latency for each category, showing why static heartbeat timeouts alone are insufficient.

Failure Type	Typical Detection Time	Why
Process crash	5–30 seconds	Heartbeat timeout + verification retries
GPU hang	30–120 seconds	Must distinguish from legitimately slow kernel
Network partition	60–180 seconds	Must distinguish from temporary congestion
Silent data corruption	Minutes to hours	Requires statistical anomaly detection

7.7.2 Recovery procedures

Once a failure is classified, the recovery procedure executes a rigid sequence to restore consistency:

1. **Job termination**: A `SIGTERM` is broadcast to all surviving workers. In synchronous distributed data parallel (DDP) training, the loss of one worker invalidates the global communicator, forcing a full tear-down.
2. **Resource reclamation**: The scheduler marks the failed node as "draining" to prevent immediate rescheduling and requests a replacement from the spare pool.
3. **Job restart**: New containers are launched (from cache if available), and the training binary is re-initialized on all nodes.
4. **Checkpoint loading**: Each worker reads its state shard from the distributed filesystem. For sharded checkpoints, each worker loads only its partition.
5. **State synchronization**: Ranks handshake to establish a new communicator (for example, `ncclCommInitRank`), and workers verify they are all at the same training step.
6. **Training resumption**: The data loader fast-forwards to the correct batch index, and the training loop resumes from the checkpoint step.

Automatic recovery systems perform these steps without human intervention. Modern training frameworks integrate with cluster managers to automate parts of the sequence. DeepSpeed documents save/load routines for model and ZeRO optimizer checkpoints, which provide the durable state needed after failure (DeepSpeed Developers 2026a). PyTorch's `torchrun` elastic launch documents failure and membership-change behavior through its rendezvous mechanism, the coordinator-backed protocol by which surviving workers agree on membership and ranks after a restart (PyTorch Contributors 2026b, 2026a).[34]

Recovery validation is the final and often overlooked step. After loading a checkpoint, validation confirms successful recovery by verifying model parameters match expected shapes and dtypes, running a few training steps and checking that the loss is consistent with prefailure values, and confirming gradient computations produce expected statistics. If the loss diverges immediately after recovery, the checkpoint itself may be corrupted, requiring fallback to an earlier snapshot.

34 **Rendezvous**: From French "rendez-vous" (present yourselves), this coordination protocol requires all surviving workers to discover each other and agree on the new group membership before training can resume. The protocol's cost is a synchronization barrier that blocks all workers until the slowest one arrives, creating a recovery latency proportional to cluster heterogeneity.



Napkin Math 7.4: The recovery time budget

Problem: Consider our 175B parameter model training on 1,024 GPUs. The checkpoint size is approximately 3.7 TB (weights + Adam optimizer state). Estimate the cost of a single failure event.

Setup: The recovery budget T_{recovery} has four terms.

- $T_{\textbf{detect}}$: 60 s (conservative heartbeat timeout with verification retries).
- $T_{\textbf{restart}}$: 3 min (scheduler queue time + container launch + Python import overhead + NCCL initialization).
- $T_{\textbf{load}}$: 37 s (this is the aggregate-shared-storage read of the full 3.7 TB checkpoint at 100 GB/s aggregate; with sharded local-NVMe reads at 5 GB/s, each 3.6 GB shard loads in well under one second).
- $T_{\textbf{warmup}}$: 2 min (JIT kernel compilation, data pipeline buffer fill, TCP connection re-establishment).

Total: $T_{\text{recovery}} = T_{\text{detect}} + T_{\text{restart}} + T_{\text{load}} + T_{\text{warmup}} \approx 6.6$ min per failure event.

Impact: Recovery time optimization matters more as clusters grow. With the GPU-only baseline from table 7.1, a 1,024-GPU cluster has MTBF of 49 hours and experiences about 0.49 failures/day, losing about 3 minutes daily, a modest 0.2 percent overhead. However, a 10,000-GPU cluster has GPU-only MTBF of 5 hours and experiences about 4.8 failures/day, losing about 32 minutes daily—a 2.2 percent overhead equivalent to wasting roughly 5,293 GPU-hours every day (the daily lost minutes multiplied across all 10,000 GPUs).

7.7.3 Warm restart vs. cold restart

The standard recovery procedure described earlier is a **cold restart**: every process in the cluster is killed, and the entire state is reloaded from persistent storage. Cold restart is robust and simple (it

makes no assumptions about the validity of in-memory state) but it is wasteful. When a single GPU fails in a 1,000-GPU cluster, a cold restart discards the valid memory state of 999 healthy workers, forcing them all to reload from disk.

A **warm restart** preserves the state of surviving workers. When a rank fails, surviving ranks detect the failure but do not exit. They enter a waiting state, preserving loaded model weights and optimizer states in GPU memory. The scheduler replaces only the failed node. The new node joins, loads its partition of the state from disk (or receives it via broadcast from a peer), and the communicator is rebuilt. Training resumes with minimal disruption.

Warm restarts can reduce T_{load} and T_{warmup} to near-zero for 99.9 percent of the cluster, cutting total recovery time from minutes to seconds. For the 1,024 GPUs example, a warm restart avoids reloading 3.7 TB from storage, saving the 37 s T_{load} and 2-minute T_{warmup} for all but the replacement node.

The trade-off is software complexity. Warm restarts require the application to handle dynamic membership changes without leaking CUDA memory, corrupting shared state, or deadlocking during communicator reconstruction. Frameworks like TorchElastic and DeepSpeed provide this capability, but the failure modes during warm restart itself (a crash during communicator rebuild, for instance) must be handled by falling back to cold restart. A mature deployment can use warm restart as the fast path with cold restart as the safety net. Table 7.9 contrasts the two strategies:

Table 7.9: Cold vs. Warm Restart Trade-Offs: Cold restart kills every process and reloads from disk; warm restart preserves surviving workers and replaces only the failed node. Warm restart cuts recovery from minutes to seconds for the typical single-node failure but introduces software complexity around dynamic membership management. Mature deployments can run warm restart as the fast path and cold restart as the safety net.

Aspect	Cold Restart	Warm Restart
Recovery time	4–10 minutes (full reload)	30–90 seconds (single node reload)
State guarantee	Clean: all state from checkpoint	Assumes surviving state is valid
Implementation	Simple: kill all, reload all	Complex: dynamic membership management
Failure during recovery	Retry cold restart	Fall back to cold restart
Best for	Correlated failures, SDC events	Single-node failures, GPU errors

7.7.4 Recovery automation pipeline

At the scale of 10,000+ GPUs, human intervention for every failure is impossible: failures occur multiple times per day. Recovery must be an autonomous control loop managed by the cluster's control plane. Published large-scale systems from Meta, Google, and Microsoft illustrate multi-stage automation pipelines that classify failures and select the minimum viable remediation.

The pipeline operates in four ordered stages:

1. **Health monitoring daemon**: Continuously scrapes GPU telemetry (ECC error counters, temperature, fan speed, NVLink status) alongside application metrics like training loss and step throughput, often from a sidecar container.
2. **Failure classifier**: Determines whether a signal indicates a fatal error (for example, NVIDIA Xid error 48: double-bit ECC error), a transient stall (for example, temporary network congestion), or a performance degradation (for example, thermal throttling).
3. **Action selector**: Chooses the appropriate response. A process hang triggers a container restart (fast, local); a GPU hardware error triggers node drain and replacement (slower, requires spare capacity); a network partition triggers a pause-and-wait strategy (preserving in-memory state).
4. **Validation stage**: Runs a "canary batch" after recovery to confirm the loss matches prefailure values. If the loss diverges immediately, the checkpoint may be corrupted, triggering automatic fallback to an earlier snapshot.

This automation reduces Mean Time To Recovery (MTTR) from the 30–60 minutes typical of manual intervention to under 10 minutes for most failure types. The classification stage is critical: treating every failure as a cold restart wastes compute on transient issues, while treating a hardware failure as transient allows corrupted computation to continue.

7.7.5 Distinguishing stragglers from failures

Definition 7.2: Straggler

Straggler is a worker in a distributed training job that processes tasks significantly slower than its peers, creating a synchronization bottleneck.

1. **Significance**: In a synchronous system (BSP), cluster throughput is bounded by the speed of the slowest rank. A single 10 percent performance drop on one node can reduce the effective compute capacity of thousands of nodes by 10 percent.
2. **Distinction**: Unlike a hardware failure (where the node stops), a straggler continues to produce correct results but violates the temporal consistency required for efficient parallel execution.
3. **Common pitfall**: A frequent misconception is that stragglers are caused only by "bad hardware." In reality, they are often caused by system jitter: background OS processes, network congestion, or thermal throttling that varies across the data center floor.

A straggler is a worker that remains functionally correct but performs significantly slower than its peers. In synchronous training, stragglers are performance poison: the speed of the entire cluster is determined by its slowest component, because AllReduce cannot complete until every rank has submitted its gradients.

For the simplified no-overlap straggler model, the step time becomes:

$$T_{\text{step,straggler}}(N) = \max(T_{\text{rank}_0}, T_{\text{rank}_1}, \ldots, T_{\text{rank}_{N-1}}) + T_{\text{comm}}(N)$$

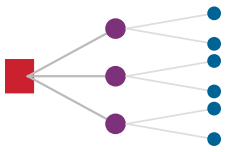
Stragglers arise from "gray failures" that do not trigger explicit errors: thermal throttling reduces clock speed, degrading interconnect cables increase communication latency, OS background processes (memory scrubbing, log rotation) consume CPU cycles, and data loading from a congested network filesystem introduces variable I/O delays. Unlike hard failures, stragglers do not trigger timeouts, allowing them to silently drag down global efficiency for hours.

The challenge is distinguishing stragglers from failures. Stragglers should trigger *mitigation* (redistribute work, replace the slow node). Failures should trigger *recovery* (checkpoint-restart). Aggressive timeouts treat stragglers as failures, causing unnecessary job restarts that waste more compute than the straggler itself. Conservative timeouts waste compute waiting for stragglers that will never speed up.

Straggler mitigation strategies span a spectrum of aggressiveness:

- **Backup workers**: Replicate work assigned to slow workers and use the first result, trading compute for latency.
- **Bounded staleness**: Allows training to proceed with stale gradients from slow workers, accepting a small convergence penalty.
- **Dynamic load balancing**: Redistributes data shards away from slow workers, reducing their per-step workload.
- **Proactive replacement**: Uses GPU telemetry trends (rising temperature, increasing ECC error counts) to detect degrading workers and replace them before they become stragglers.

A simple cost calculation shows why a severe straggler may be cheaper to replace than tolerate.



A single straggler idles every healthy accelerator.

Napkin Math 7.5: The straggler tax

Consider our 1,024-GPU cluster where a normal training iteration takes 1 second. A single GPU enters a thermally throttled state, clocking down to 50 percent speed, and now takes 2 seconds to complete its computation.

Because AllReduce cannot complete until every rank has submitted its gradients, the other 1,023 healthy GPUs sit idle waiting for the straggler.

Impact:

- **Normal step time**: 1 second
- **Straggler step time**: 2 seconds
- **Effective cluster speed**: 1 step/2s = 0.5 steps/sec

A single failing device (0.1 percent of the hardware) has reduced the throughput of the entire cluster by 50 percent. At $3/GPU-hour, this straggler wastes $1,536/hour in idle compute.

Systems insight: It is mathematically optimal to treat a severe straggler as a hard failure. Detecting and killing a slow node to force a restart onto healthy hardware yields higher long-term throughput than tolerating the degradation. The break-even point: if a straggler slows the cluster by more than $T_{\text{recovery}}/\text{MTBF}_{\text{system}}$ (the fraction of time spent recovering from failures), replacing it immediately is cheaper than waiting.

However, killing a slow node and restarting the job implies waiting for a replacement to maintain the original GPU count. The checkpoint-restart recovery model assumes restoring the *same* resource allocation: checkpoint, wait for a replacement, restart. Elastic recovery breaks that assumption by allowing the job to continue with *fewer* workers rather than idling the entire cluster until the original count is restored.

Checkpoint 7.5: Tuning detection and recovery

Verify your understanding of how detection latency and recovery strategy trade against one another:

- ☐ **Timeout tuning**: Given the heartbeat timeout rule $T_{\text{timeout}} = H + k\sigma_d$, explain what failure mode you trade away when you lower k from 5 to 3, and which one you make worse.
- ☐ **Warm vs. cold restart**: Choose a **warm restart** or a **cold restart** for a single GPU failure in a 1,000-GPU job with surviving state intact, and identify two conditions that push recovery back toward the cold path.
- ☐ **Hot spare cost**: Explain why a **hot spare** cuts recovery latency relative to detect-then-restart, including the standing cost the cluster pays to keep that spare available.
- ☐ **Straggler decision**: Determine what governs whether to wait out a slowed node or kill it and recover onto healthy hardware.

7.8 Elastic Recovery

Suppose a 1,024-GPU training job loses an 8-GPU node to a hardware fault, but the cluster has no spare nodes available. Under checkpoint-restart recovery, the remaining 1,016 GPUs sit idle for hours waiting for a repair. Elastic recovery instead allows the job to dynamically resize and continue with slightly less compute, breaking the rigid assumption of a fixed worker count.

Definition 7.3: Elastic training

Elastic Training is the capability of a distributed training job to dynamically adjust its worker count during execution without requiring a full restart.

1. **Significance**: It converts hard failures into throughput degradations rather than full stops. A 1,024-GPU job that loses 8 GPUs continues at 99.2 percent capacity instead of dropping to zero. It requires learning rate recalibration and gradient accumulation adjustment to maintain mathematical consistency as the global batch size changes.
2. **Distinction**: Unlike checkpoint-restart recovery (which halts the job, waits for replacement hardware, and reloads state), elastic recovery is always-live: the optimization loop never stops, it only changes its parallel width.
3. **Common pitfall**: A frequent misconception is that elasticity is "automatic." In reality, recovery requires coordinated adaptation: the training framework, the data loader, and the orchestrator must all synchronize their state to ensure that no data samples are lost or double-counted during the resize.

Figure 7.28 traces the recovery sequence: detect the failure, pause, rescale across the surviving workers, and resume.

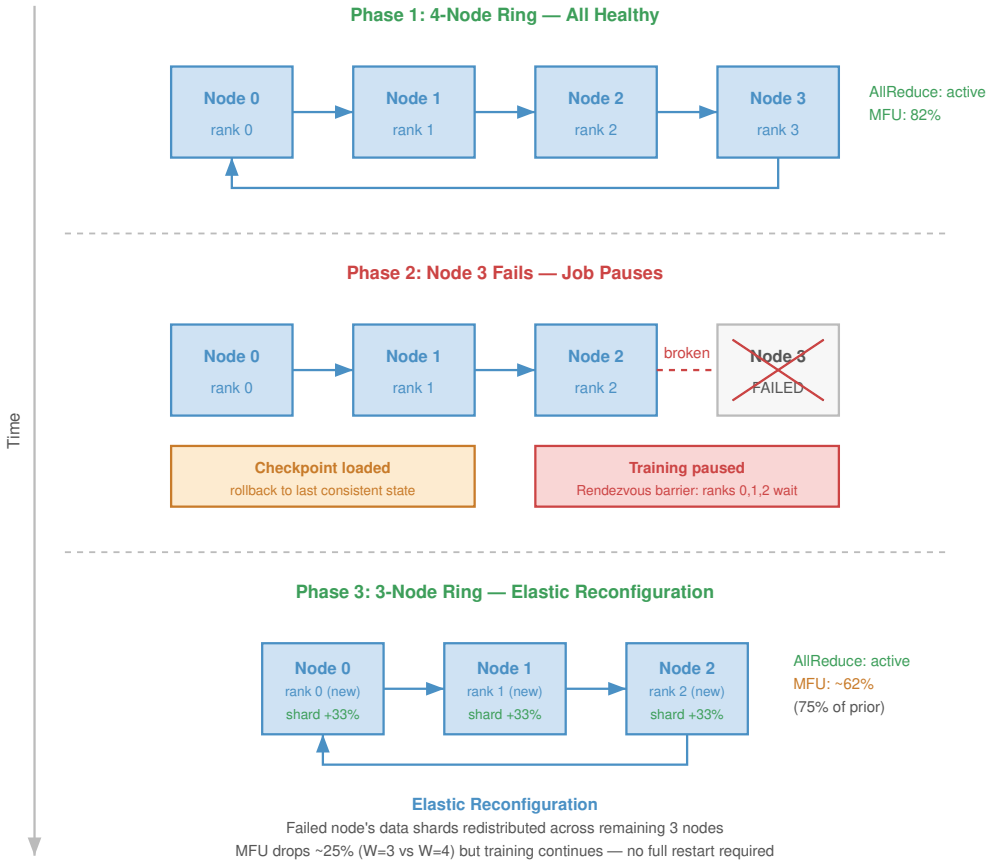


Figure 7.28: Elastic Training Recovery: Unlike static training which aborts on failure, elastic training adapts. When a worker fails, the job pauses, redistributes the dataset and model shards across the remaining $N-1$ workers, and resumes training from the last consistent state. This capability transforms hard failures into temporary throughput degradations.

7.8.1 Recovery adaptation mechanisms

As figure 7.28 illustrates, elastic recovery converts what would otherwise be hard failures into graceful capacity adjustments: a lost worker reduces the active count rather than aborting the job, the running step adapts to the smaller resource allocation, and the cluster resumes productive work within seconds rather than waiting minutes to hours for replacement hardware. When a failure reduces the worker count, the surviving workers must adapt several training components before they can resume. Each adaptation addresses a specific consistency requirement that, if violated, would corrupt the model or waste computation.

Batch size adjustment is the first concern: with fewer workers, each worker must process more samples to maintain the global batch size, or the global batch size must be reduced. Reducing global batch size during recovery may require learning rate adjustment to preserve convergence properties.

The relationship between batch size and optimal learning rate determines how aggressively the job can resize without destabilizing training. Goyal et al. (2017) demonstrated that a linear scaling rule works well for large-batch ImageNet training with warmup: when scaling the batch size by factor k, scale the learning rate also by factor k. Equation 7.10 expresses an alternative square-root heuristic that provides more conservative adjustment during recovery:

$$\eta_{\text{new}} = \eta_{\text{base}} \times \sqrt{\frac{N_{\text{new}}}{N_{\text{base}}}} \tag{7.10}$$

where N represents the number of workers (and thus the global batch size). The linear rule (Goyal et al. 2017) is often preferred for large-batch training with warmup when the other optimizer assumptions match the original recipe, while square-root scaling is a conservative recovery policy rather than a paper-backed law.

Gradient accumulation offers an alternative to learning rate adjustment: to maintain the original effective batch size with fewer workers, each surviving worker can accumulate gradients over multiple micro-batches before synchronization. If worker count drops by half, doubling the accumulation steps preserves the same effective batch size and avoids any learning rate change, at the cost of doubling per-step wall time.

Data loader redistribution is a coordination requirement that recovery must handle atomically. When workers are lost, the data loader must redistribute data shard assignments to ensure all training data is still processed and no samples are duplicated or dropped. A failed redistribution silently corrupts the training distribution.

State resharding adds further complexity when using sharded model parallelism (ZeRO/FSDP), because the loss of a worker means one shard of the model state is no longer locally resident. Recovery can proceed online (migrating the orphaned shard to a surviving worker) or through checkpoint reload (resharding from the last durable checkpoint).

7.8.1.1 Spot instance preemption as a failure class

Hardware faults are not the only event that removes workers from a running job. Cloud providers often price preemptible ("spot") instances below on-demand capacity, with the caveat that they can be reclaimed with little warning. From the training framework's perspective, a spot reclamation is indistinguishable from a hardware failure: workers vanish, and the job must decide whether to halt or adapt.

Traditional static jobs cannot survive spot preemption because a single lost node kills the entire run. Elastic recovery handles preemption identically to hardware faults:

1. The job detects the node loss (the mechanism is the same timeout or heartbeat failure).
2. It pauses briefly to redistribute the workload among surviving nodes.
3. Training resumes at reduced scale.

This recovery capability can unlock a significant cost advantage. Organizations can train on cheaper, preemptible hardware because elastic recovery converts each preemption into a temporary throughput reduction rather than a fatal error. In a scenario where spot capacity costs \$0.60/hour instead of \$3.00/hour, the savings can outweigh the efficiency loss from occasional resizing pauses.

7.8.1.2 Framework support for elastic recovery

The framework question is which resize invariants it can enforce automatically: group membership, shard ownership, data coverage, and batch math. The recovery adaptation mechanisms above (batch size adjustment, learning rate recalibration, data loader redistribution, state resharding) must be implemented by the training framework, and frameworks differ in how much of this recovery logic they automate after a failure.

PyTorch Elastic (TorchElastic) detects worker failures through a rendezvous mechanism[35] that re-executes whenever the worker group changes (PyTorch Contributors 2026b, 2026a). Surviving workers re-form the communication group with new `RANK` and `WORLD_SIZE` assignments after restart. TorchElastic handles rank reassignment and communication-group reconstruction, but leaves batch size, learning rate, and checkpoint semantics to user code.

DeepSpeed (Rasley et al. 2020) provides ZeRO-based distributed training and checkpointing building blocks for large models (DeepSpeed Developers 2026a). Its recovery model is checkpoint-based: after a failure, the job reloads from the last durable checkpoint, and Universal Checkpointing addresses portability across some changes in parallelism and topology (DeepSpeed Developers 2026b). The elastic behavior still depends on the surrounding launcher and resource manager rather than being made transparent by the framework alone.

Ray Train, built on Ray's actor model, provides a checkpoint-oriented recovery path for worker failure and node preemption. Ray can restart a worker group after a failure and resume from the latest available checkpoint, while the training function remains responsible for saving and loading sufficient state (Ray Project 2026).

Horovod Elastic builds on Horovod's data parallel training system (Sergeev and Balso 2018) with a documented elastic state API for worker additions and removals (Horovod Developers 2026). When the worker group changes, Horovod can reset ranks and reconstruct communication, while the training script remains responsible for convergence-sensitive choices such as learning-rate or batch-size policy.

The elastic training framework support summarized in table 7.10 compares the recovery automation and state management approaches across these frameworks.

35 **Rendezvous**: From French "rendez-vous" (present yourselves), this coordination protocol requires all surviving workers to discover each other and agree on the new group membership before training can resume. The protocol's cost is a synchronization barrier that blocks all workers until the slowest one arrives, creating a recovery latency proportional to cluster heterogeneity.

Table 7.10: Elastic Recovery Framework Comparison: Frameworks differ in how they detect worker failures and how much of the recovery sequence (group reconstruction, state redistribution, batch size adjustment) they automate.

Framework	Failure Detection	Automatic Recovery	State Resharding	Cluster Integration
PyTorch Elastic	Rendezvous timeout	Yes	Manual	Kubernetes
DeepSpeed	External	Checkpoint-based	Deployment-dependent	External schedulers
Ray Train	Actor supervision	Checkpoint retry path	Checkpoint reload	Ray Cluster
Horovod Elastic	Driver heartbeat	Yes	Manual	SLURM, Kubernetes

7.8.2 Model-specific training fault tolerance

The checkpoint and recovery strategies developed above require adaptation because workloads lose different kinds of state when they fail. The design question is therefore not simply how often to checkpoint, but which state variable would make a restart mathematically or economically wrong. For recommendation workloads, incremental checkpointing, tiered checkpointing, and embedding versioning protect freshness when full-state exactness is too expensive. Table 7.11 turns that question into a compact recovery map.

Table 7.11: Training Recovery State Invariants: Workloads require different checkpoint contents because they make different state variables quality-bearing. The correct recovery policy starts from the state that would make a restart wrong, then chooses sharding, incrementality, or domain-state capture accordingly.

Workload	State at Risk	Checkpoint Implication
LLM training	Optimizer state, curriculum position, document position, long-context schedule	Preserve the FP32 optimizer state where precision matters, shard writes with ZeRO/FSDP, move serialization off the critical path, and include the data-schedule position so the optimization trajectory resumes correctly.
Recommendation	Embedding freshness, feature-store version	Favor freshness over full-state exactness with incremental checkpointing, tiered checkpointing, and embedding versioning.
Vision	Augmentation seeds, shuffling order, batch-normalization statistics, progressive-resizing schedule	Capture the stochastic and normalization state that controls the input stream; weights alone are not enough to reproduce the training trajectory.
Scientific ML	Simulator state, search frontier, explored configurations, validation seeds	Treat the model, simulator, search process, and random state as one consistency unit so recovery does not duplicate exploration or invalidate comparisons.

The common pattern is that recovery correctness is broader than weight restoration. LLMs can reload weights but resume with the wrong curriculum position; recommendation models can reload dense parameters but serve stale embeddings; vision models can reload weights but alter augmentation or normalization state; and scientific models can reload a neural network while losing the simulator state that made the data meaningful.

Elasticity and checkpointing provide the necessary resilience for long-running batch training jobs, but the operational calculus changes completely once the model is deployed to users. The challenge shifts from protecting *weeks of batch computation* to protecting *milliseconds of real-time latency*.

7.9 Serving Fault Tolerance

When a user asks a voice assistant to turn off the lights, they will not tolerate a five-minute pause while the inference server reloads from a checkpoint. Serving models presents a fundamentally different challenge: users expect millisecond-level responsiveness even when backend GPUs crash.

Serving systems rely on replicas, routing, readiness checks, load balancing, KV cache state (cached transformer attention state), and graceful degradation as building blocks. Serving fault tolerance asks how those mechanisms behave when replicas, GPUs, or state stores fail under live latency budgets. The focus here is narrower than a complete serving architecture: each mechanism is treated as a reliability boundary for real-time inference.

7.9.1 Stateless vs. stateful serving

The first serving decision is where request state lives, because that choice determines whether failover is a retry or a state-reconstruction problem. In stateless serving, each request is independent. The serving system maintains no per-session state; all information needed to process a request is contained in the request itself plus the static model weights.

The stateless pattern appears when the input itself carries all context: an image classifier processes one image, an object detector processes one frame, a single-turn text classifier processes one snippet, and an embedding service maps one input to one vector. Fault tolerance can then focus on replica health and request routing. **Redundant replicas** serve requests in parallel, **load balancing** sends traffic only to healthy replicas, **health checks** remove failed replicas from rotation, and automatic replacement starts a new replica when capacity falls. When a replica fails, in-flight requests to that replica can be retried elsewhere, because no quality-bearing session state has been lost.

The simplicity of stateless serving makes it the simpler architecture when the application permits it. However, many ML applications inherently require state across requests. Consider a chatbot: a single-turn question-answering system can operate statelessly, processing each question independently. A conversational assistant that remembers previous exchanges, however, must maintain conversation history, transforming fault tolerance from simple retry to state preservation.

Stateful serving appears wherever the current request depends on earlier requests. LLM conversations accumulate KV cache (the cached key and value projections for all prior tokens, whose management Chapter 10 develops in full) across turns, streaming speech recognition maintains context from previous audio, recommendation sessions accumulate user context, and interactive editing maintains document state across edits. The fault-tolerance consequence is that failure loses quality-bearing state, not just serving capacity. KV cache loss requires reprocessing previous turns, session context loss forces users to repeat previous interactions, and accumulated user state loss degrades quality when context is unavailable. The mitigation choice follows the size, update rate, and quality value of that state: session affinity routes a session to the same replica, state checkpointing periodically saves session state, state replication maintains a standby copy, and graceful degradation keeps the service available at reduced quality if state cannot be recovered.

Table 7.12 contrasts stateless and stateful serving fault tolerance, showing how the fundamental difference manifests in every aspect of design, from request routing to recovery complexity.

Table 7.12: Stateless vs. Stateful Serving Fault Tolerance: Stateful serving introduces significant complexity in fault tolerance due to the need to preserve accumulated session state.

Aspect	Stateless Serving	Stateful Serving
Request routing	Any replica	Session-affine replica
Failure impact	Retry on another replica	Potential state loss
Recovery complexity	Restart and load weights	Reload state + reconstruct context
Redundancy approach	Active-active replicas	Replicated state + standby
Failover latency	Milliseconds (load balancer)	Seconds (state transfer)

7.9.2 Redundancy and replication

Redundancy buys availability only when replica placement and spare capacity match the failure domain being defended against. Multiple copies of serving capability let the system continue operating when individual replicas fail.

7.9.2.1 Availability calculations

For a single replica with availability A_{single} (probability of being operational at any given time), equation 7.11 quantifies how multiple independent replicas achieve higher system availability:

$$A_{\text{system}} = 1 - (1 - A_{\text{single}})^k \tag{7.11}$$

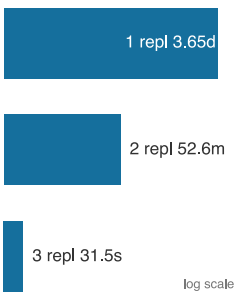
where k is the number of replicas. For a service whose single replica is available 99 percent of the time, redundancy compounds as follows.

For a single replica, $A_{\text{single}} = 99\%$ corresponds to 3.65 days of downtime per year. Adding independent replicas changes the tail of the failure distribution quickly: two replicas give $A = 1-(0.01)^2 = 99.99\%$, or 52.6 minutes of downtime per year, and three replicas give $A = 1-(0.01)^3 = 99.9999\%$, or 31.5 seconds. The math is powerful, but it rests on the independence assumption; shared power, shared networking, and shared software bugs reduce actual availability below these theoretical values.

7.9.2.2 Replication strategies

The availability equation says how much redundancy can help; the replication strategy determines what that redundancy costs during normal operation and failover. In **active-active replication** (figure 7.29, left), all replicas actively serve requests, so capacity is used efficiently but a failed replica immediately increases load on the survivors. In **active-passive replication** (figure 7.29, right), a primary serves traffic while standby replicas remain idle but ready, simplifying failover at the cost of unused resources during normal operation.

Geographic replication extends the same choice across regions. It protects against data-center or regional network failures, but requests routed to distant regions pay additional latency. Multi-tier designs combine several replication policies at once: edge caches replicate for latency, regional serving clusters replicate for availability, and a global primary or coordination layer preserves consistency and freshness.



Redundancy turns days of downtime into seconds.

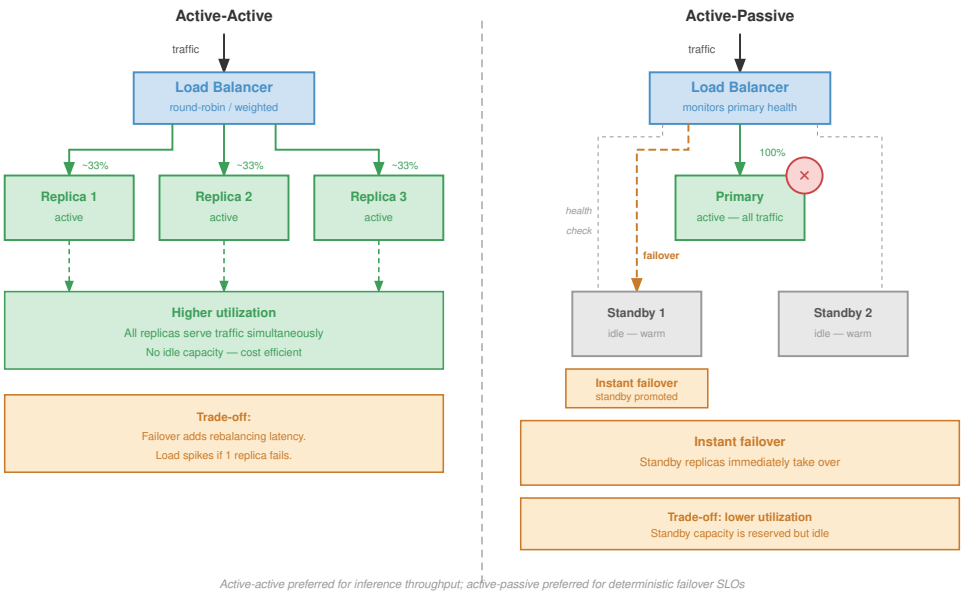


Figure 7.29: Serving Redundancy Strategies: Comparison of Active-Active vs. Active-Passive replication. Active-Active (left) distributes load across three active replicas, each receiving roughly one-third of traffic, maximizing utilization but requiring capacity headroom to absorb failures. Active-Passive (right) keeps standby replicas idle and synchronized via heartbeat, simplifying failover logic at the cost of idle resource utilization.

7.9.2.3 Replica placement and failure domains

Effective redundancy requires placing replicas in independent failure domains. Different machines tolerate individual machine failures; different racks tolerate rack-level failures from power or top-of-rack switch issues; different availability zones tolerate data-center-section failures; and different regions tolerate entire data-center failures. The independence level should match the availability requirement and cost constraint: regional replication is expensive because it duplicates compute and network capacity, but it is necessary when regional failure must not become user-visible downtime.

Placement only creates the possibility of recovery. The serving system still needs a control loop that notices the failure, removes the affected replica from traffic, and preserves enough state that the user-visible service continues with bounded degradation.

7.9.3 Failover mechanisms

When a replica fails, traffic must be redirected to healthy replicas. The speed and reliability of this failover determines the impact of failures on users. The mechanism decomposes into three questions: how the system detects health, how the routing layer acts on that signal, and what happens to stateful sessions already attached to the failed replica.

7.9.3.1 Health checking

Health checks decide when traffic should move, so they must test not only liveness but readiness and inference correctness. Liveness checks verify that the process is running and responsive, often through a simple HTTP endpoint that returns 200 and triggers restart when it fails. Readiness checks go further: for ML serving, a replica is not ready until model weights are loaded, the GPU is initialized and responsive, warmup has completed, and dependencies such as feature stores and caches are available. Inference health checks add a correctness probe by running a known input through the model and verifying the expected output, catching silent failures where the process is healthy but the model response is wrong.

Health check parameters set the false-positive/latency trade-off. A check interval such as 5 s controls sampling frequency, a 2 s timeout controls patience, a failure threshold of 3 missed probes marks a replica unhealthy, and a success threshold of 2 clean probes returns it to service.

7.9.3.2 Load balancer integration

Load-balancer design sets how much request context the routing tier can use during failover. L4 load balancing routes based on IP and port, offering simple and fast operation. L7 load balancing routes based on HTTP and gRPC content, enabling more specific routing. Service mesh adds traffic management, observability, and security around those routing decisions.

Load-balancer failover latency depends on health check frequency and failure detection logic. Aggressive settings enable fast failover but increase false positives, marking healthy replicas as unhealthy during transient issues.

7.9.3.3 Session affinity and stateful failover

For stateful serving, session affinity routes all requests within a session to the same replica. Load balancers maintain session-to-replica mapping through sticky sessions implemented with cookies, headers, or IP hashing. State failover choices then trade recovery speed against consistency and operational complexity: state loss accepts degraded quality by regenerating state from scratch, checkpointing periodically saves session state for recovery, replication copies state to a standby replica, and distributed state stores move session state into external systems such as Redis or Memcached.

The choice depends on state size, update frequency, and the quality impact of state loss; table 7.13 summarizes the trade-offs across the four common approaches:

Table 7.13: Stateful Failover Approaches: Four strategies for handling session state when a serving replica fails. State loss is fastest and simplest but discards quality on failover; checkpointing trades recovery latency for eventual consistency; synchronous replication offers strong consistency at the cost of operational complexity; distributed state stores split state management into a separate tier with configurable consistency.

Approach	Recovery Latency	Consistency	Operational Complexity
State loss	Fast	None	Low
Checkpointing	Medium	Eventual	Medium
Synchronous replication	Fast	Strong	High
Distributed state	Fast	Configurable	Medium

7.9.4 Model-specific serving fault tolerance

Model-specific serving differences follow from what state is expensive to lose and how quickly the user needs a response. The policy question is therefore a state-loss budget: which state can be rebuilt, which state must be replicated, and which state may be degraded temporarily without violating the product contract. Table 7.14 maps that question across the three common regimes.

Table 7.14: Serving Recovery State Invariants: Serving fault tolerance starts from the state whose loss changes user-visible quality. LLMs protect live context, recommendation systems protect feature freshness, and vision services protect preprocessing and device correctness.

Serving workload	State expensive to lose	Fault-tolerance policy	Degradation path
LLM conversation	KV cache, conversation context, prefix state	Replicate or checkpoint high-value session state; regenerate only when the latency budget allows.	Rebuild from transcript, reuse cached prefixes, or ask for retry.
Recommendation	Fresh user features, item features, embeddings	Replicate feature stores, cache hot features, and monitor freshness as a reliability signal.	Serve stale or default features with measured quality loss.
Vision service	Preprocessing state, device health, edge input	Retry stateless requests on healthy replicas and fail closed on preprocessing or accelerator-health faults.	Use a smaller local model, defer prediction, or return low confidence.

The KV cache[36] can be substantial (gigabytes for long contexts across attention layers). Losing the KV cache requires regenerating all previous turns, which can take seconds to minutes.

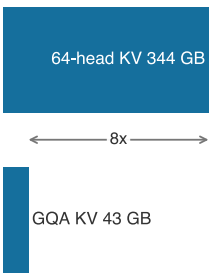
LLM serving choices trade KV-cache recovery latency against memory and storage overhead. The simplest strategy is to accept regeneration cost by rebuilding KV cache from conversation history after failure, but this can turn a long conversation into a multi-second or multi-minute restart. KV-cache checkpointing saves live state periodically and bounds the amount of regeneration, while KV-cache replication keeps a standby copy for fast failover at the cost of extra memory. **Prefix caching** narrows the state that must be recovered by storing common system prompts and shared context separately, so only session-specific state requires regeneration. Productized as prompt caching services like those offered by cloud providers, this approach stores and reuses KV cache for common prefixes, reducing both cost and recovery time for failures.

Recommendation serving protects freshness rather than conversational continuity. Recommendations depend on user and item features from feature stores, so feature-store unavailability can either degrade ranking quality or block recommendations entirely. The serving decision is the staleness budget: replicated feature stores keep recent features available across zones, local caches cover hot features, fallback to stale features accepts measured quality loss, and default features let the service return a lower-quality result when lookup fails. Real-time features such as recent user actions make freshness monitoring part of fault tolerance, because a pipeline that keeps serving old features is available operationally but wrong semantically. Large embedding tables may also sit behind dedicated embedding services, which need their own replication and failover plan.

Vision serving is usually closer to the stateless end of the spectrum because each image or frame can be retried on another replica. That simplicity does not eliminate fault tolerance work; it changes where the checks belong. GPU health monitoring must remove replicas with thermal throttling or memory errors before they return corrupted results, and preprocessing must fail closed when resize, crop, color conversion, or normalization stages malfunction. For edge vision, the main failure mode may be disconnection rather than a data-center replica crash, so the system often degrades by using a smaller local model, delaying nonurgent predictions, or returning a lower-confidence result until cloud connectivity returns.

When full redundancy fails (such as when an edge device loses connectivity, or a massive traffic spike overwhelms the available data center replicas), the system cannot simply crash. Instead, it must actively trade output quality for continued availability, a strategy known as graceful degradation.

[36] **KV Cache**: Stores key and value projections for all previous tokens across all attention layers, scaling as $2 \times N_L \times H_{\text{KV}} \times S \times d_{\text{head}} \times$ bytes. For a Llama-family 70B model with 80 layers, 8 key-value heads, 128K context, 128-dimension heads, and BF16 storage, the KV cache is about 43 GB per conversation; using 64 independent key-value heads would be about 344 GB. Losing this state on failure forces regeneration of all prior tokens, converting a subsecond failover into a minutes-long re-computation that violates serving latency SLAs.



A long conversation carries tens of GB of live state.

7.10 Graceful Degradation

During a major regional network outage, an e-commerce site suddenly loses access to its heavy, GPU-accelerated recommendation cluster. Rather than showing users empty pages or crashing, the

site instantly switches to serving precomputed, generic popular items. The serving fault tolerance mechanisms developed above aim to maintain full service, but graceful degradation dictates what happens when those defenses are overwhelmed.

Definition 7.4: Graceful degradation

Graceful Degradation is a fault tolerance strategy in which a system responds to resource exhaustion or component failure by deliberately reducing service quality (falling back to a smaller model, serving cached results, or returning partial outputs) rather than failing completely, maintaining measurable availability at reduced capability.

1. **Significance**: Graceful degradation converts total outage risk into a controlled quality reduction. A recommendation system that falls back from a 7B-parameter ranker to a collaborative-filtering model on GPU failure may reduce click-through rate by 8–15 percent but maintains request availability instead of turning every affected request into an error. The engineering calculation changes from total outage duration to measured quality loss during fallback.
2. **Distinction**: Unlike complete system failure (where the service returns errors to all requests), graceful degradation provides a managed transition through a predefined capability hierarchy—each fallback level has known accuracy, latency, and resource characteristics that allow the system to continue satisfying SLOs at reduced quality.
3. **Common pitfall**: A frequent misconception is that degradation is automatic in well-designed systems. Graceful degradation requires explicit preengineering: fallback models must be preloaded or precomputed, switchover logic must detect the failure condition and trigger the transition without human intervention, and each degradation level must have been validated to actually satisfy its reduced SLO before it is needed in production.

7.10.1 Degradation dimensions

A degradation plan must name which service property will be sacrificed before an incident begins. Table 7.15 summarizes the main levers. The table is a pre-incident design checklist, not an outage-time brainstorming aid: each sacrificed property needs a measured quality budget, an activation condition, and a recovery path before the failure occurs.

Table 7.15: Graceful Degradation Dimensions: Degradation plans choose which service property can be reduced while still returning a useful answer. Naming the dimension in advance prevents the system from treating every failure as a binary serve-or-crash decision.

Dimension	What the System Sacrifices	Typical Fallback
Quality	Model accuracy or ranking quality	Use a simpler model, such as a collaborative-filtering fallback for a multi-tower recommender.
Latency	Response speed	Batch more requests together to preserve throughput and model quality under high load.
Coverage	Result completeness	Return top-10 search results instead of top-100.
Freshness	Recency of computed results	Serve cached or stale candidates, such as hour-old news recommendations during an outage.
Feature completeness	Input richness	Use content features when user history or real-time context is unavailable.

These dimensions are not interchangeable. A search product may sacrifice coverage by returning fewer results while keeping freshness intact; a recommender may sacrifice freshness by serving cached candidates while preserving latency; and an interactive assistant may sacrifice quality by using a smaller model while keeping the session alive. The correct fallback is therefore workload-specific: it follows the property whose temporary loss least harms the application and whose recovery path can be validated before the incident.

7.10.2 Graceful degradation strategies

The dimensions in table 7.15 become operational only when the service binds each sacrificed property to a specific fallback mechanism. The strategy ladder starts with model fallback when the primary inference path is too expensive or unavailable, then moves to feature fallback when input quality degrades, and finally to load shedding when demand exceeds capacity and the system must preserve the highest-value requests first.

7.10.2.1 Model fallback

Model fallback turns quality into an explicit availability lever by maintaining multiple model versions with different resource requirements. The primary model provides full capability at the highest resource cost, a secondary model reduces capability and resource demand, a tertiary model preserves minimal capability, and a static fallback serves precomputed defaults with no inference at all. When the primary path is unavailable or overloaded, the policy descends only as far as resource pressure requires, then returns upward after the primary path is healthy again.

An image classification cascade makes the trade-off concrete. The primary path might run a ViT-Large model with 307 million parameters and 88 percent ImageNet top-1 accuracy, fall back to EfficientNet-B4 with 19 million parameters and 83 percent accuracy under pressure, then fall back again to MobileNet-V3-Large with 5.4 million parameters and 75 percent accuracy when only minimal classification remains affordable. If even that path cannot meet the SLO, the service can return cached labels, a coarse category, or "classification unavailable" rather than blocking the request indefinitely. For that ladder to be real rather than aspirational, the production system must keep multiple models deployed, route requests to the appropriate level, monitor fallback frequency, and measure the quality impact of each level.

Recommendation and ranking systems use the same pattern when a complex deep learning model falls back to a top-n popularity list, a linear ranker, or a cached response during primary serving failures. Complex models often fight for the last mile of accuracy while simple heuristics provide the bulk of the utility, so the fallback path must be monitored as a first-class quality signal rather than treated as a silent failover. Otherwise the product appears healthy while quality is silently traded for availability.

7.10.2.2 Feature fallback

Feature fallback controls how much input quality the model may lose before the request must be blocked. When feature retrieval fails, precomputed population-level defaults can substitute for missing user or item features: a recommender might use the average user embedding, genre-level item defaults, or the most recent cached value for a real-time signal. When defaults are too weak, the system can compute approximate features from available data, such as demographic similarity for missing user history, text embeddings for missing item attributes, or time-based defaults for missing context.

The key design step is prioritizing features by their contribution to prediction quality. Table 7.16 summarizes a four-level policy in which critical features block the request, important features use defaults, useful features use cached values, and optional features can be omitted with bounded quality impact:

Table 7.16: Graceful Feature Degradation Tiers: Features classified by importance to prediction quality, with the corresponding response when each tier is unavailable. Critical features block the request entirely; lower tiers allow the system to fall back to defaults, cached values, or omission with bounded quality impact. The tiers convert a binary "feature available or not" question into a four-step degradation ladder.

Tier	Example Features	Missing Action	Quality Impact
Critical	User ID, Item ID	Block request	Cannot serve
Important	User history, Item attributes	Use defaults	5–10% quality loss
Useful	Real-time context	Use cached	2–5% quality loss
Optional	Secondary signals	Omit	< 2% quality loss

7.10.2.3 Load shedding

Load shedding protects the system by sacrificing selected requests before overload spreads to every request. Random shedding is the simplest policy: drop a fraction of incoming requests and preserve enough capacity for the rest. Its weakness is that it treats all requests as equally valuable. Priority-based shedding turns quality-of-service degradation into an explicit policy by serving premium users before free users, revenue-generating requests before analytics requests, and interactive traffic before background batch work. The system is still degrading, but it is degrading according to a predeclared service contract rather than whichever queue happens to fill first.

Admission control applies the same idea at system entry points. It rejects requests that would exceed capacity rather than accepting work that will degrade every request already in the system. Circuit breakers[37] protect downstream dependencies by failing fast when a model replica, feature store, or supporting service is unhealthy.

Circuit breakers operate in three states: closed (normal operation), open (failing fast to prevent resource exhaustion), and half-open (probing for recovery). Figure 7.30 shows how these state transitions both shield the system from cascading failures and automatically test whether conditions have improved.

37 **Circuit Breaker**: Named after the electrical safety device that cuts power during overload. In software (popularized by Michael Nygard's 2007 *Release It!*), the pattern wraps service calls and "trips open" after a failure threshold, failing fast instead of waiting on a dead dependency. For ML serving, this prevents a single failing model replica or feature store from exhausting connection pools and cascading failures across the entire inference fleet.

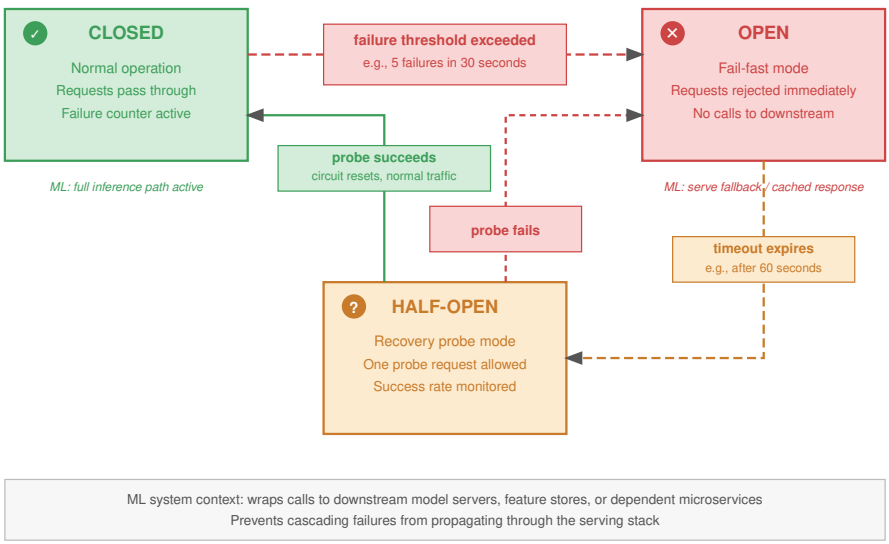


Figure 7.30: Circuit Breaker States: The circuit breaker protects the system from cascading failure. Closed: normal operation. Open: error threshold exceeded; all requests fail fast to prevent resource exhaustion. Half-Open: after a timeout, a limited number of requests are allowed through to probe the dependency's health. Success resets to Closed; failure returns to Open.

7.10.2.4 Graceful degradation implementation

The three-state cycle prevents a single failing dependency from consuming the entire system's connection pool: the open state fails fast, and the half-open state probes for recovery before restoring full traffic. Implementation begins by turning the degradation ladder into a control loop. Tail-latency percentiles show when users experience delay, error rates show which dependency is failing, CPU/GPU/memory utilization shows whether the system is saturated, and queue depth shows whether the service is accepting work faster than it can finish it. These health signals choose when to enter and leave each degradation level.

Degradation triggers define conditions that activate degradation. Listing 7.1 illustrates three common trigger conditions that progressively activate fallback mechanisms based on latency, error rate, and feature store health.

Listing 7.1: Progressive Degradation Triggers: Conditions that activate graceful degradation based on tail latency, error rate, and feature store health. Each trigger activates a different fallback mechanism appropriate to the detected problem.

```
# Monitor tail latency for user-facing impact
if p99_latency > threshold:
    activate_model_fallback()  # Switch to faster, simpler model

# Track error rates to detect downstream failures
if error_rate > threshold:
    activate_circuit_breaker()  # Fail fast, prevent cascade

# Feature store slowdowns degrade recommendation quality
if feature_store_latency > threshold:
    activate_feature_fallback()  # Use cached/default features
```

Degradation should increase progressively as conditions worsen rather than switch at one cliff. The system can raise the fallback percentage gradually as load increases, then require sustained improvement before traffic returns to higher-capability paths. Hysteresis prevents oscillation between primary and fallback modes when the service hovers near a threshold.

7.10.3 Degradation monitoring and alerting

Because graceful degradation trades quality for availability, monitoring must make that trade visible rather than letting the system appear healthy while serving lower-fidelity results. The monitoring surface must expose how much service quality has been traded away. Table 7.17 turns that surface into an action map: fallback share reveals how often model quality was reduced, default-feature share reveals input-quality loss, drop rate reveals capacity protection, and the primary-vs-fallback quality gap reveals whether the degraded path remains acceptable.

Table 7.17: Degradation Monitoring Action Map: Graceful degradation is safe only when the quality trade is observable. Each signal ties a degraded serving mode to the escalation and postincident question that prevent silent quality loss.

Signal	Quality trade exposed	Escalation rule	Postincident question
Fallback-model request share	Model capability reduced to preserve serving	Sustained activation beyond 5 minutes becomes a warning.	Did the fallback preserve enough user value?
Default-feature request share	Input richness reduced because dependencies lag	Severe degradation affecting over 50% of requests becomes critical.	Which feature dependency needs replication or caching?
Load-shed request rate	Coverage reduced to protect the service	Any growth after the circuit breaker opens triggers capacity investigation.	Was admission control early enough to prevent cascade?
Primary-vs-fallback quality gap	Product quality lost during the incident	Degradation beyond 1 hour escalates even when availability metrics look healthy.	Is the fallback path still a valid product experience?

After the event, postincident analysis should identify the root cause, evaluate fallback effectiveness, measure user and business impact, and define improvements that prevent recurrence. Implementing these fallback mechanisms safely requires deep visibility into the system's runtime state. When a complex recommendation pipeline begins degrading, operators must rapidly determine which microservice is failing, and that diagnosis depends on distributed debugging and observability.

7.11 Distributed Debugging and Observability

At 2:00 AM, latency on the flagship generative API spikes from 200 ms to 5 seconds, but CPU, memory, and network metrics all look perfectly normal. Deciding whether to shed load, restart replicas, or degrade gracefully requires knowing exactly where the request is stalling across hundreds of microservices. Distributed debugging and observability provide the diagnostic capability required to locate these invisible bottlenecks.

7.11.1 Why distributed ML systems are hard to debug

Distributed ML debugging is hard because the evidence needed for recovery is split across timing, state, scale, and model behavior. Distributed systems exhibit nondeterministic behavior[38] from multiple sources. Network timing variations change execution order, thread scheduling differences alter race conditions, GPU kernel execution order varies across runs, and floating-point operation ordering changes results. A bug that manifests on one execution may not reproduce on subsequent executions. These "Heisenbugs" appear to disappear when observed.

Partial failures turn that nondeterminism into a recovery problem. Unlike single-machine systems where failures are typically total, distributed systems experience partial failures where some components fail while others continue, and the interaction between working and failed components produces complex failure modes. Scale then removes manual inspection as a viable debugging method: with thousands of components, automated tools must filter massive telemetry streams and identify the few signals that explain the incident. ML systems add model behavior to the same diagnostic problem because silent accuracy degradation produces wrong results without errors, numerical issues like NaN and infinity propagate through computation, data-dependent bugs manifest only for specific inputs, and learning causes expected behavior changes that can resemble bugs.

38 **Heisenbug**: Coined in hacker culture circa 1983, formalized by Jim Gray in his 1985 analysis of computer failures, as a pun on Heisenberg's uncertainty principle. The bug disappears under observation because adding instrumentation changes timing. In distributed ML training, this is especially pernicious: inserting gradient logging alters NCCL collective timing, masking the very race condition that caused the silent corruption.

7.11.2 Observability pillars

Observability is useful only when it connects a symptom to an action: shed load, restart a replica, roll back a checkpoint, or degrade gracefully. Suppose a recommendation service suddenly violates its latency SLO while the model replicas still report healthy GPU utilization. Metrics establish the timing and scope of the symptom, traces show which service span is consuming the request budget, and logs explain what happened inside that service at the moment the cascade began. Figure 7.31 summarizes those three evidence types, but the operating rule is correlation: no signal is sufficient unless it can be joined to the same request, model version, feature version, and deployment event.

Metrics are the first evidence because they compress fleet behavior into time series that can trigger action. In ML fleets, useful metrics span infrastructure signals such as CPU and GPU utilization, memory allocation, network bandwidth, and storage I/O; application signals such as request rate, latency percentiles, error counts, queue depths, and cache hit rates; and ML-specific signals such as inference latency by model, batch utilization, feature retrieval latency, feature freshness, and prediction distributions for drift detection. A metric stream becomes fault-tolerance infrastructure only when an alert maps to a recovery decision. High p99 latency may trigger load shedding, a falling cache hit rate may trigger fallback features, and a sudden prediction-distribution shift may block a rollout before it corrupts user-facing decisions.

Logs provide the event context that a metric intentionally discards. Structured logging uses formats such as JavaScript Object Notation (JSON) with consistent fields, so incident tooling can search by service, request, model version, GPU, feature store, or trace identifier. Listing 7.2 shows a GPU memory allocation failure that records both the local failure context and the trace identifiers needed to connect the event to distributed metrics and spans.

Listing 7.2: Structured JSON Log Entry: A structured log entry for a GPU memory allocation failure, including trace and span identifiers for correlation with distributed traces and resource metrics for diagnosis.

```
{
  "timestamp": "2024-01-15T10:23:45.123Z",
  "level": "ERROR",
  "service": "inference-server",
  "trace_id": "abc123",
  "span_id": "def456",
  "message": "GPU memory allocation failed",
  "gpu_id": 3,
  "requested_bytes": 4294967296,
  "available_bytes": 2147483648
}
```

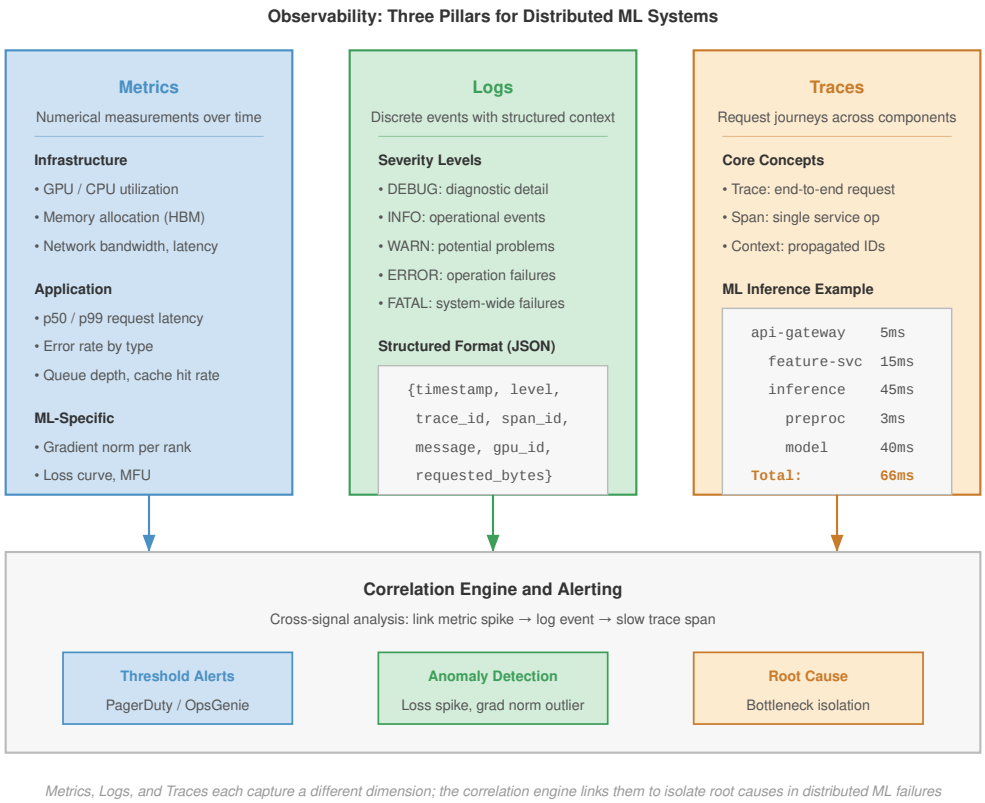


Figure 7.31: The Three Pillars of Observability: Diagnosis of fleet failures requires correlating three signal types. Metrics reveal *when* and *where* a problem occurs (for example, GPU utilization drops). Logs provide the *what* through event context (for example, Out-of-Memory exception). Traces provide the *why* by linking events across the distributed lifecycle (for example, identifying the specific request that triggered the memory spike).

For fault tolerance, the important logging property is consistent correlation rather than the particular backend. Log levels should map incidents to escalation consistently across components, but the recovery system depends more on whether operators can query events by the same request, trace, model version, feature version, and deployment identifiers that appear in metrics and spans. Log aggregation is therefore useful because it preserves the join between local failure context and fleet-wide symptoms.

Traces show where latency or failure propagated across distributed components. A **trace** is the end-to-end journey of a request, a **span** is one operation within that journey, and **context propagation** carries the trace identity across service boundaries. A trace through an ML inference pipeline illustrates how these spans compose:

```
Trace: user-request-12345
  Span: api-gateway (5 ms)
    Span: auth-service (2 ms)
  Span: feature-service (15 ms)
    Span: user-feature-lookup (8 ms)
    Span: item-feature-lookup (12 ms)
  Span: inference-service (45 ms)
    Span: preprocessing (3 ms)
    Span: model-inference (40 ms)
    Span: postprocessing (2 ms)
  Span: response-formatting (1 ms)
Total: 66 ms
```

OpenTelemetry provides a standard API for distributed tracing. Backend systems like Jaeger, Zipkin, or cloud tracing services store and visualize traces. The durable requirement is not the particular tracing backend; it is preserving enough context to answer whether the fault lives in request routing, feature retrieval, model execution, postprocessing, or a dependency outside the model service.

7.11.3 ML-specific debugging

Operational signals are not enough when a computation is numerically valid but wrong for the model. ML systems also require specialized debugging capabilities.

7.11.3.1 Numerical debugging

Numerical debugging targets failures where tensors remain valid objects but contain invalid values. NaN detection[39] is essential because NaN values propagate silently through all downstream computations. Listing 7.3 shows a minimal check that catches corruption before it reaches users.

Listing 7.3: NaN Detection: Checking model outputs for NaN values prevents silent corruption from propagating to downstream consumers. Logging the input hash enables reproducibility during diagnosis.

```
# Check tensor for NaN values
# (propagate from any corrupted computation)
if torch.isnan(output).any():
    log.error("NaN detected in output", input_hash=hash(input))
    # Log input hash for reproducibility, then fallback or fail fast
```

During training, gradient statistics make numerical faults visible before they corrupt the whole run. Gradient norm detects explosion or vanishing, gradient distribution detects anomalies, and layer-wise gradients identify problematic layers.

Mixed-precision training[40] can introduce numerical issues. Monitor for loss scale adjustments indicating underflow, gradient overflow exceeding FP16 range, and inconsistency between FP16 and FP32 results.

7.11.3.2 Data debugging and validation

Data bugs surface as valid-looking but wrong inputs, so the debugging target is not just numerical validity but semantic correctness at each pipeline boundary. Validation checks expected format, shape, value ranges, required fields, and encoding before data reaches the model, catching malformed examples while the failure is still local to the input path.

Once the input passes those static checks, the system needs evidence about behavior over time. Distribution monitors track feature drift, null-rate changes, and outliers as they emerge, while transformation instrumentation logs intermediate shapes and statistics so teams can compare each stage against known-good data. Together, these signals separate a corrupted input path from a numerical failure inside the model.

7.11.3.3 Straggler detection and analysis

Straggler diagnosis asks which component is slow enough to become a failure-equivalent bottleneck. The same request may succeed on every replica, but the slowest replica still determines tail latency and can force retries that look like failure under load. Operation-level timing instrumentation measures time for each operation, enabling latency attribution across pipeline stages. Listing 7.4 wraps operations in context managers that emit per-span timing metrics, making it straightforward to compare performance across replicas.

[39] **NaN Propagation**: IEEE 754 specifies that any arithmetic operation involving NaN produces NaN, meaning a single NaN in one gradient computation silently corrupts every downstream parameter update. Common triggers in ML training include division by zero in normalization layers (batch norm with zero variance), log of nonpositive values, and FP16 overflow to infinity. Without per-step NaN detection, an entire training run can produce a model full of NaN weights before any monitoring alarm fires.

[40] **Mixed-Precision Numerical Issues**: FP16's dynamic range spans only 6×10^{-8} to 65,504, compared to FP32's 1.2×10^{-38} to 3.4×10^{38}. Loss scaling (multiplying loss by 1,024 before backpropagation, then dividing gradients) prevents underflow, but overflow still triggers automatic step-skipping that wastes computation. BF16 mitigates this by matching FP32's exponent range at the cost of reduced mantissa precision, trading numerical resolution for training stability.

Listing 7.4: Operation-Level Timing: Context manager instrumentation attributes latency to individual pipeline stages, enabling straggler detection by comparing the same span across replicas.

```
# Wrap operations in timing context managers for latency attribution
with timer("feature_lookup"):
    features = feature_store.lookup(
        ids
    )  # Often the latency bottleneck
with timer("model_inference"):
    predictions = model(features)  # GPU time, compare across replicas
```

Compare component timing across replicas using percentile analysis. p50 shows typical performance, while p99 shows tail latency. Comparing p99 across replicas identifies workers whose slow path has become a fleet-wide bottleneck.

The diagnosis must then separate the root causes that produce the same slow-worker symptom. Hardware issues include thermal throttling and memory errors, data skew means some inputs are slower to process, resource contention occurs when other processes consume resources, and network issues create slow connections to data stores.

7.11.4 Common failure patterns

The observability pillars enable detection of recurring failure patterns that experience with large-scale ML systems has identified. Figure 7.32 catalogs the most common training loss signatures, each requiring a different diagnostic and recovery response.

7.11.4.1 Training failures

A training-loss signature is useful because it maps a curve to a recovery decision. Figure 7.32 shows the curves first; table 7.18 then makes the operational mapping explicit, treating the curve as evidence for a specific recovery action.

Table 7.18: Training Failure Response Map: Training-loss curves are operational evidence only when they lead to different recovery actions. The response map ties each signature to the failure class most likely to explain it and to the first action operators should take.

Signature	Likely Interpretation	Recovery Response
Spike followed by recovery	Transient data issue or numerical instability	Continue training, but investigate the incident to rule out a systematic input problem.
Spike followed by plateau	Learning rate too high, corrupted checkpoint, or data bug	Roll back to an earlier checkpoint and validate the data path before resuming.
Gradual divergence	Silent data corruption, hardware error, or distributed-training desynchronization	Isolate the failing rank, validate checkpoint integrity, and compare telemetry across workers.
Hang without an error	Collective deadlock or crashed worker blocking synchronization	Use timeout detection and restart or reconstruct the worker group.

7.11.4.2 Serving failures

Serving failures require the same symptom-to-action mapping under much tighter latency constraints. Table 7.19 separates the common symptoms by what they reveal and what the serving system must do first.

Table 7.19: Serving Failure Response Map: Serving incidents must be classified quickly because every diagnosis competes with a user-facing latency budget. The table ties each symptom to the first operational response rather than treating all serving failures as generic errors.

Symptom	Likely Interpretation	Immediate Response
Latency spikes	Resource contention, garbage collection, cold caches, or model reloads	Check placement and capacity; repeated spikes usually mean the problem is no longer transient.

Symptom	Likely Interpretation	Immediate Response
Rising error rates	Dependency failure, data-format change, or model bug	Investigate immediately because errors compound across dependent services.
Silent quality degradation	Model drift, feature degradation, or data-pipeline fault	Use quality monitoring beyond standard operational metrics.
Cascade failure	Timeout exhaustion, resource depletion, or error propagation from one failing component	Trip circuit breakers and preserve isolation boundaries before the failure spreads.

Cascade failures are particularly insidious because the root cause is obscured by the symptoms it generates. A concrete diagnosis shows how the three observability pillars work together to trace a cascade back to its origin.

Example 7.4: Diagnosing a cascade failure

Scenario: A recommendation system experiences sudden latency spikes.

Diagnosis: The three observability pillars work together to diagnose the root cause. Metrics reveal the symptom: p99 latency jumped from 45 ms to 800 ms at 14:32, with error rate increasing from 0.1 percent to 15 percent. Traces isolate the bottleneck: the feature-service span consumes 700 ms instead of the normal 12 ms, while the model-inference span remains normal at 40 ms. Logs identify the root cause: feature-service logs show repeated connection timeouts to the user embedding cache at 14:31, followed by cache miss storms as requests bypass the failed cache and hit the embedding database directly.

Systems lesson: The cache node failure caused a cache-miss avalanche, overwhelming the embedding database and propagating latency to all requests. The fix is a circuit breaker on cache access, falling back to default embeddings when cache is unavailable.

7.12 Case Studies

Published production systems make the chapter's scale lesson concrete: failure detection, state preservation, and observability reappear as checkpoint tax, topology reconfiguration, chaos testing, and elastic recovery. The cases below use public reports from Meta, Google, Netflix, and Microsoft to show representative engineering patterns. Treat the operational numbers as reported magnitudes that reveal the design pressure, not as universal constants. Meta's OPT-175B training run on a 992-GPU cluster provides the first example: dozens of hardware failures had to be absorbed without aborting training.

7.12.1 Large-scale LLM training at Meta

The training of the Open Pre-trained Transformer (OPT-175B) at Meta provides a well-documented study in the physics of failure for large deep-learning jobs (Zhang et al. 2022). Using a cluster of 992 NVIDIA A100 GPUs continuously for two months, the engineering team faced a statistical certainty: hardware components would fail, and they would fail often. Over the course of the training run, the team logged approximately 90 manual restarts driven by hardware failures (ECC memory errors, NCCL/InfiniBand network issues, lost GPUs) and training-stability incidents, alongside the cycling of more than 100 hosts (Meta AI Research 2022). In a synchronous data-parallel regime, a single GPU failure halts the entire cluster, making the mean time between failures (MTBF) of the aggregate system a fraction of any individual component's reliability. For OPT-175B, roughly two machines went down per day, dropping the effective system-wide MTBF to a fraction of a day and necessitating a fault tolerance strategy that treated interruption as the norm rather than the exception.

During the OPT run, the team monitored progress through train-log freshness checks: the public logbook records a 15-minute modified-file threshold, later relaxed to one hour, as part of restart monitoring (Meta AI Research 2022). Detection was only half the battle; the critical engineering challenge was minimizing the "restart tax"—the time lost reloading the model and optimizer states from persistent storage. Early in the project, synchronous checkpointing to a remote distributed file system consumed nearly 12 percent of the total effective training time, a prohibitive overhead

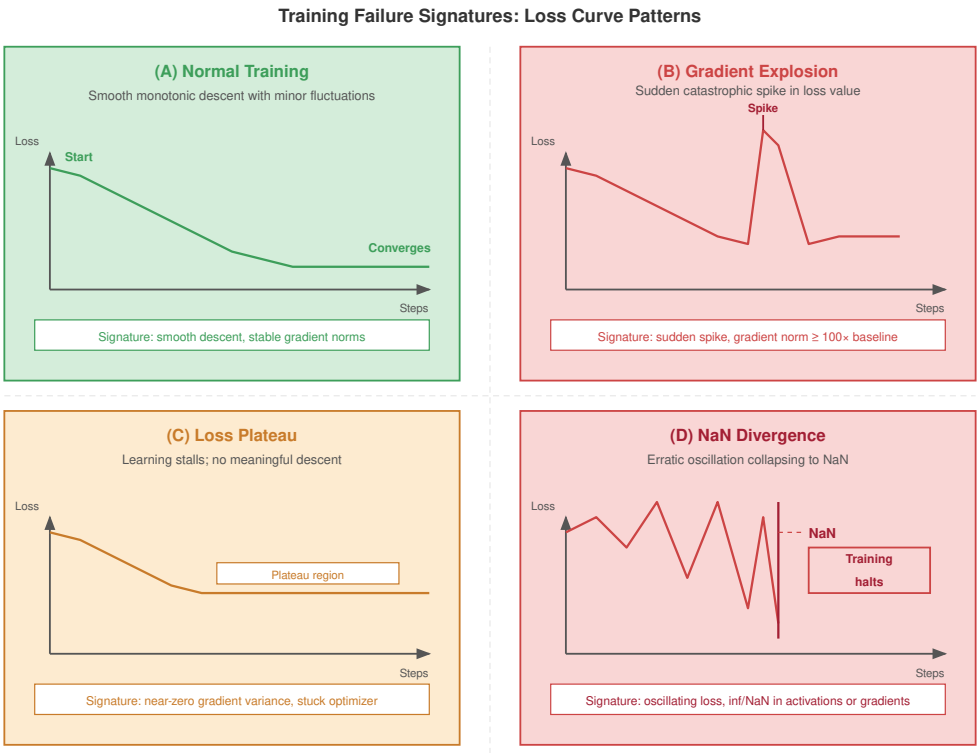


Figure 7.32: Training Failure Signatures: Common patterns of model loss over time that indicate underlying system or algorithmic failures. (A) Normal Training. (B) Gradient Explosion. (C) Loss Plateau. (D) NaN Divergence. Identifying these signatures automatically is essential for rapid recovery in autonomous training pipelines.

that extended the project timeline by weeks. To combat this, the team implemented asynchronous checkpointing, offloading the serialization of the 350 GB model state to CPU memory first, then streaming it to disk in the background while computation for the next batch resumed immediately. This optimization reduced checkpoint overhead to less than 3 percent, reclaiming hundreds of GPU-hours.

Recovery procedures also had to account for nonhardware failures, specifically "loss spikes" caused by numerical instabilities. Unlike a hardware crash where the last checkpoint is valid, a loss spike implies the model weight trajectory has become corrupted. The recovery strategy combined a "last good checkpoint" rollback with intervention in the optimizer: when gradient overflows or loss-scale floors were observed, the team reverted to an earlier checkpoint and resumed training with a lowered learning rate (ultimately running at roughly two-thirds of the rate OpenAI used for GPT-3) to stabilize the trajectory (Meta AI Research 2022). This dual-layer resilience—handling both physical silicon failures and mathematical divergence—allowed Meta to sustain approximately 147 TFLOP/s per A100 (near 50 percent of FP16 peak) despite daily interruptions (Zhang et al. 2022). The enduring lesson from OPT-175B is that at scale, the trade-off between checkpoint frequency and training throughput is a central variable in determining the feasibility of a model; checkpointing too often wastes compute, while checkpointing too rarely risks losing days of progress to a single bit flip.

7.12.2 Google TPU pod resilience

Google's TPUv4 supercomputer takes a fundamentally different architectural approach to fault tolerance, driven by a custom Inter-Chip Interconnect (ICI) fabric and a network of Optical Circuit

Switches (OCS) that dynamically reconfigure the topology. A TPUv4 pod contains 4,096 chips organized as 64 "cubes" of 64 chips each, with each cube arranged in a $4\times4\times4$ 3D mesh (Zu et al. 2024). In this architecture, the network effectively is the computer: a single chip or optical link failure does not merely reduce capacity by $1/4096$th—it creates a hole in the communication topology that would deadlock the synchronous mesh required for collective operations such as AllReduce.

Consequently, Google's strategy focuses on slice abstraction rather than in-place repair. The OCS network combines multiple healthy cubes into logical "slices" sized to the training job's needs. When a fault is detected—via two software components, `libtpunet` and `healthd`, that monitor link integrity and machine health—the orchestration system does not attempt to route around the failure within the active slice. Instead, the control plane preempts the job and reconfigures the OCS fabric to attach the workload to spare healthy cubes elsewhere in the pod. This treats hardware as immutable infrastructure during a job's execution: rather than repairing the running topology, the system reconstitutes one with identical shape from surviving healthy components.

The quantitative success of this approach is measured in system availability. Across Google's production TPUv4 fleet, the dynamic reconfiguration strategy achieves 99.98 percent system availability while gracefully handling hardware outages experienced by approximately 1 percent of training jobs (Zu et al. 2024). Daily component failure rates are low in relative terms—0.08 percent of TPU machines, 0.005 percent of ICI cables, and 0.04 percent of OCS units—but at fleet scale they aggregate to a steady stream of reconfiguration events that the control plane must absorb without disrupting training throughput. The key lesson from the TPU experience is that for tightly coupled, high-bandwidth systems, attempting to repair a running topology is often futile; it is more efficient to treat the cube as the atomic unit of failure and reconfigure the optical fabric to assemble a fresh slice around the surviving healthy components.

7.12.3 Netflix chaos engineering for serving dependencies

While training resilience focuses on long-running batch jobs, Netflix's published work on Chaos Engineering (Basiri et al. 2016) provides the serving-side contrast. The premise is simple but operationally demanding: because production systems have many interacting dependencies, teams should validate steady-state behavior by deliberately injecting failures instead of waiting for real incidents to discover whether fallbacks work.

For ML serving, the same discipline applies to feature stores, model-serving dependencies, caches, and fallback paths. A model endpoint can be healthy while a feature service is slow, stale, or unavailable; if the system lacks a tested fallback, a local dependency problem can become user-visible latency or degraded recommendations. Chaos tests make those assumptions executable by injecting latency, terminating dependencies, or forcing fallback paths in controlled conditions.

The recovery procedure for these injected faults centers on fallback hierarchies. If the primary deep learning model fails or times out, the serving system can switch to a lighter model, cached response, or simple popularity-based default. The systems lesson is that resilience in ML serving is not a static property but a continuous practice of active verification: a fallback path that is never exercised is just documentation, not fault tolerance.

7.12.4 Microsoft DeepSpeed fault tolerance

Microsoft's DeepSpeed library illustrates a narrower framework-level lesson: large-model fault tolerance has to coordinate checkpointing, sharded optimizer state, and resource management instead of relying on a monolithic restart script. DeepSpeed provides ZeRO-based distributed training and checkpointing building blocks for models with more than 100 billion parameters (Rasley et al. 2020; DeepSpeed Developers 2026a). With the Zero Redundancy Optimizer (ZeRO), the model parameters, gradients, and optimizer states are sharded across all available GPUs rather than replicated (Rajbhandari et al. 2020). A single node failure can therefore strand the state shard it held, so recovery depends on durable checkpoints and orchestration that knows how to restore or redistribute those shards.

Rather than implying in-memory repair after every node loss, the supported recovery model is checkpoint-centered. Each rank writes its state shard as part of a distributed checkpoint, and after a failure the job reloads the last durable checkpoint under the worker topology chosen by the launcher and resource manager. Universal Checkpointing extends that model by making checkpoints more

portable across selected parallelism and topology changes, but it still relies on explicit save/load discipline (DeepSpeed Developers 2026b). If replacement capacity exists, the job can resume at the original scale; if the surrounding elastic launcher resumes with fewer workers, the training script must still preserve data coverage and batch-size or learning-rate invariants.

The systems lesson is therefore not that DeepSpeed alone makes worker loss transparent. It is that sharded training frameworks must expose checkpointing and state-management primitives that schedulers can compose into elastic recovery policies. Building those primitives into the framework layer reduces bespoke recovery engineering, but the end-to-end guarantee still comes from the combination of framework support, checkpoint discipline, and scheduler integration.

These case studies reveal three universal principles:

- **Detection speed determines recovery cost**: Meta's log-freshness monitoring, Google's automated ICI reconfiguration, and Netflix's chaos experiments all demonstrate that faster detection means less state lost.
- **The atomic unit of failure matters**: Google reroutes the communication fabric, DeepSpeed restores shards through checkpoints, and Netflix validates service-level fallbacks. Each organization chose the granularity that matches its architecture's coupling.
- **Fault tolerance is a spectrum, not a binary**: From Meta's checkpoint-rollback to Netflix's graceful degradation hierarchy, systems implement multiple layers of defense, each trading fidelity for speed.

Despite these patterns, engineering teams frequently stumble over common misconceptions when designing resilient ML systems.

7.13 Fallacies and Pitfalls

An infrastructure team might spend weeks hardening their storage layer against disk failures, only to have their entire training run destroyed by a subtle software bug in their PyTorch distributed backend. Fault tolerance for distributed ML systems involves counterintuitive mathematics and subtle trade-offs where conventional data center wisdom often fails.

Fallacy: *Hardware failures are the main concern.*

This intuition comes from traditional systems where disk failures, power outages, and network partitions dominate. In ML systems, hardware failures are only one part of a broader incident mix.

Industry experience from large-scale ML systems points to a broader failure mix because ML jobs are long-lived, stateful, and sensitive to small control-plane mistakes. Hardware failures still matter, but software bugs, configuration errors, resource exhaustion, and cross-layer causes often dominate the incident count: a malformed checkpoint path can make recovery impossible, a mismatched collective library can hang all ranks, an undersized shared-memory limit can crash data loaders, and a stale feature schema can silently corrupt training. These failures are not less serious because they are "software"; they can destroy the same multi-day job that hardware redundancy was meant to protect.

Investing heavily in hardware redundancy while neglecting software robustness (input validation, gradual rollouts, configuration management) leaves many important failure modes unaddressed. The most reliable ML systems treat software bugs as inevitable and design defensively.

Pitfall: *Setting checkpoint interval by intuition.*

Organizations commonly set checkpoint intervals based on "feels right": "every hour seems reasonable" or "every 1000 steps." The Young-Daly formula reveals these intuitions are often wrong. For a 1000-GPU cluster with MTBF of 4 hours and checkpoint time of 5 minutes:

$$\tau_{\text{opt}} = \sqrt{2 \times 5 \times 240} = \sqrt{2400} \approx 49 \text{ minutes}$$

The intuitive "every hour" is close but suboptimal. If checkpoint time increases to 15 minutes (larger model, slower storage), the optimal interval becomes 85 minutes, not the "every 15 minutes" that some teams adopt to "stay safe." Too-frequent checkpointing wastes more compute than it saves. The quantitative approach reveals that intuition-based intervals often deviate 2–3$\times$ from optimal in either direction.

Fallacy: *If each GPU is 99.99 percent reliable, a 10,000-GPU cluster is also 99.99 percent reliable.*

Reliability does not compose by averaging—it compounds by multiplication. For N serial components each with availability A, and assuming their failures are independent, the aggregate availability is A^N. With $A = 0.9999$ and $N = 10{,}000$, the cluster availability is $0.9999^{10{,}000} \approx 0.37$: the cluster is down 63 percent of the time. Individual component reliability is necessary but nowhere near sufficient at fleet scale. System-level fault tolerance (checkpointing, elastic recovery, redundancy across failure domains) must be designed explicitly, not assumed to emerge from per-component MTTF.

Pitfall: *Using MTBF calculations as if failures were independent.*

The reliability equation $\text{MTBF}_{\text{system}} = \text{MTBF}_{\text{component}}/N$ assumes component failures are statistically independent. In production, failures correlate:

- **Shared power domain**: UPS failure takes down an entire rack.
- **Shared switch**: Top-of-rack switch failure partitions all connected GPUs.
- **Shared software**: A bug triggered by specific input fails all replicas simultaneously.
- **Thermal correlation**: Cooling failure causes clustered GPU throttling.

Correlated failures change two quantities that the simple MTBF equation hides: incident frequency and blast radius. A cluster with 1000 independent GPUs each with 50,000-hour MTBF has a first-GPU-failure MTBF of 50 hours. If a shared power, cooling, or software hazard adds an incident rate 10 times larger than that independent first-failure rate, the effective incident MTBF drops to about 5 hours. Separately, a correlated incident may take out 10 GPUs at once, increasing recovery cost even if the independent GPU failure rate itself has not changed. Reliability engineering must identify and mitigate correlation through diversity: different power feeds, different network paths, different software versions in canary deployments.

Fallacy: *Component MTTF values predict individual failure timing.*

Engineers read a GPU MTTF of 50000 hours and plan around a five-year replacement cadence per device. MTTF is a statistical property of a population, not a prediction for any single component. A GPU with MTTF 50000 hours might fail at 200 hours or last 100,000 hours; the distribution is broad. What MTTF reliably predicts is the aggregate failure rate: in a fleet of 10,000 GPUs, the steady-state failure rate is approximately $10{,}000/50{,}000 = 0.2$ failures per hour, or roughly one failure every 5 hours. Fleet-scale reliability engineering relies on this statistical regularity to size automated recovery, spare-pool depth, and on-call rotations. Trying to predict which individual GPU will fail next is a category error and a waste of monitoring effort.

Pitfall: *Ignoring restart overhead in checkpoint planning.*

The Young-Daly formula accounts for checkpoint save time, but practitioners often forget restart overhead:

- **Job scheduling delay**: Acquiring replacement GPUs takes minutes in shared clusters.
- **Checkpoint loading**: Reading distributed checkpoint from storage.
- **Warmup time**: Learning rate warmup, batch normalization statistics recalculation.
- **Communication re-establishment**: NCCL ring topology reconstruction.

Total restart time can be 3–5$\times$ checkpoint save time. A 5-minute checkpoint save followed by a 20-minute restart means each failure costs 25 minutes before accounting for lost work since the last checkpoint. That recovery term should be included in the expected waste and recovery-SLO budget; it should not be added to T_{write} as though it were paid at every checkpoint unless a specific recovery-aware checkpoint model derives that dependency.

Fallacy: *Every failure should be handled as a full restart.*

Engineers often treat all failures as "node crashed, restart from checkpoint," but different failure modes require different responses. Transient failures (network congestion, thermal throttling) should trigger retry/pause, not restart, since state remains in memory. Permanent failures (GPU death, node crash) require checkpoint-restart with migration to new hardware. Silent corruption (bit flips, ECC errors) demands rollback to a previous checkpoint, not just the latest one, requiring checkpoint history retention. Resource exhaustion (OOM, memory fragmentation) needs reconfiguration before restart; otherwise the job crashes again immediately. As section 7.1.5 details, misdiagnosing failure type wastes compute: treating transient network blips as permanent failures wastes hours re-initializing, while ignoring silent corruption poisons model weights undetected.

Pitfall: *Assuming checkpoints are consistent without validating state.*

Modern frameworks checkpoint transparently, creating the illusion of automatic consistency. In practice, distributed checkpoints require coordination that can fail subtly:

- **Rank desynchronization**: If rank 0 checkpoints iteration 1000 while rank 1 checkpoints iteration 1001, the checkpoint is inconsistent.
- **Partial writes**: Storage failure mid-checkpoint leaves incomplete shards.
- **Optimizer state lag**: Sharded optimizer state may not match model weights if captured at different times.
- **In-flight gradients**: AllReduce in progress during checkpoint may or may not be included.

Production systems must implement checkpoint validation: verify all shards exist, verify iteration numbers match, verify optimizer state matches model state. Organizations that discover corrupted checkpoints during recovery from a failure have no recourse except restarting from an earlier (potentially much earlier) checkpoint.

Fallacy: *Fault tolerance can be tested only when failures happen.*

Fault tolerance mechanisms are code paths that execute rarely in normal operation. Like backup systems never tested until disaster strikes, fault tolerance code paths accumulate bugs:

- Checkpoint restoration logic untested because training never crashed
- Fallback model never loaded because primary never failed
- Circuit breaker thresholds tuned for old traffic patterns

Chaos engineering (intentionally injecting failures) transforms fault tolerance from "we think it works" to "we know it works." Organizations that regularly kill random GPUs during training, inject network partitions, and fail primary models discover bugs before they matter. The cost of regular fault injection (some failed experiments, some minor outages) is far less than the cost of discovering broken fault tolerance during an actual failure.

Pitfall: *Using elastic training as a substitute for checkpointing.*

Elastic training adjusts parallelism degree when workers fail, continuing with reduced capacity, which appears to eliminate checkpoint-restart overhead. However, state consistency challenges remain: reducing the active worker count requires redistributing model shards, optimizer states, and data assignments consistently. Below some minimum viable size, training becomes infeasible (model does not fit, batch size too small), requiring checkpoint-restart regardless. Each removed worker reduces throughput; accumulated failures progressively degrade training speed until checkpoint-restart becomes preferable to continued degradation. If a failure was caused by a software bug triggered by specific data, the bug persists in remaining workers. Elastic training is complementary to checkpointing, not a replacement; the reduced checkpoint frequency still requires occasional checkpoints for catastrophic failures and training completion.

Fallacy: *Overhead budgets are fixed fractions of training time.*

Reference tables that show "pipeline bubble: 10 percent, checkpoint overhead: 5 percent, failure recovery: 3 percent" are easy to read as physical laws and pass through as line items in capacity plans. They are engineering targets, not constants. Pipeline bubble overhead depends on the number of microbatches and the interleaved schedule; failure-recovery overhead drops dramatically with elastic training that avoids full restarts; checkpoint overhead is a function of model size, storage bandwidth, and whether checkpointing is synchronous or asynchronous. Treating these numbers as fixed percentages leads to passive acceptance of avoidable inefficiency. The Young-Daly formula already shows that checkpoint cadence is a decision; the same is true of every other overhead listed in the reference tables.

Pitfall: *Adding GPUs without accounting for communication and recovery overhead.*

Capacity plans typically account for Amdahl's Law and AllReduce overhead but treat reliability as a fixed background condition. At extreme scale, each additional GPU increases the aggregate failure rate, which inflates the expected recovery and replay time per job. There exists a cluster size beyond which the time lost to failures and recovery exceeds the time saved by additional parallelism, and wall-clock training time increases rather than decreasing with N. This is the reliability version of diminishing returns and it applies on top of the Amdahl ceiling. The two effects

must be modeled together: useful goodput equals MFU $\times$ scaling efficiency $\times$ (1 - failure overhead), and the failure-overhead term grows with N.

Fallacy: *Silent data corruption is negligible because hardware has ECC.*

GPUs and memory systems include extensive error correction (ECC, CRC, parity), creating the intuition that silent data corruption is negligible. Large-scale CPU SDC and memory-error studies reveal otherwise: silent corruptions and memory faults still occur at fleet scale and can escape ordinary error reporting (Dixit et al. 2021; Sridharan et al. 2015). For a 10,000-GPU cluster, even rare per-device corruption events become operationally relevant because the system continuously executes enormous volumes of memory and arithmetic operations. Silent corruption causes mysterious training anomalies: loss spikes attributed to "bad batches" may be hardware errors, gradient NaNs blamed on learning rates may be bit flips, and models failing to converge despite correct hyperparameters may have corrupted weights. Detection strategies include redundant computation (computing batches on multiple workers and comparing), gradient checksums (verifying AllReduce consistency), and statistical monitoring of gradient/activation distributions. Unlike detectable failures, silent corruption does not trigger errors; training "succeeds" but produces subtly broken models, requiring detection mechanisms as discussed in section 7.11.3.

Pitfall: *Relying only on hardware ECC instead of end-to-end validation.*

ECC is a necessary layer, but it is not an end-to-end correctness proof for a training run. The pipeline also needs checksums on data shards, validation of checkpoint shards, gradient and activation anomaly detection, and replayable tests that can distinguish a bad batch from corrupted state. End-to-end validation makes the silent-corruption problem observable at the ML-system boundary, where the damage would otherwise appear only as unexplained training behavior.

Recognizing these fallacies prevents engineers from optimizing for the wrong failure modes. The core principles required to build resilient machine learning fleets follow from the quantitative reasoning developed throughout this chapter.

7.14 Summary

Fault tolerance transforms the statistical certainty of hardware failure from a project-ending catastrophe into a manageable operational routine. The mathematics are unforgiving: individual component reliability compounds multiplicatively across thousands of devices, driving system-level MTBF from years down to hours. At large fleet scale, failure is not an exceptional event to be debugged but a continuous background condition that systems must absorb without losing forward progress. The engineering challenge, therefore, is not to prevent failures but to build systems where recovery is automatic, fast, and invisible to the training or serving workload.

Checkpointing provides the foundational mechanism for preserving training progress across failures. Synchronous checkpointing offers simplicity but imposes I/O overhead that scales with model size, while asynchronous approaches overlap checkpoint writes with computation at the cost of additional consistency complexity. The Young-Daly formula, $\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}}$, gives engineers a principled way to balance checkpoint frequency against overhead; depending on checkpoint size and cluster MTBF, the optimum can range from seconds for small models to tens of minutes for large jobs. Beyond basic checkpointing, elastic training breaks the rigid assumption that worker count must remain fixed: when nodes fail, the system redistributes data and model shards across the surviving workers, adjusts batch size and learning rate, and resumes training with reduced throughput rather than halting entirely.

Serving fault tolerance presents a fundamentally different challenge from training. Training tolerates minutes of recovery latency and benefits from SGD's mathematical tolerance of approximate restarts, while serving demands millisecond-level responsiveness and must preserve per-session state such as KV caches and conversation histories. Stateless serving achieves fault tolerance through straightforward replica redundancy and load balancing, but stateful serving for LLMs requires active state replication, session-affine routing, and graceful degradation hierarchies that fall back to lighter models when primary systems are unavailable. The case studies examined in this chapter, from Meta's OPT-175B training through roughly 90 restarts driven by hardware failures and training-stability incidents to Netflix's chaos engineering for ML serving, demonstrate that these principles are not theoretical but operational necessities at production scale.

Engineers who internalize these principles gain a diagnostic framework for reasoning about resilience at any scale. When a training run stalls, they can immediately assess whether the bottleneck is checkpoint I/O overhead, insufficient detection speed, or a failure mode that elastic training cannot absorb. When a serving system drops requests, they can trace the fault through the redundancy hierarchy to determine whether the root cause is replica health, state replication lag, or an inadequate fallback strategy. This systematic reasoning distinguishes organizations that treat fault tolerance as an afterthought from those that engineer it as a first-class system property. As ML systems scale, the cost of unplanned downtime grows proportionally: a 10,000-GPU cluster idled for an hour represents tens of thousands of dollars in wasted compute, making the techniques developed in this chapter economic necessities rather than optional refinements.

Key Takeaways: Failure is normal operation

- **Scale makes failure routine**: A 10,000-GPU cluster encounters hardware failures every few hours under the chapter's component-rate assumptions; software must treat failure as a normal state.
- **Checkpointing is the baseline**: Synchronous checkpointing is simple but incurs high overhead; asynchronous approaches hide I/O latency at the cost of consistency complexity.
- **The Young-Daly formula governs checkpoint intervals**: The optimal checkpoint interval $\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}}$ balances the cost of saving state against the cost of lost work, often landing in the tens-of-minutes range for large training jobs while remaining much shorter for small checkpoints.
- **Elasticity enables persistence**: Designing training jobs to be "elastic" (reconfiguring around lost nodes, as figure 7.28 illustrates) can reduce idle time and lost work when the training script preserves the necessary data, batch-size, learning-rate, and checkpoint invariants.
- **Stateful serving exposes the serving-side state problem**: For LLMs, the KV cache represents a massive serving state that must be replicated or migrated to prevent high-latency session restarts, with strategy trade-offs summarized in table 7.13.
- **Training and serving require different strategies**: Training fault tolerance is checkpoint-centric and tolerates minutes of recovery, while serving fault tolerance is state-migration-centric and demands sub-second failover.
- **Production scale validates the theory**: The chapter's training and serving case studies show that large systems only stay reliable when fault tolerance is continuously exercised rather than assumed.

If failure cannot be prevented, the only move left is to make it cheap, and that is what turns resilience from a wish into a calculation. Checkpointing spends compute and communication on nothing a user ever asked for, copies of state held only against the day a node dies, and the Young-Daly interval is the arithmetic of how much to spend: not how to avoid losing work, but how much work is affordable to lose between saves. Set it well and a dead node costs minutes, not days. The failures still arrive exactly as often; what changes is that they stop deciding the outcome. This is the coordination tax made explicit, paid on purpose so that at fleet scale, where component failure is a statistical certainty, the certainty stops mattering.

What's Next: From resilience to resource management

Keeping the fleet running despite inevitable hardware failures requires three layers: checkpointing preserves progress, elasticity absorbs node losses, and redundancy protects stateful serving sessions. A fault-tolerant cluster still needs orchestration, however, to decide which jobs run where, when to preempt, and how to share resources fairly among competing workloads.

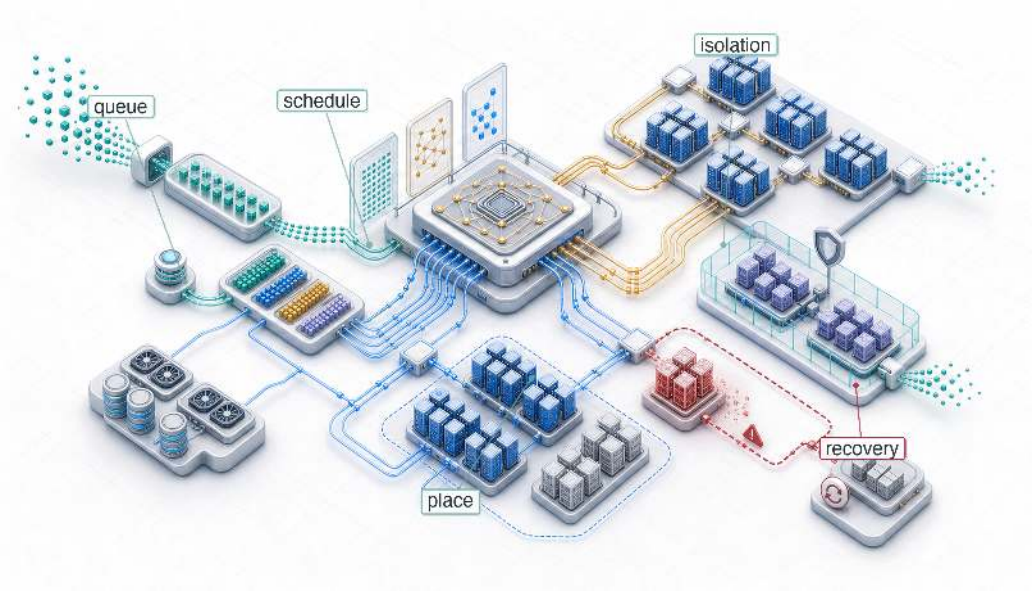
Chapter 8 examines the orchestration layer, from Slurm and Kubernetes scheduling to gang scheduling, multi-tenancy, and capacity planning, that transforms a collection of fault-tolerant nodes into a managed production system.

Research Questions: For further inquiry

- How should checkpoint cadence account for MTBF, checkpoint write time, restart overhead, and validation cost together?
- How do parallelism and collective choices change the blast radius of a single node, link, or rank failure?
- When is elastic recovery preferable to full restart, and which training invariants must it preserve?
- What end-to-end evidence can detect silent data corruption before it appears as degraded model behavior?

8

Fleet Orchestration

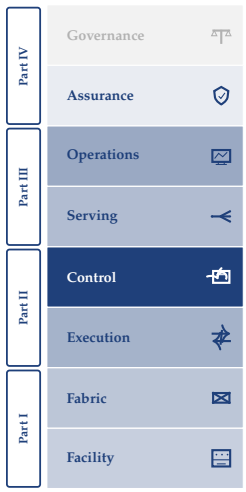


8.1 The Scheduling Problem
8.2 Orchestration Paradigms
8.3 Topology-Aware Scheduling
8.4 Elastic Scheduling
8.5 Cost Optimization
8.6 Custom ML Schedulers
8.7 Serving Resource Management
8.8 Multi-Tenancy and Quotas
8.9 Debugging Cluster Utilization
8.10 Fallacies and Pitfalls
8.11 Summary

Purpose

Why does resource allocation become the primary bottleneck when hardware is plentiful?

A thousand accelerators sitting idle waiting for scheduling decisions cost the same as a thousand accelerators computing useful work. At scale, the limiting factor shifts from having enough hardware to using it efficiently: jobs waiting in queues while resources sit idle, fragmentation leaving gaps too small for any pending job, deadlocks where multiple jobs each hold partial resources while waiting for more. Orchestration is the discipline of extracting useful work from shared infrastructure, deciding which jobs run on which nodes, when preemption serves the greater good, how to balance fairness across teams against raw utilization, and how to prevent the coordination mechanisms themselves from becoming bottlenecks. Poor orchestration transforms expensive hardware into expensive waste; effective orchestration transforms a collection of machines into a coherent computing resource where capacity translates reliably into completed work. In C^3 terms, orchestration is the coordination tax paid on purpose: theoretical peak utilization is traded away so that the fleet's shared resources do not deadlock or fragment under load.



Learning Objectives

- Analyze scheduling objectives, fragmentation, and deadlock risks for distributed training jobs under shared fleet constraints
- Compare HPC, cloud-native, and hybrid orchestration paradigms for batch training and self-healing serving workloads
- Apply gang, bin-packing, and topology-aware placement policies across NVLink domains and switch hierarchies
- Design elastic training and preemption policies that trade throughput, recovery overhead, and spot-instance savings
- Evaluate ML-specific schedulers using job duration, iteration progress, fairness, and adaptive resource-allocation signals
- Design inference autoscaling, isolation, and routing policies using queue depth, tail latency, and GPU memory pressure
- Diagnose utilization losses from quota hoarding, noisy neighbors, fragmentation, topology mismatch, and data starvation

8.1 The Scheduling Problem

Imagine two research teams sharing a 1,000-GPU cluster: Team A submits a 512-GPU job that will run for a month, while Team B submits hundreds of 8-GPU experiments that each run for an hour. If the scheduler simply processes jobs as they arrive, Team B's tiny experiments might indefinitely block Team A's massive training run. The scheduling problem is the challenge of navigating this multi-dimensional conflict between throughput, fairness, and cluster utilization.

Partitioning computation, synchronizing it efficiently, and recovering from hardware failures solve the problem of running a single job across many machines. Orchestration solves the next problem: deciding which jobs run, where they run, and how resources are shared among competing demands.

In the fleet stack shown in figure 1.13, orchestration operates at the Distribution Layer, but its decisions are fundamentally constrained by the Infrastructure Layer. The scheduler must reason about physical realities: GPU heterogeneity, NVLink topology, rack power limits, and network bisection bandwidth. When scheduling goes wrong, debugging requires examining all four layers to identify whether the root cause is infrastructure (hardware constraints), distribution (algorithm limitations), serving (workload mismatch), or governance (policy misconfiguration). Orchestration is where physical infrastructure meets the demands of distributed algorithms, translating resource requests into placement decisions that respect both hardware topology and organizational policy.

To make the scheduling problem concrete, consider a research organization operating a 10,000-GPU cluster. Brown et al. (2020) report that training GPT-3 175B used 3.14×10^{23} floating-point operations on a V100 GPU cluster. Imagine 100 large-scale training jobs of that class queued, alongside thousands of smaller experiments and inference workloads all competing for the same resources. The system must decide which jobs run, when they run, and where they run. A naive first-come-first-served policy would let the first few large jobs monopolize the cluster for weeks, starving smaller experiments that researchers need to iterate quickly. A strict fair-share policy would fragment GPUs across many small allocations, preventing any large job from assembling the contiguous allocation it needs. Neither extreme works. The scheduler must navigate a multi-dimensional trade-off space where every decision affects throughput, fairness, cost, and researcher productivity simultaneously.

The economic stakes make these decisions consequential at every scale. A 10,000-GPU cluster at $2/GPU-hour costs $480,000 per day to operate, whether the GPUs are computing useful work or sitting idle in a queue. If scheduling inefficiencies leave 30 percent of GPUs idle, that translates to $144,000 per day in wasted capacity, or about $52.6M annually. Conversely, improving utilization from 50 percent to 80 percent adds 3,000 fully utilized GPU-equivalents of productive capacity without purchasing additional hardware. That gain is equivalent to buying 6,000 raw GPUs if they ran at the old 50 percent utilization. At this scale, a 1 percentage-point improvement in utilization is

worth more annually than the salary of the engineer who achieves it. *Scheduling is not operational overhead; it is one of the highest-leverage engineering investments in ML infrastructure.*

ML workloads present scheduling challenges that distinguish them from ordinary cloud service placement and from HPC workloads that can tolerate flexible resource counts. Gang scheduling[1] represents the most critical difference for synchronous distributed training: a job requiring 1,024 GPUs *cannot* make useful iteration progress with only 512. Many HPC jobs also need co-scheduled ranks, but synchronous data parallel training exposes the cost at every AllReduce. Every worker must participate in every collective, and a missing worker blocks all others. Section D.2.1 develops the ring and tree AllReduce cost model that makes this constraint physical: the collective stalls until the last worker arrives, so allocating N GPUs but landing $N-1$ leaves the missing rank's peers burning fully idle on the synchronization barrier. A scheduler that partially allocates resources creates deadlocks where multiple jobs each hold some GPUs while waiting for more, with none able to proceed. This all-or-nothing requirement means the scheduler cannot simply "pack jobs tightly" as a traditional bin packer would; it must reason about atomic, multi-resource allocations across the entire cluster.

If gang scheduling is the first hard constraint, topology awareness via **placement groups**[2] is the second: to support tensor parallelism, the scheduler must ensure GPUs are placed within the same high-bandwidth NVLink domain rather than scattered across the data center.

GPU heterogeneity is the third dimension. Many clusters contain mixtures of A100 and H100 accelerators with different compute throughput, comparable HBM capacity (80 GB on A100 versus 80 GB on H100), and different interconnect bandwidth. An H100 provides roughly twice the training throughput of an A100 for transformer workloads but costs proportionally more. Some models fit in A100 memory with appropriate parallelism strategies; others require the larger H100 memory to avoid excessive model sharding. The scheduler must match workloads to appropriate hardware based on actual requirements, not user preferences, while maintaining high utilization across heterogeneous pools. When users request specific GPU types out of habit rather than necessity, the mismatch between requested and required resources can leave entire hardware pools idle while queues overflow on preferred hardware.

Job duration unpredictability compounds these challenges further. A training run may converge early and complete in days rather than weeks. Hardware failures may abort jobs unexpectedly, returning resources at unpredictable times. Hyperparameter searches may reveal early that certain configurations will not succeed, leading to voluntary termination. Traditional scientific computing workloads, such as weather simulations or molecular dynamics, have more predictable runtimes that enable better scheduling decisions: the scheduler can project when resources will become available and plan accordingly. ML workloads deny this luxury, forcing schedulers to operate with deep uncertainty about when current jobs will release resources.

The interaction between these challenges creates combinatorial complexity. A scheduler must simultaneously handle gang scheduling constraints (atomic allocation), heterogeneous resources (capability matching), topology requirements (NVLink locality for tensor parallelism), priority and fairness policies (organizational equity), preemption decisions (which running jobs to interrupt), and cost optimization (spot vs. on-demand placement). Each dimension constrains the others: a topology-optimal placement may violate fairness policies, a cost-optimal spot placement may violate reliability requirements, and a fairness-driven allocation may fragment resources in ways that prevent gang scheduling.

These constraints define the central problem of orchestration: the scheduler is not merely packing jobs onto GPUs, but maintaining a consistent, economically useful view of a failing, heterogeneous fleet. The first step is to see why cluster scheduling is fundamentally harder than single-machine process scheduling. At fleet scale, placement stops being only an optimization challenge and becomes a distributed systems problem of keeping state coherent across thousands of machines.

8.1.1 Distributed scheduling complexity

A single-machine scheduler enjoys luxuries that a cluster scheduler cannot: it has instantaneous, consistent visibility into all resource states; it can make atomic decisions that take effect immediately; and failures are binary (the machine is up or it is down). Cluster scheduling surrenders all three of these properties, and several fundamental distributed systems problems emerge as a result.

1 | **Gang Scheduling**: Formalized by Ousterhout (1982) for parallel systems, where the "Ousterhout matrix" co-schedules related threads across processors in synchronized time slices. In ML clusters, the constraint is stricter: synchronous AllReduce requires all workers simultaneously, so partial allocation wastes 100 percent of held resources rather than merely degrading throughput. Under the chapter's cloud-equivalent $2/GPU-hour scenario, a 1,024-GPU job holding 900 idle GPUs while waiting for 124 more burns approximately $1,800/hour.

2 | **Placement Group**: A cloud-native abstraction (AWS, GCP) that requests instances be placed within a single high-bandwidth, low-latency network domain. For the ML Fleet, placement groups act as the topology contract that guarantees the bisection bandwidth (BW_{bisect}) required for AllReduce, preventing the "topology lottery" where nodes are scattered across different data center racks.

Partial failures pose the first challenge. A node can fail between allocation and job start, creating a gap between the scheduler's decision and its effect. The scheduler may successfully allocate 32 GPUs across 4 nodes, only to have one node fail before the job launches. The remaining 24 GPUs sit idle while the scheduler detects the failure, updates its state, and re-plans the allocation. Meanwhile, other jobs that could have used those 24 GPUs have already been placed elsewhere, potentially on suboptimal hardware. The fault tolerance mechanisms from Chapter 7 handle failures during *execution*; the scheduler must handle failures during *placement*, a fundamentally different problem because the job has not yet established any state to recover from.

Network partitions create a second problem that is unique to distributed scheduling. The scheduler may lose connectivity to a subset of nodes while those nodes continue operating normally. From the scheduler's perspective, the nodes appear failed and their GPUs appear unavailable. From the nodes' perspective, jobs may still be running and producing useful work. This ambiguity creates a dilemma with no universally correct resolution. Reallocating the GPUs to new jobs risks double-allocation if the partition heals; waiting for reconnection wastes capacity that may be perfectly functional. The duration of the partition is unknowable in advance, so any fixed timeout represents a guess about network behavior that may prove wrong.

State inconsistency emerges as a third challenge that compounds the first two. Resource state may differ between the scheduler's view and reality on individual nodes. A GPU the scheduler believes is free may still be cleaning up from a previous job: flushing caches, deallocating device memory, running a zombie process from a failed container, or completing a CUDA driver reset after an error. Conversely, a GPU marked as "in use" may have been freed by a job that terminated between heartbeat intervals. This inconsistency means the scheduler operates on a model of cluster state that is always slightly stale, and the degree of staleness varies across nodes depending on heartbeat frequency, network latency, and the complexity of cleanup operations.

Ordering without global time presents the fourth fundamental issue. Without a global clock, determining whether allocation A happened before allocation B across different nodes requires careful protocol design using logical clocks or consensus algorithms. Two jobs may both believe they "own" the same GPU if the system does not enforce ordering through consensus protocols or centralized coordination. This scenario is not theoretical: during recovery from a network partition, allocation messages from before and after the partition may arrive in the wrong order at different nodes, creating conflicting resource assignments that must be detected and resolved.

These four challenges echo the consistency, availability, partition-tolerance trade-off described by the CAP theorem[3]. CAP is a theorem about distributed data systems, not a literal proof about every scheduler, but the analogy is useful: a control plane must choose how much stale or uncertain state it is willing to tolerate before making allocation decisions. Slurm-style centralized batch control emphasizes an authoritative allocation view before jobs start. Kubernetes-style reconciliation emphasizes continuously driving the actual cluster toward declared desired state. Custom ML schedulers often accept bounded inconsistency in exchange for scheduling throughput, using optimistic concurrency where conflicts are detected and resolved after the fact rather than prevented through locking.

[3] **CAP Theorem**: Conjectured by Brewer (2000) and proven by Gilbert and Lynch (2002), CAP establishes that no distributed data system can simultaneously guarantee Consistency, Availability, and Partition tolerance. For ML schedulers, use CAP as a design analogy rather than a product claim: the concrete policy question is how much uncertainty the control plane accepts before it allocates, reclaims, or reschedules expensive accelerators.

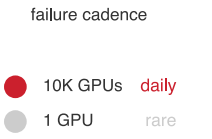
The control-plane tension becomes concrete when the scheduler interacts with the ML framework's own distributed coordination plane during a partition. A scheduler may decide that a subset of nodes is unavailable while workers on those nodes are still blocked inside a collective operation. That creates a brief window of conflicting intent: the scheduler wants to recover the allocation, but the training job has not yet converged on a new membership view. Elastic launch and rendezvous mechanisms such as TorchElastic reduce this ambiguity by using heartbeats and re-rendezvous so the framework can either reform a viable worker group or terminate cleanly, giving the scheduler a clearer recovery signal than an opaque collective timeout (PyTorch Contributors 2026b, 2026a). This layered coordination—scheduler-level policy for coarse allocation, framework-level membership for workers—is the characteristic architecture of production distributed training clusters.

8.1.2 Failure rates at scale

At scale, failure is normal operation, not exceptional. This principle, established in the reliability analysis of Chapter 7, has direct consequences for scheduling: the scheduler must not merely tolerate failure but actively plan for it in every allocation decision. Component reliability does not change

with cluster size, but aggregate system reliability degrades multiplicatively. Counting GPU hardware faults alone understates the rate; with 99.9 percent annual GPU reliability (typical for data center hardware), the hardware-only expectation for a 4,096-GPU cluster is modest:

$$\text{Expected failures per day} = 4096 \times \frac{0.001}{365} \approx 0.01 \text{ GPU failures/day}$$



Normal component failure rates make fleet failures routine.

This calculation captures only GPU hardware failures. Including software failures (driver crashes, CUDA context corruption), thermal events (throttling, emergency shutdowns), network interface failures, host OS issues, and container runtime errors, large clusters can see 1 to 4 failures per day per 1,000 GPUs. A 10,000-GPU cluster would then experience 10 to 40 component failures daily. A multi-week training run on 4,096 GPUs is therefore very likely to encounter multiple failures. Section E.1.1 tabulates the FIT rates and MTTF figures behind the 99.9 percent annual reliability assumption, letting an operator substitute measured hardware rates and confirm whether the empirical 1-to-4 failures-per-day band still holds.

The scheduling implications are profound and shape nearly every design decision in the scheduler. The scheduler cannot treat the cluster as a static resource pool where resources, once allocated, remain available until voluntarily released. Instead, it must anticipate that allocated resources will disappear during job execution, maintain spare capacity or implement rapid rescheduling to keep jobs running through the constant churn of hardware entering and leaving operational status, and distinguish between transient failures (which may self-resolve within minutes and should not trigger expensive reallocation) and permanent failures (which require new resource allocation and checkpoint recovery) (Tiwari et al. 2015).

Consider the scheduler's decision when a node heartbeat is missed. The scheduler has three options:

- **Wait for recovery**: The scheduler waits to see if the heartbeat resumes, risking wasted GPU time if the node is truly dead.
- **Reallocate immediately**: The scheduler assigns the node's resources to other jobs, risking conflict if the node recovers and its original jobs are still running.
- **Mark suspect**: The scheduler marks the node as suspect and begins prepositioning replacement resources while waiting for confirmation, consuming spare capacity that could be used for other work.

Each option trades off between responsiveness and correctness, and the effective choice depends on the failure mode distribution for the specific hardware in the cluster, information the scheduler must learn from historical failure data. Section E.3.1 decomposes recovery time into its detect, reschedule, reload, and replay phases, which bound how long the scheduler can afford to wait on a missed heartbeat before committing to reallocation rather than betting on a transient self-resolution.

These requirements connect the scheduler directly to the checkpoint and recovery infrastructure from Chapter 7: scheduling decisions about spare capacity, preemption policies, and elastic training support determine how quickly the system recovers from statistically expected failures. A scheduler that reserves spare capacity or supports elastic recovery can restore throughput much faster than one that requires every failed job to wait for a full fresh allocation and reload, but the correct reserve fraction and recovery time are workload- and fleet-specific parameters rather than universal constants.

8.1.3 Scheduling objectives and their conflicts

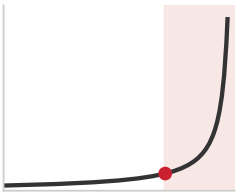
Every scheduling decision represents a trade-off between four fundamental and often contradictory objectives:

- **Throughput**: The total useful work completed per unit time naturally favors large, long-running jobs that saturate hardware and minimize context-switching and data-movement overhead.
- **Fairness**: The equitable distribution of resources across users or teams prevents a single large team from monopolizing the fleet. Dominant resource fairness (Ghodsi et al. 2011) equalizes each user's largest normalized share across resources such as GPUs, CPU, and memory.

- **Latency**: Rapid completion of short jobs maintains researcher velocity, but aggressive preemption can starve large training jobs.
- **Cost efficiency**: Interruptible spot instances, off-peak scheduling, and tight packing reduce budget burn, but often increase failure rates or completion time.

The objectives conflict because each one rewards a different scheduling behavior. A throughput-maximal policy fills the cluster with massive training runs, achieving near-perfect utilization but forcing every other user to wait weeks for a slot. Fairness-driven allocation fragments resources, leaving small gaps that cannot be filled by large jobs and sacrificing aggregate throughput for social harmony. A latency-optimal scheduler, similar to Shortest Job First, aggressively preempts long-running training jobs to service interactive debugging sessions or small experiments. Cost optimization often means accepting higher failure rates and longer completion times, trading researcher productivity for budget preservation.

Optimizing all four simultaneously is impossible; every scheduling policy represents a specific point in this four-dimensional trade-off space. The most direct conflict exists between throughput and latency, a tension analogous to the throughput-latency trade-off in computer architecture. A throughput-optimal scheduler prioritizes the largest, most parallelizable jobs because they use the hardware most efficiently. However, this policy is latency-catastrophic for small jobs: a researcher submitting a 10-minute debugging task might wait days for the large job to finish. Conversely, a latency-optimal scheduler minimizes average wait time but potentially starves large jobs indefinitely, reducing aggregate cluster throughput by leaving large blocks of resources idle while waiting for "just one more" small job to finish.



Past about 70 percent utilization, queue wait time explodes.

Consider a 64-GPU fine-tuning run for our 175B model, a scheduling-demo workload distinct from the full 1,024-GPU pretraining run that recurs elsewhere in the fleet discussion. From a throughput perspective, this is an ideal job: it runs for days with high utilization and zero scheduling overhead once started. From a latency perspective, it is a boulder in the stream. To schedule it, the system might need to drain 64 GPUs of all other work, forcing hundreds of smaller jobs to wait. Once running, it occupies those resources immovably. If the scheduler prioritizes this run (throughput), the P99 latency for small jobs explodes. If it prioritizes small jobs (latency) by allowing them to preempt or fragmentation-fill the cluster, the 64-GPU job may never assemble the contiguous block it needs to start. This zero-sum game forces organizations to make explicit policy choices. In scheduler terminology, those choices often appear as Quality of Service classes: job-priority tiers that decide which objective wins when throughput, latency, fairness, and cost collide.

Napkin Math 8.1: The queuing theory of GPU clusters

A GPU cluster can be modeled as an M/G/1 queue, where jobs arrive according to a Poisson process (λ_{job}) and service times follow a general distribution (G) with mean $1/\mu$ and standard deviation σ. The Pollaczek-Khinchine formula defines the expected waiting time in the queue (W_q):

$$\frac{W_q}{1/\mu} = \frac{\rho}{1-\rho} \cdot \frac{1+C_s^2}{2}$$

Here, $\rho = \lambda_{\text{job}}/\mu$ represents cluster utilization, $1/\mu$ is the average job duration, and $C_s = \sigma\mu$ is the **coefficient of variation** of job duration. The right-hand side is therefore the waiting time expressed as a multiple of the average job duration.

In standard web serving, request service times are often approximated by an exponential distribution, yielding $C_s \approx 1$. In ML clusters, job durations follow a heavy-tailed distribution: a vast number of short debugging jobs (minutes) mixed with a few massive training runs (weeks). The representative $C_s = 3$ calculation in this callout captures that variance without claiming a universal fleet constant. The impact on wait times is multiplicative. For the scheduling argument here, the single-server multiplier is sufficient; section F.1.1 generalizes the result to the M/G/c/K queue and validates it against Little's Law for readers sizing a scheduler's admission queue under heavy-tailed arrivals.

At 80 percent utilization ($\rho = 0.8$): with an exponential workload ($C_s = 1$), $W_q = 4 \times$ the average job duration. With a typical ML workload ($C_s = 3$), $W_q = 20 \times$ the average job duration. This explains the "utilization wall" in ML infrastructure. A web server cluster feels responsive at 80 percent load, but an ML cluster at the same utilization feels broken, with jobs languishing in the queue for days. The heavy tail of the service distribution acts as a latency multiplier, forcing operators to run ML clusters at lower utilization (often 60 to 70 percent) to maintain acceptable responsiveness for interactive users.

8.1.4 Bin packing

The most fundamental scheduling algorithm is **bin packing**[4]: fitting jobs of varying sizes into fixed-capacity nodes. This problem is NP-hard in its general form, meaning no known algorithm finds optimal solutions in polynomial time. Fortunately, practical heuristics such as first-fit decreasing and best-fit decreasing achieve near-optimal results for the workload distributions typical of ML clusters, where job sizes follow a heavy-tailed distribution (many small jobs, few large ones).

Consider a 64-node cluster with 8 GPUs per node, totaling 512 GPUs. If jobs request 6 GPUs each, each job occupies one full node but wastes 2 GPUs per node, reducing effective capacity to 75 percent. The remaining 2 GPUs per node cannot be combined across nodes because GPU workloads require local memory access. This **fragmentation** grows worse with heterogeneous job sizes: a mix of 1-GPU, 3-GPU, and 7-GPU jobs creates irregular gaps distributed across many nodes, where no single pending job can fit into any individual gap, yet the total free capacity would be sufficient if the gaps were contiguous.

ML workloads make bin packing multi-dimensional in ways that traditional scheduling rarely encounters. Each job requires GPUs together with CPU cores for data preprocessing, host memory for data loading and augmentation pipelines, local SSD storage for dataset caching, and network bandwidth for gradient synchronization. A job requesting 4 GPUs, 32 CPU cores, and 256 GB of RAM may not fit on a node that has 4 free GPUs but only 16 free CPU cores because other jobs have consumed the host CPU for preprocessing. The scheduler must simultaneously satisfy all resource dimensions, and the job fits only if every dimension has sufficient capacity on the selected node. This multi-dimensional constraint dramatically reduces the solution space compared to single-dimensional packing, because a bottleneck in any single resource dimension can strand capacity in all others.

Locality constraints further restrict placement beyond simple resource availability. A training job using tensor parallelism requires GPUs connected via NVLink within the same node, since cross-node communication over InfiniBand is an order of magnitude slower (450 GB/s per direction intra-node vs. 50 GB/s per port inter-node). A job requesting "4 GPUs with NVLink connectivity" cannot use 2 GPUs from node A and 2 from node B, even if all 4 GPUs are individually available and the total capacity is sufficient. These topology constraints transform a packing problem into a placement problem where which specific resources matter as much as how many resources are available. The placement problem is strictly harder than the packing problem because it adds spatial constraints to the existing capacity constraints.

Schedulers address fragmentation through several complementary strategies, each targeting a different aspect of the problem. Backfill scheduling allows smaller jobs to fill gaps while larger jobs wait for contiguous resources, improving utilization without violating priority ordering. The insight is that small jobs can execute and complete in the gaps, freeing those resources before the large job's target start time. Backfill scheduling requires estimating when current jobs will complete (to determine whether a backfill candidate will finish before the large job can start), which is why accurate runtime estimates are so important.

Periodic **defragmentation** migrates or preempts low-priority jobs to consolidate free resources into contiguous blocks, analogous to memory compaction in operating systems. The scheduler identifies nodes where partial allocations leave stranded resources, preempts the jobs occupying those resources, and re-schedules them on nodes where they can be packed more efficiently. The cost of defragmentation (preempting running work, which wastes compute between the last checkpoint and the preemption event) must be weighed against the benefit (enabling large jobs to start sooner,

4 **Bin Packing**: The one-dimensional version is NP-hard; ML scheduling adds four to five dimensions (GPU, CPU, memory, network, topology), making exact solutions intractable for clusters above a few hundred nodes. First-fit-decreasing heuristics achieve within 11/9 of optimal for typical workloads, but the real cost in ML clusters is not *suboptimality* in packing but *fragmentation*: stranded GPUs that individually satisfy no pending job yet collectively represent millions of dollars in idle capacity.

improving overall throughput). Operators often schedule defragmentation during low-demand periods when the impact of preemption is lower.

A third strategy, **over-subscription**, allows more jobs to be admitted than strictly fit, relying on statistical multiplexing to avoid simultaneous peak usage across all resource dimensions. If each job's GPU utilization averages 70 percent (alternating between compute-intensive and data-loading phases), the cluster can support approximately 1.4× as many jobs as the strict GPU count would allow. Over-subscription works well when resource utilization is genuinely bursty and jobs' peak usage periods are uncorrelated, but can cause severe performance degradation (thrashing, memory pressure, network congestion) when multiple jobs peak simultaneously. The key to safe over-subscription is monitoring actual utilization in real time and throttling admission when contention is detected. Figure 8.1 shows the combined effect: although 19 GPUs (roughly 30 percent of the cluster) sit idle, no single node has a contiguous block of eight, so a full-node job cannot be placed.

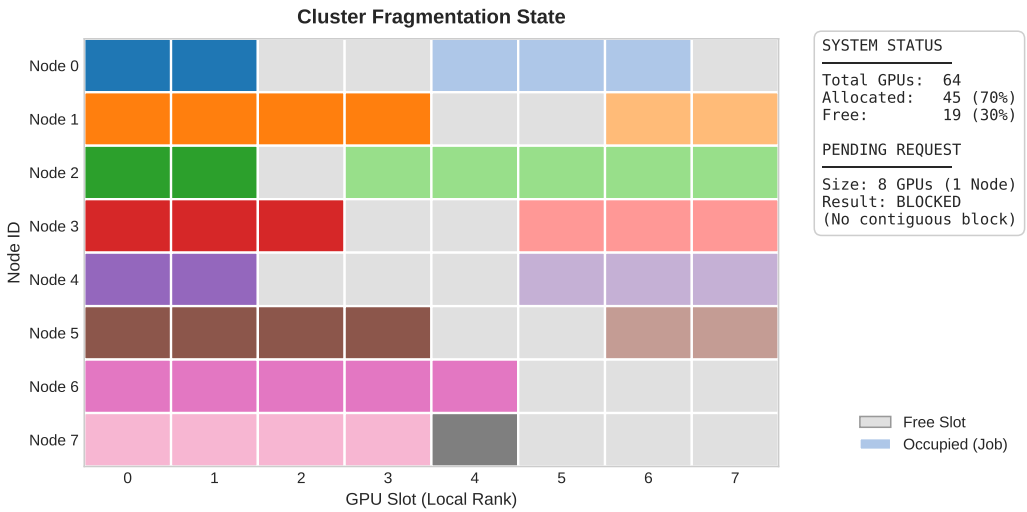


Figure 8.1: The Fragmentation Problem: The heatmap shows a snapshot of an 8-node cluster (rows) with 8 GPUs each (columns). Different colors represent distinct active jobs, while gray slots represent free GPUs. Although 19 GPUs (~30 percent of capacity) are available globally, they are fragmented across nodes. A new job requiring 8 contiguous GPUs (a full node) cannot be scheduled because no single row is completely empty, illustrating how bin-packing inefficiencies reduce effective cluster capacity.

8.1.5 Gang scheduling

Partial allocation of multi-GPU jobs creates a failure mode unique to ML clusters: two jobs can each hold half the resources they need, blocking each other indefinitely while every allocated GPU sits idle. Figure 8.2 contrasts this deadlock scenario with the atomic allocation approach that eliminates it.

Definition 8.1: Gang scheduling

Gang Scheduling is the ML cluster scheduling policy that allocates all of a job's accelerators atomically (all-or-nothing) because synchronous distributed training makes the marginal value of a partial allocation zero: every worker must reach every collective, so a job cannot trade fewer GPUs for proportionally slower progress.

1. **Significance**: A 1,024-GPU synchronous job holding 512 GPUs completes zero training steps, not half as many: each AllReduce stalls until every rank arrives, so the held GPUs burn power and capital at 0 percent useful output. Without atomic allocation, two such jobs can each acquire half the cluster and deadlock, both at zero progress while consuming the full cluster's operating cost.

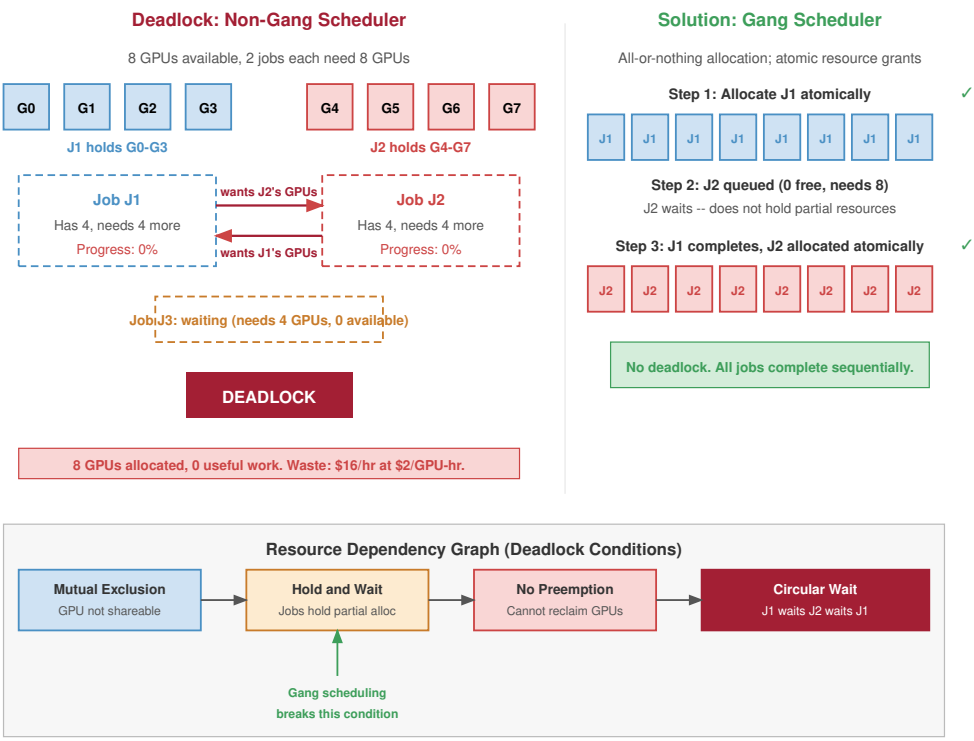


Figure 8.2: The Distributed Deadlock (Hold and Wait): In a non-gang scheduler (left), job J1 acquires half the cluster and waits for the rest, while job J2 acquires the other half and waits for J1 to release its GPUs. Neither can proceed, and 100 percent of the cluster sits idle. Gang scheduling (right) solves this by mandating atomic allocation: a job is either granted its entire N-GPU requirement simultaneously or it waits in the queue without holding any resources.

2. **Distinction**: Unlike cloud service placement (where a service granted half its requested replicas serves roughly half its traffic) and unlike flexible HPC workloads (which degrade gracefully with fewer ranks), synchronous training has step-function utility: full allocation or nothing, which forces the scheduler to reason about atomic multi-resource allocations rather than incremental bin packing.
3. **Common pitfall**: A frequent misconception is that priorities or quotas alone solve this. Atomicity must extend to every co-scheduled resource the job needs to make progress (accelerators, high-bandwidth interconnect domains, network capacity for collectives); enforcing it at GPU granularity alone recreates the deadlock one resource layer down.

Example 8.1: The physics of deadlock

Problem: A 1024-GPU cluster receives 2 large training jobs from separate teams, each requiring the full cluster. A naive non-gang scheduler allocates 512 GPUs to Team A and 512 GPUs to Team B, then waits for more GPUs to become available. What is the steady-state outcome?

Analysis:

1. **Team A status**: Holds 512 GPUs, Needs 1024 GPUs. Progress = 0 percent.

2. **Team B status**: Holds 512 GPUs, Needs 1024 GPUs. Progress = 0 percent.
3. **Cluster status**: 1024 GPUs allocated, 0 samples processed.
4. **Waste**: $2048 per hour in idle electricity and capital.

Systems insight: This is a circular dependency: Team A is waiting for Team B's GPUs, and Team B is waiting for Team A's. Without gang scheduling (all-or-nothing allocation), cluster efficiency drops to zero. In a large fleet, even a 5-minute deadlock can cost thousands of dollars. Batch schedulers and Kubernetes batch extensions address this by enforcing a job-level admission boundary: if the full gang cannot start, the job should remain queued rather than hold a partial allocation.

The partial-allocation pattern in figure 8.2 is a **hold-and-wait deadlock**[5] in which two jobs each hold half the cluster and neither can proceed. The implementation guarantee is straightforward: for a job J requesting N GPUs, the scheduler must guarantee that either all N GPUs are allocated atomically, or the job remains in the queue without holding any resources. This binary outcome eliminates deadlock by construction, since a job that holds no resources cannot block other jobs. Implementing this guarantee efficiently, however, is the challenge that defines much of cluster scheduling algorithm design.

Naive gang scheduling is wasteful in predictable ways. If the cluster has 900 free GPUs and a job requests 1,024, the 900 GPUs sit idle until 124 more become available, potentially wasting thousands of GPU-hours. **Backfill scheduling** addresses this by identifying jobs in the queue that can fit within the 900 available GPUs without delaying the large job's expected start time, an idea popularized in the ANL/IBM SP scheduling system (Lifka 1995). A 128-GPU job with estimated runtime of 2 hours can safely backfill if the 124 additional GPUs needed by the large job will become available within 2 hours anyway (from other completing jobs). The backfilled job uses resources that would otherwise be idle, improving utilization without violating the large job's scheduling guarantee.

The accuracy of runtime estimates determines backfill effectiveness, and this creates a game-theoretic challenge. If users consistently overestimate runtimes, backfill slots are artificially narrow and fewer jobs fit, reducing utilization. If users underestimate, backfilled jobs may still be running when the large job's resources become available, forcing preemption that wastes the backfilled job's progress. In practice, users have strong incentives to overestimate (to avoid being killed before completion) and weak incentives to estimate accurately, leading to systematic inflation that degrades backfill performance. Research schedulers like Tiresias (Gu et al. 2019) address this fundamental problem by eliminating the requirement for runtime estimates entirely, instead using observed resource consumption to dynamically adjust priority. This approach, discussed in detail in section 8.6.1, turns the runtime estimation problem from a user-facing burden into a system-level observation.

A quick cost estimate shows why even modest improvements in this balance justify engineering effort.

Systems Perspective 8.1: The economics of idle GPUs

The cluster economics introduced in section 8.1 set the stake; the gang-scheduling decision is where a scheduler first earns or burns it. Taking the same 10,000-GPU cluster:

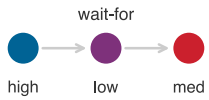
- **Operating cost**: $480,000/day ($175.2M/year)
- **Lower utilization** (60 percent): 6,000 productive, 4,000 idle = $192,000/day wasted
- **Higher utilization** (80 percent): 8,000 productive, 2,000 idle = $96,000/day wasted
- **Improvement value**: Moving from 60 percent to 80 percent saves $96,000/day, or about $35.0M/year

The increment gang scheduling owns is the deadlock it prevents: a single hold-and-wait deadlock can strand the whole cluster, erasing more than a full day of this utilization gain in one event.

[5] **Hold-and-Wait Deadlock**: One of the four classical conditions jointly sufficient for deadlock in resource-allocation systems. In GPU clusters, this condition is uniquely expensive: two jobs each holding 500 GPUs while waiting for 200 more strand 1,000 GPUs indefinitely, burning approximately $2,000/hour. Gang scheduling eliminates hold-and-wait by construction, requiring atomic all-or-nothing allocation at the cost of reduced packing flexibility.

Gang scheduling removes the hold-and-wait deadlock from the utilization example, but another wait-for pattern remains: priority inversion can still strand reserved GPUs behind jobs that cannot finish their own exit path.

8.1.6 Deadlock prevention and detection



Priority inversion is a wait-for chain, not a root-failure tree.

Gang scheduling eliminates the hold-and-wait condition, one of the four formal Coffman conditions for deadlock, but it does not immunize the cluster against all forms of resource contention. Deadlocks can still emerge from the interaction of priority rules, preemption policies, and auxiliary resource dependencies. In a cluster with thousands of GPUs and petabytes of state, these edge cases transition from theoretical curiosities to daily operational incidents.

The most pernicious of these is priority inversion, a scenario borrowed from real-time systems where a high-priority job is indefinitely blocked by a low-priority job. Consider a 175B parameter training run (high priority) requesting a gang of 64 GPUs. The scheduler has reserved 60 available GPUs, but the remaining 4 are held by a low-priority data processing job. Normally, the scheduler would preempt the low-priority job to satisfy the high-priority request. However, if a stream of medium-priority development jobs saturates the cluster's CPU or network bandwidth, starving the low-priority job of the resources it needs to checkpoint and exit, the low-priority job stalls. It cannot release the GPUs because it cannot complete its exit sequence. The high-priority job waits for the low-priority job, which is effectively blocked by the medium-priority jobs. The result is a 60-GPU idle block that persists until an operator manually intervenes.

Definition 8.2: Priority inversion

Priority Inversion is an ML cluster scheduling pathology in which a high-priority task is forced to wait for a lower-priority task to release a shared resource.

1. **Significance**: It reduces the progress rate of the entire fleet to that of the lowest-priority job. In ML clusters, this typically occurs when a low-priority job holding GPUs is starved of auxiliary resources (for example, BW for checkpointing), preventing it from finishing and releasing the accelerators needed by high-priority workloads.
2. **Distinction**: Unlike Standard Queuing (where tasks wait their turn), Priority Inversion involves an Active Blockage: the high-priority task is ready to run but is transitively dependent on a task that the scheduler does not prioritize.
3. **Common pitfall**: A frequent misconception is that strict priority levels solve this. In reality, without Holistic Preemption (reserving all resources needed for a task to exit), increasing priority levels can actually increase the likelihood of inversion by creating more complex dependency chains.

Solving this requires a choice between prevention and detection. Prevention mechanisms, like strict gang scheduling and aggressive timeouts, eliminate deadlock states by construction but often sacrifice utilization. Detection mechanisms allow the system to enter potentially unsafe states to maximize throughput, relying on monitoring to identify and resolve deadlocks when they occur.

For large-scale fleets, exact deadlock detection is computationally nontrivial. A cluster with 10,000 GPUs and 500 pending jobs has a state space exceeding 10^{15} possible allocations, so constructing and traversing a global wait-for graph in real time is often intractable. Large-scale schedulers therefore approximate liveness rather than prove it. A **lease-based resource** gives every allocation a time-to-live (TTL), allowing the scheduler to reclaim GPUs when a job stops renewing its claim. **Progress monitoring** then checks whether the job is actually advancing, using signals such as GPU utilization, step counters, or heartbeat timestamps to identify zombie allocations. When the scheduler must preempt a lower-priority job, **holistic preemption** reserves the CPU, network, and storage bandwidth needed for that job to checkpoint and exit cleanly; otherwise, the victim can be too starved to release the accelerators. These strategies accept a nonzero rate of false positives to preserve cluster liveness.

8.1.7 Heterogeneous gang scheduling

The traditional gang scheduling model assumes a homogeneous Bulk Synchronous Parallel (BSP) workload: N identical workers executing the exact same compute graph. Some training workflows break this assumption because one logical job contains several roles with different resource profiles. In preference-training pipelines such as reinforcement learning from human feedback (RLHF), for example, one component may update the policy while another scores outputs or supplies reference behavior. The scheduler still sees one job, but the gang contains trainers, evaluators, and inference-style workers that need different GPU counts, memory footprints, and lifetimes.

> Systems Perspective 8.2: Routing over synchrony
>
> Heterogeneous gang scheduling reframes this allocation. The scheduler must atomically allocate a diverse constellation of resources: high-HBM nodes optimized for memory-bound inference generation, alongside high-compute nodes for compute-bound backpropagation. The orchestration challenge shifts from static topological bin-packing to dynamic data routing. Activations and gradients are no longer just reduced within a homogeneous ring; instead, data must stream continuously between inference nodes generating rollouts and training nodes computing policy updates. If the heterogeneous gang allocation is not perfectly balanced across these distinct workload profiles, the entire cluster falls out of sync, leaving expensive accelerators starved for data.

Heterogeneous RLHF gangs are the exception; for common homogeneous training jobs, the recurring tension is gang scheduling's safety against backfill's utilization. Gang scheduling prevents deadlock but wastes resources; backfill improves utilization but requires runtime estimates that users cannot accurately provide. Two common orchestration paradigms resolve this tension through fundamentally different architectural philosophies.

8.2 Orchestration Paradigms

The gang-scheduling/backfill tension does not have a tool-neutral answer: the orchestration paradigm determines which guarantee the platform protects first. Slurm, shaped by HPC batch systems, favors explicit resource requests and predictable allocation. Kubernetes, shaped by cloud-native service management, favors declared desired state and continuous reconciliation. The choice is therefore not Slurm vs. Kubernetes as products; it is which control model best matches the workload mix, failure behavior, and operational culture of the fleet.

The fundamental distinction is between *imperative* and *declarative* resource management. In an imperative system, users specify exactly what resources they need and the system allocates them directly. In a declarative system, users specify what they want running and the system converges toward that state. This distinction, familiar from programming language design, determines how ML workloads are scheduled, monitored, and recovered from failure.

8.2.1 Slurm: The HPC heritage

Slurm Workload Manager[6] originated at Lawrence Livermore National Laboratory as a scalable, fault-tolerant resource manager for Linux clusters (Yoo et al. 2003). Its batch-oriented design fits the distinctive requirements of scientific computing: long-running jobs, expensive shared hardware, and users who can specify resource needs in advance. In ML infrastructure, the same model maps to GPU training through partitions for heterogeneous accelerator pools and predictable allocations for long-running distributed jobs.

Slurm uses an **imperative scheduling model**: users submit job scripts specifying exact resource requirements, and the scheduler places jobs into partitions based on priority, fairness, and resource availability. This directness makes reasoning about allocation guarantees straightforward. When a user submits `sbatch --gres=gpu:8 --nodes=4`, Slurm guarantees that if the job starts, it will have exactly 32 GPUs across 4 nodes. The user knows precisely what they are getting, and the scheduler knows precisely what it must provide. The cost of this directness is that users must understand their resource needs precisely; requesting more than needed wastes resources, while requesting less causes out-of-memory errors or reduced performance.

[6] **Slurm Workload Manager**: Originally "SLURM," an acronym for Simple Linux Utility for Resource Management, the project dropped the acronym capitalization in 2012. Developed at Lawrence Livermore National Laboratory beginning in 2002, Slurm's centralized controller (`slurmctld`) maintains a single authoritative view of resource allocations. At very large scale, operators may partition fleets or federate schedulers to keep scheduling latency, policy isolation, and failure domains manageable.

Slurm's architecture consists of a central controller daemon (`slurmctld`) that maintains the global view of cluster state and makes scheduling decisions, and per-node daemons (`slurmd`) that manage local resources and execute jobs. This centralized architecture provides the authoritative allocation model discussed in section 8.1.1: the controller is the source of truth for resource assignments, reducing the risk of conflicting allocations. The cost is a potential single point of failure (addressed through active-passive failover) and a scheduling-throughput limit. As fleets grow, organizations may partition the cluster or federate schedulers to bound scheduling latency and failure impact.

A typical ML cluster configuration defines Slurm partition configuration by accelerator type and interconnect, as table 8.1 shows:

Table 8.1: Slurm Partition Configuration: Partitions organize heterogeneous accelerators into logical pools matched to workload characteristics. NVLink-connected partitions support tensor parallelism, while PCIe partitions serve workloads that rely primarily on data parallelism. Separating inference and debug partitions prevents experimental workloads from impacting production serving.

Partition	GPUs/Node	Interconnect	Typical Use
dgx-a100	8× A100	NVLink + IB NDR	Large LLM training
a100-pcie	4× A100	PCIe + IB HDR	Medium training
inference	2× A10G	Ethernet	Model serving
debug	1× V100	Ethernet	Development

Those partitions make GPU allocation policy concrete, and Slurm provides several mechanisms for controlling placement. The `--gres=gpu:N` flag requests `N` GPUs per node, while `--gpus=N` requests a total GPU count for the job. Naive allocation can fragment nodes: if jobs request 6 GPUs on 8-GPU nodes, each job wastes 2 GPUs per node, reducing effective capacity to 75 percent. Slurm's `select/cons_tres` plugin enables GPU-aware consumable-resource scheduling, tracking individual GPUs rather than treating nodes as indivisible units. The `--gpus-per-node` flag requests a fixed GPU count on each allocated node when NVLink communication patterns make partial-node allocation counterproductive, while `--gpus-per-task` distributes GPUs evenly across tasks for data-parallel workloads.

The interaction between GPU allocation and node selection creates subtleties that affect both performance and utilization. A job requesting `--gpus=16 --gpus-per-node=8` will always receive exactly 2 complete nodes, ensuring all 8 GPUs within each node communicate via NVLink. The same job requesting `--gpus=16` without the per-node constraint might receive GPUs spread across 3 or 4 partially occupied nodes, degrading intra-node communication performance. For training jobs using tensor parallelism, the per-node constraint is essential; for data-parallel jobs that communicate only through AllReduce, the flexibility of unconstrained placement improves the scheduler's ability to find valid allocations and reduces fragmentation.

The **fair-share scheduling** mechanism prevents any single user or project from monopolizing resources over time. The core insight is that past usage should affect future priority: heavy users should yield to lighter users when the cluster is contended. The effective-priority heuristic combines base priority, the user's target fair share (F_{target}), the user's recent actual consumption (F_{actual}), and a small stabilizing constant (ϵ):

$$\text{Priority}_{\text{effective}} = \text{Priority}_{\text{base}} \times \frac{F_{\text{target}}}{F_{\text{actual}} + \epsilon} \tag{8.1}$$

The fair-share formula in equation 8.1 naturally deprioritizes heavy users while allowing burst access when resources are idle. A researcher who has consumed twice their fair share sees their priority halved, pushing new submissions behind colleagues who have used less. Conversely, a researcher who has used no resources recently receives maximum priority, allowing rapid access when they return to active work.

The time decay of usage history determines how quickly the system "forgives" past heavy usage. A half-life of one week means that a researcher's heavy usage from two weeks ago contributes only 25 percent to their current usage calculation. Short half-lives create rapid rebalancing but can lead to oscillatory behavior where users alternate between starving and gorging on resources. Long

half-lives create stable priority ordering but may penalize researchers who completed a legitimate large project and now want resources for a smaller one. Most production clusters configure half-lives between 3 and 14 days, balancing responsiveness against stability.

Preemption policies enable high-priority jobs to reclaim resources from running workloads, and for ML training, this requires careful coordination with the checkpoint infrastructure. Slurm can notify selected jobs with `SIGTERM` and use `GraceTime` to delay final termination, typically giving 60 to 300 seconds for a checkpoint before the eventual `SIGKILL`. `PreemptMode=REQUEUE` requeues eligible batch jobs (for example, jobs submitted with `--requeue` or clusters with `JobRequeue=1`), and the job's startup path must resume from the latest checkpoint. The checkpoint and recovery infrastructure developed in Chapter 7 makes this preemption practical; without reliable checkpointing, preemption would lose all progress since the last manually triggered save, making preemption economically ruinous for long-running training jobs.

Preemption introduces a scheduling paradox: it improves the scheduler's ability to serve high-priority work but can degrade overall cluster throughput. Every preemption wastes the compute between the last checkpoint and the preemption event, plus the time to restart and reload from checkpoint. If checkpoint intervals are long (for example, hourly) and preemptions are frequent, the wasted compute can be substantial. Production systems balance preemption frequency against checkpoint overhead, often configuring preemption cooldown periods that prevent the same job from being preempted more than once within a configurable interval.

Slurm's strengths for ML training are clear: predictable allocation guarantees, mature fair-share policies, straightforward integration with MPI-based distributed training frameworks, and decades of operational experience in managing large-scale scientific computing. Its weaknesses emerge for inference workloads and mixed training-serving clusters, where the batch-oriented model struggles with the dynamic, latency-sensitive nature of serving traffic. Slurm has no native concept of "a service that should always be running," making it awkward for inference endpoints that must respond to requests continuously. Adding or removing inference replicas requires submitting or canceling Slurm jobs, a much heavier operation than scaling a Kubernetes deployment.

8.2.2 Advanced Slurm configuration for ML

Once Slurm owns batch training, the remaining question is how much ML semantics the batch system can see. Standard partitions and fair-share policies schedule ordinary jobs; large-scale deep learning needs configuration that exposes sweep structure, asymmetric pipeline resources, hardware health at job boundaries, and the economic weight of different accelerator types.

Job arrays transform the chaotic submission of thousands of experimental hyperparameter sweep trials into a coherent, schedulable unit. Instead of flooding the controller with individual `sbatch` requests, a user submits a single array job: `#SBATCH --array=0-99`. Slurm treats this as a single object for parsing and queuing overhead but schedules each element as an independent task, allowing the scheduler to backfill small holes in the cluster with individual trials. Each task receives a unique `SLURM_ARRAY_TASK_ID` environment variable, which the training script uses to index into a hyperparameter configuration file (for example, selecting learning rate $\eta = 10^{-4}$ for task 0 and $\eta = 3 \times 10^{-4}$ for task 1). This mechanism is common for large-scale ablation studies, enabling researchers to launch 1,000 experiments with a single command while preserving scheduler throughput.

Heterogeneous jobs address the preprocessing bottleneck where expensive GPUs sit idle while CPUs prepare data. Traditional jobs allocate symmetric resources to every node, forcing users to reserve GPU nodes for the entire pipeline. Slurm heterogeneous job support lets a single submission span disparate resource types by separating components with `:` on the command line or `#SBATCH hetjob` inside a script. The 64-GPU fine-tuning run for our 175B model (the scheduling-demo workload, not the full 1,024-GPU pretraining run) uses this pattern: it requests a heterogeneous allocation of 8 GPU nodes (64 A100s) for the model and 2 CPU-only nodes for on-the-fly data tokenization. The `srun --het-group=0` command launches the training process on the GPUs, while `srun --het-group=1` launches the data workers, allowing the expensive accelerators to focus purely on gradient computation while cheaper CPU nodes feed them data.

Prolog and epilog scripts serve as the cluster's immune system, running administrative code before a job starts and after it finishes. In ML clusters, a typical prolog script performs a preflight check on allocated hardware: it validates NVLink topology (using `nvidia-smi topo -m`), checks ECC

error counters, and verifies InfiniBand link width. If a node fails these checks, the prolog returns a nonzero exit code, causing Slurm to drain the node and requeue the job elsewhere, preventing a silent hardware fault from corrupting a week-long training run. The epilog script ensures hygiene by killing orphaned Python processes, clearing shared memory segments (`/dev/shm`), and logging final GPU utilization metrics to the accounting database. For the 64-GPU fine-tuning run, the prolog script specifically validates that all 8 GPUs on each node have full P2P bandwidth access, preventing a single degraded NVLink lane from bottlenecking the entire tensor parallel group.

Finally, **Trackable Resources (TRES)** extend Slurm's accounting beyond simple CPU/memory tracking to assets like GPU-hours, license tokens, or power budget. The capability that matters is economic-weight accounting per accelerator class: each tracked resource carries a billing weight, so fair-share can be driven by actual economic cost rather than raw core counts. This ensures that a team using 100 older GPUs does not deplete its budget as fast as a team using 100 flagship H100s, and it lets organizations implement project billing for expensive resources directly inside the scheduler's fairness calculation.

These extensions make Slurm more ML-aware while preserving its batch-scheduler model. Kubernetes starts from a different contract: users declare the desired state of services and jobs, and controllers continuously reconcile the cluster toward that state.

8.2.3 Kubernetes: Declarative orchestration

Kubernetes is a common platform for ML infrastructure, particularly for organizations requiring unified management of training and serving workloads (Burns et al. 2016; Cloud Native Computing Foundation 2024). Where Slurm's model is imperative ("run this job on these resources"), Kubernetes uses a **declarative model** ("ensure this state exists"), using control loops that continuously reconcile desired state with actual state. This fundamental difference shapes every aspect of ML workload management, from job submission to failure recovery.

The declarative model's power lies in its approach to failure handling. Slurm's recovery is configuration-dependent and batch-oriented: slurmctld can detect node failures through slurmd heartbeats and requeue jobs when `Requeue=1` is set, but the recovery semantics require explicit configuration to behave like a continuously-reconciling control loop. In Kubernetes, the control loop continuously reconciles desired state ("4 replicas of this serving worker should be running" for a Deployment, or the configured retry policy for a Job) against actual state, automatically creating a replacement pod on a healthy node when divergence is detected. For distributed training specifically, replica-replacement semantics typically come from a higher-level operator such as Kubeflow's PyTorchJob or MPIJob rather than core Kubernetes primitives. This self-healing behavior reduces operational burden but introduces complexity: the system's actions are emergent from control loop interactions rather than explicitly commanded, making debugging more challenging when things go wrong.

Kubernetes' control loop architecture is built on the **controller pattern**[7]: a controller watches the cluster's actual state (stored in etcd, Kubernetes' distributed key-value store for API state), compares it to desired state (specified by users through API objects like Deployments, Jobs, and StatefulSets), and takes corrective action to close the gap. This pattern repeats at every level: the Deployment controller ensures the right number of pods exist, the scheduler assigns pods to nodes, the kubelet on each node ensures containers are running, and the node controller detects node failures. Each controller operates independently, communicating only through shared state in etcd, which makes the system resilient to individual controller failures but can create emergent behaviors when multiple controllers interact.

Native Kubernetes lacks ML-aware scheduling, but extensions address this gap. GPU scheduling relies on **device plugins** that expose accelerators as schedulable extended resources. The NVIDIA device plugin registers GPUs with the kubelet (the node-level agent), enabling pod specifications that request GPU resources declaratively through the standard Kubernetes resource model. Listing 8.1 shows the resulting pod fragment.

[7] **Controller Pattern (Reconciliation Loop)**: A "level-triggered" design where the controller continuously compares desired state to actual state, as opposed to "edge-triggered" systems that react to individual events. The distinction matters for ML clusters: if a controller crashes and restarts, it re-reads current state and self-heals without needing an event log. The trade-off is that emergent interactions between multiple independent controllers can create unexpected scheduling behaviors that are difficult to debug sequentially.

Listing 8.1: Kubernetes GPU Allocation: Pod resource specification requesting GPUs through the NVIDIA device plugin. The `nvidia.com/gpu` resource name follows Kubernetes extended resource conventions, where the domain prefix identifies the device plugin vendor. This declarative syntax enables portable GPU workload definitions across any Kubernetes cluster with the NVIDIA device plugin installed.

```
# Kubernetes pod resource specification for GPU allocation
resources:
  limits:
    nvidia.com/gpu: 4  # Request exactly 4 GPUs for this pod
  requests:
    cpu: "32"
    memory: "256Gi"
```

This binary allocation model creates a significant inefficiency for inference workloads. A small model serving occasional requests might need only a fraction of a GPU's compute and memory capacity, yet Kubernetes allocates an entire GPU, wasting the remaining capacity. For training workloads that saturate GPU compute, full-device allocation is appropriate. For inference workloads with variable and often modest resource needs, it creates the same kind of fragmentation that partial-node allocation creates in Slurm.

Multi-Instance GPU (MIG)[8] technology addresses this inefficiency by partitioning A100 (80 GB) and H100 (80 GB) GPUs into hardware-isolated instances. Unlike software-based GPU sharing, MIG dedicates memory, cache, and compute resources to each instance, preventing one workload from interfering with another and eliminating the noisy neighbor[9] effect. The A100 MIG profiles in table 8.2 show the scheduling consequence: a single physical A100 becomes several fixed-size resources that the device plugin can expose as independently schedulable accelerators.

Table 8.2: A100 MIG Profiles: Multi-Instance GPU configurations trade GPU memory and compute resources for increased sharing density. Each profile provides hardware-isolated resources, making MIG suitable for multi-tenant inference clusters where workloads from different teams or customers must not interfere. The SM (Streaming Multiprocessor) count determines compute throughput within each instance.

MIG Profile	GPU Memory	SM Count	Typical Workload
1g.10gb	10 GB	14 SMs	Small inference
2g.20gb	20 GB	28 SMs	Medium inference
3g.40gb	40 GB	42 SMs	Large inference
7g.80gb	80 GB	98 SMs	Training

As section 8.1.5 explains, default Kubernetes pod scheduling is independent, so ML platforms add admission or gang semantics when all-or-nothing startup matters. The default scheduler evaluates pods one at a time, placing each on the best available node. For a distributed training job with 64 pods, this means 64 independent scheduling decisions, with no guarantee that all 64 will succeed. If the cluster has resources for 60 pods but not 64, the default scheduler may place 60 pods and leave the remaining 4 pending, creating exactly the partial-allocation waste that gang scheduling prevents. PodGroup-style abstractions, whether implemented by scheduler plugins or higher-level batch controllers, give the platform a unit of admission that matches the training job rather than the individual pod.

The **Volcano**[10] batch scheduler and **Coscheduling** scheduler plugin address this gap by implementing gang semantics through **PodGroup** abstractions. A PodGroup declares that a set of pods must be scheduled together, specifying a minimum member count that must be satisfiable before any pod in the group starts. The scheduler evaluates the entire group atomically: either all minimum members can be placed, and they are placed simultaneously, or none are placed and the group waits in the queue. This transforms Kubernetes' pod-level scheduling into job-level scheduling that respects the all-or-nothing requirements of distributed training.

Priority classes control preemption behavior in Kubernetes, establishing a hierarchy that determines which workloads yield resources to which others. A typical production configuration assigns

8 **Multi-Instance GPU (MIG)**: Introduced with the A100 (2020), MIG partitions a single GPU into up to 7 hardware-isolated instances with dedicated memory controllers, L2 cache slices, and compute units (NVIDIA Corporation 2020). The isolation is enforced at the hardware level, not by time-slicing, reducing noisy-neighbor interference relative to software-only sharing. The trade-off is rigidity: partition profiles cannot change without draining all workloads, making MIG poorly suited for clusters with rapidly shifting workload mixes.

9 **Noisy Neighbor**: A metaphor from multi-tenant housing where one resident's activity (loud music) disturbs another's peace. In GPU clusters, this occurs when one job's bursty network traffic or memory bus utilization slows down a co-located job on the same node. For distributed training, noisy neighbors are fatal to efficiency because the slowest flow dictates the global AllReduce time.

10 **Volcano**: Open-sourced by Huawei and now a CNCF incubating project, Volcano replaces the Kubernetes default scheduler entirely to add gang scheduling via its PodGroup CRD. The replacement approach provides strong atomicity guarantees for multi-pod ML training jobs but carries operational risk: a bug in Volcano affects all scheduling decisions on the cluster, not just batch workloads.

inference workloads high priority (ensuring serving SLOs are met), training workloads medium priority, and development jobs low priority. The default PriorityClass behavior, `preemptionPolicy: PreemptLowerPriority`, allows a higher-priority pending pod to evict lower-priority pods when that would make room for it. A PriorityClass with `preemptionPolicy: Never` is non-preempting: its pods can sit ahead of lower-priority pods in the scheduling queue, but they do not evict running pods and can still be preempted by still-higher-priority pods. Organizations must carefully balance the priority hierarchy: overly aggressive inference preemption can thrash training jobs (repeatedly preempting and restarting them), while overly permissive training priorities can starve inference during demand spikes. The Kubernetes priority and preemption system integrates with checkpoint-aware preemption patterns developed later in Chapter 7, where preempted training jobs save state before yielding resources, minimizing wasted computation.

Kueue[11], a newer Kubernetes-native job queueing system, represents an emerging approach that separates *admission control* from *scheduling*. Rather than replacing the Kubernetes scheduler entirely (as Volcano does), Kueue manages when jobs enter the cluster, while the default scheduler (or any compatible scheduler) handles where pods are placed. This separation of concerns has practical advantages: Kueue can be deployed alongside existing Kubernetes infrastructure without disrupting running workloads, and it integrates naturally with the ecosystem of scheduler plugins for topology-aware placement and other features.

Kueue provides fair-share scheduling through ClusterQueues that represent organizational teams or projects. Each queue has resource quotas, borrowing policies that allow queues to use idle capacity from other queues, and preemption policies that reclaim borrowed resources when the owning queue needs them. A research team queue might borrow idle capacity from a production team queue during off-peak hours, automatically returning resources through preemption when the production team submits urgent work. This borrowing mechanism mirrors the hierarchical fair-share approaches in Slurm but expressed through Kubernetes' declarative model rather than Slurm's imperative configuration.

11 **Kueue**: Developed by Google, Kueue separates *admission control* (when jobs enter the cluster) from *scheduling* (where pods are placed), unlike Volcano which replaces the scheduler entirely. This less-invasive design can be deployed alongside existing Kubernetes infrastructure without disrupting running workloads, but it lacks Volcano's strong gang scheduling guarantees, forcing teams to choose between deployment safety and scheduling atomicity.

8.2.4 Kubernetes ecosystem for ML training

Kubernetes provides the orchestration primitives, but a specialized ecosystem of operators and plugins is required to bridge the gap between microservice orchestration and high-performance training. Each extension makes one hidden operational constraint visible to the platform: job atomicity, network bypass, checkpoint storage, or telemetry. This ecosystem augments Kubernetes with the batch processing, hardware acceleration, and observability capabilities found in HPC environments.

The **Training Operator ecosystem** simplifies distributed training by extending Kubernetes with job-specific Custom Resource Definitions (CRDs). The Kubeflow Training Operator provides controllers for `PyTorchJob`, `TFJob`, and `MPIJob` resources, automating the complex lifecycle of distributed workloads. A `PyTorchJob` specification defines the number of workers, the container image, and resource requirements; the operator then creates the necessary pods and configures distributed environment variables such as `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, and `RANK`. If a worker pod fails, the operator can detect the deviation and create a replacement pod to restore the desired replica count, but training-state recovery still depends on the framework's checkpoint and rendezvous logic.

Achieving high network performance in Kubernetes often requires bypassing networking layers that were designed for ordinary microservices rather than tightly synchronized collectives. CNI configuration, overlay networking, and CPU switching can add latency or reduce effective bandwidth when they sit on the critical gradient-synchronization path. For bandwidth-sensitive training, **host networking** (`hostNetwork: true`) allows pods to bypass the CNI entirely, granting direct access to the host's network namespace and RDMA devices. The NVIDIA Network Operator automates the low-level plumbing required for this performance, managing RDMA device plugins, SR-IOV virtual function assignment, and GPUDirect RDMA configuration. This bypass capability allows containerized workloads to approach bare-metal interconnect performance when the remaining stack is configured correctly.

Storage integration addresses the "checkpoint storm" problem where thousands of GPUs simultaneously write terabytes of state. Kubernetes PersistentVolumeClaims (PVCs) abstract the underlying storage fabric, which may be a parallel file system like Lustre or GPFS, or high-performance object

storage via CSI drivers. The critical architectural challenge is preventing these synchronized writes from saturating the storage backend. Storage classes with Quality of Service (QoS) policies and client-side throttling are essential to ensure that a 1,024-GPU training job writing a 2 TB checkpoint does not starve other workloads or crash the storage metadata server.

The **observability stack** for training clusters combines general-purpose cluster monitoring with specialized hardware telemetry. Prometheus and Grafana provide the collection and visualization layer, while the DCGM (Data Center GPU Manager) exporter exposes granular GPU metrics: utilization, memory bandwidth, temperature, and ECC errors. This stack enables the utilization dashboards essential for multi-tenant efficiency, allowing operators to distinguish between true compute saturation and I/O-bound idleness.

The 1,024-GPU pretraining run for our 175B model uses this full Kubernetes stack. The workload is defined as a `PyTorchJob` with 128 worker pods, each requesting 8 GPUs. Host networking is enabled to allow direct InfiniBand access for the NVLink-connected nodes. A PVC backed by a high-throughput Lustre file system provides access to the 100 TB training dataset and absorbs periodic checkpoints. DCGM metrics stream to Prometheus, triggering alerts if any GPU reports uncorrectable ECC errors or if NVLink bandwidth drops below the expected baseline.

8.2.5 Choosing between paradigms

The choice between Slurm and Kubernetes is rarely binary; it depends on workload composition, organizational context, and the relative importance of different system qualities. Understanding where each paradigm excels guides the architectural decision.

Slurm excels for pure training clusters where predictability and bare-metal performance matter most. Its direct resource management avoids the container overhead that Kubernetes introduces (container networking adds microseconds of latency and reduces effective network bandwidth by 1 to 5 percent), and its batch scheduling model aligns naturally with long-running training jobs that require guaranteed, uninterrupted access to specific hardware. Slurm's centralized scheduler has global visibility into cluster state, enabling sophisticated multi-factor priority decisions that account for fair-share, job size, queue depth, and resource availability simultaneously. National laboratories, university research clusters, and dedicated training infrastructure typically favor Slurm because these environments prioritize maximum hardware utilization and minimal overhead for a relatively homogeneous set of batch workloads.

Kubernetes excels for mixed training-serving clusters where unified management, rolling updates, and service mesh integration reduce operational complexity. Organizations running both training pipelines and production inference endpoints benefit from a single control plane that manages both workload types through a consistent API. Kubernetes' declarative model simplifies operational workflows: deploying a new model version means updating a Deployment specification, and Kubernetes handles rolling updates, health checks, and rollback automatically. Cloud-native organizations, teams deploying ML as microservices, and organizations with existing Kubernetes expertise typically favor Kubernetes because the marginal cost of adding ML workloads to an existing platform is lower than operating a separate scheduling system.

Many production environments recognize that neither paradigm alone satisfies all requirements, and they deploy both systems in complementary roles. A common architecture uses Slurm for dedicated training clusters where raw performance and scheduling sophistication matter, and Kubernetes for inference serving, pipeline orchestration, and lighter training workloads (fine-tuning, small-scale experimentation). The emerging pattern is to use Kubernetes as the outer orchestration layer, submitting Slurm jobs for large training runs through Kubernetes operators. This hybrid architecture achieves unified management and observability without sacrificing Slurm's batch scheduling capabilities for the workloads that need them most.

The Slurm vs. Kubernetes trade-off organizes around the dimensions most relevant to ML workloads in table 8.3:

Table 8.3: Slurm vs. Kubernetes for ML Workloads: The two paradigms reflect different design philosophies optimized for different workload characteristics. Most production ML platforms use elements of both, with Slurm managing dedicated training clusters and Kubernetes managing inference serving and pipeline orchestration.

Dimension	Slurm	Kubernetes
Scheduling model	Imperative: user specifies exact resources and scheduler allocates them	Declarative: user specifies desired state, system reconciles continuously
Failure handling	Configuration-dependent requeue and checkpoint workflows	Self-healing through control loops
Gang/atomic multi-node start	Atomic job allocation plus backfill; Slurm "gang scheduling" specifically refers to time-slicing/preemption	Production clusters commonly use PodGroup-based support such as Volcano, Coscheduling, Kueue, or upstream alpha gang scheduling
Fair-share	Mature multi-factor priority with configurable decay	Kueue provides basic fair-share with ClusterQueue borrowing
Container overhead	None from a container runtime when jobs run directly on nodes	Configuration-dependent networking overhead and container startup time
Service management	No native support for long-running services	Native Deployment, rolling updates, health checks
Ecosystem	MPI, OpenMP, scientific computing libraries	Cloud-native, microservices mesh, observability stack

As table 8.3 illustrates, the choice also reflects organizational trajectory. Teams starting with HPC-focused training infrastructure may begin with Slurm and add Kubernetes for serving. Teams starting from cloud-native application infrastructure may begin with Kubernetes and add HPC-oriented extensions (Volcano, Kueue) for training. Either path can reach a functional hybrid architecture, but the starting point determines which capabilities are strongest and which require the most investment to develop.

8.2.6 Hybrid architectures in practice

While the choice between Slurm and Kubernetes often appears binary, sophisticated engineering organizations increasingly deploy **hybrid architectures** that use the strengths of both systems. These architectures acknowledge a fundamental reality: training workloads behave like HPC applications (batch, synchronous, monolithic), while serving workloads behave like microservices (continuous, asynchronous, elastic). Forcing both into a single paradigm often results in a "lowest common denominator" system that serves neither well.

The **Kubernetes-over-Slurm** pattern wraps the HPC scheduler within a cloud-native control plane. In this architecture, Kubernetes acts as the outer orchestration layer, managing the lifecycle of training jobs through custom operators while delegating the actual pod placement to Slurm. A developer submits a `TrainingJob` Custom Resource Definition (CRD) to Kubernetes, which the operator translates into a Slurm `sbatch` script. The operator then monitors the job status via Slurm's REST API, streaming logs and metrics back to the Kubernetes control plane. This approach provides the best of both worlds: the unified observability, CI/CD integration, and access control of Kubernetes, combined with the gang scheduling sophistication, topology awareness, and bare-metal performance of Slurm.

The **Slurm-alongside-Kubernetes** pattern takes a coarser-grained approach, maintaining separate clusters for training and serving that draw from a shared pool of physical nodes. A meta-scheduler or capacity broker acts as an arbitrator, dynamically reassigning nodes between the Slurm partition and the Kubernetes cluster based on demand signals. During intensive training campaigns, nodes are drained from Kubernetes and joined to Slurm; during product launches or traffic spikes, the flow reverses. The primary challenge is the mode-switching cost: draining a Kubernetes node, re-imaging or reconfiguring it, and joining it to Slurm (or vice versa) typically requires 5 to 15 minutes. This latency limits the granularity of sharing: the system can adapt to diurnal patterns or weekly cycles but cannot use Slurm capacity to absorb second-by-second inference bursts.

Emerging unified platforms like Ray attempt to eliminate the dichotomy entirely by providing a single runtime for both training and serving. Ray implements its own distributed scheduler that handles actor placement, task scheduling, and resource management natively. This eliminates the impedance mismatch between training (stateful actors) and serving (stateless replicas), allowing a

single Python script to define a training loop that seamlessly transitions into a deployment pipeline. The trade-off is maturity: while Ray excels at orchestration flexibility, its scheduling logic lacks the decades of hardening found in Slurm and the massive ecosystem support of Kubernetes, effectively requiring the platform team to assume more responsibility for reliability and isolation.

Threading these patterns together, consider the lifecycle of our 175B parameter model. The training phase runs on Slurm to maximize NVLink topology efficiency and gang scheduling reliability for the multi-month run. Once trained, the model artifacts are handed off to Kubernetes for serving, where rolling updates and autoscaling ensure high availability for end users. The bridge between them is a Kubernetes operator that monitors training progress; when validation loss stabilizes, it automatically triggers a quantization pipeline and deploys the resulting model to the inference cluster.

The operational impact of these choices is quantifiable, but the numbers depend on workload mix and local platform maturity. A pure Slurm-only environment minimizes batch-job overhead and can keep dedicated training clusters highly utilized, but it requires extra engineering to build production serving infrastructure. A Kubernetes-only environment reduces operational fragmentation by using standard cloud-native tools, but distributed training may need batch extensions to avoid scheduling fragmentation. The hybrid approach targets the efficient frontier: preserving strong batch-training placement guarantees while keeping a unified management surface for serving and pipeline workloads, at the cost of increased architectural complexity.



Systems Perspective 8.3: The convergence of HPC and cloud

The sharp distinction between HPC and cloud-native scheduling is rapidly blurring as both communities adopt features from the other. Kubernetes is evolving batch capabilities through the Volcano and Kueue projects, introducing concepts like queues, priorities, and gang scheduling that were previously unique to HPC. Conversely, Slurm is adopting cloud features like REST APIs, container runtime support, and elasticity. The industry is converging toward a unified fleet operating system: a system that offers the declarative API and fault tolerance of the cloud with the strict placement guarantees and performance of the supercomputer.

This convergence turns the paradigm choice into a workload-specific design decision: teams must decide which scheduling guarantees matter before layering on topology constraints.



Checkpoint 8.1: Scheduling paradigm trade-offs

Before the chapter turns to topology-aware scheduling, review the core trade-offs:

- ☐ **Gang scheduling**: Explain why gang scheduling is essential for distributed training but not for inference workloads, including the property of synchronous data parallelism that creates the all-or-nothing requirement.
- ☐ **Slurm vs. Kubernetes**: Identify the CAP theorem trade-off Slurm makes differently from Kubernetes, and describe how this difference appears during a network partition between the scheduler and a subset of nodes.
- ☐ **MIG partitioning**: Determine when MIG partitioning improves cluster utilization through concurrent inference workloads on fewer GPUs and when it reduces utilization for training jobs that need full GPU memory.
- ☐ **Backfill incentives**: Explain why backfill scheduling depends on accurate job runtime estimates and why users have game-theoretic incentives to provide inaccurate estimates.

The orchestration paradigms treat GPUs as interchangeable units within a partition, but physical location matters as much as count: the same 64 GPUs run far slower scattered across racks than packed into one. Topology-aware scheduling exploits that structure.

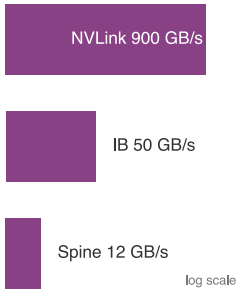
8.3 Topology-Aware Scheduling

Randomly assigning 64 GPUs across a massive data center cripples a synchronous training job, causing it to run 15 to 30 percent slower than if those same 64 GPUs were packed into a single rack. The scheduling algorithms discussed so far treat GPUs as interchangeable units, but topology-aware scheduling recognizes that physical proximity in the network fabric is as critical as raw compute.

8.3.1 The topology hierarchy

Modern GPU clusters exhibit a multi-level communication hierarchy, where bandwidth decreases and latency increases at each level. This hierarchy is not an implementation detail that the scheduler can ignore; it is a physical constraint that directly determines training throughput for communication-intensive workloads. Understanding the hierarchy is essential for making placement decisions that translate allocated resources into useful work.

Within a single node, GPUs communicate via NVLink, providing 600 GB/s per GPU on A100 systems and 900 GB/s on H100 systems. This high-bandwidth, low-latency interconnect enables efficient tensor parallelism, where matrix operations are split across GPUs that must exchange intermediate results (activations and gradients) at every transformer layer. The NVLink bandwidth is sufficient to overlap communication with computation for most model architectures, meaning tensor parallel operations can proceed at near-ideal throughput when confined to a single node.



Bandwidth cliffs from NVLink to spine across the fabric.

Between nodes within the same rack or **leaf switch domain**, communication traverses InfiniBand or RoCE at 400 Gb/s (NDR InfiniBand), roughly 50 GB/s per link. This represents an order of magnitude reduction from NVLink bandwidth. Data parallelism, which requires AllReduce of gradients after each training step, operates efficiently at this level because the gradient synchronization pattern allows overlap between communication and the next iteration's forward pass. The latency penalty is modest (microseconds vs. nanoseconds), and the aggregate bandwidth is sufficient for gradient tensors that are typically smaller than the activation tensors exchanged in tensor parallelism.

Between racks connected through **spine switches**, communication crosses additional switch hops, introducing both higher latency and potentially reduced effective bandwidth due to network congestion. The fat-tree topologies analyzed in section 3.5.2 provide full bisection bandwidth in theory, but in practice, congestion from concurrent flows, routing inefficiencies, and switch buffer limitations reduce effective cross-rack bandwidth by 10 to 30 percent compared to intra-rack communication. Cross-rack communication also adds tail latency variability: while median latency may be only slightly higher, P99 latency can spike during congestion events, creating straggler effects in synchronous training where the slowest communicator determines the pace for all workers.

This hierarchy means that a 64-GPU job allocated entirely within a single rack (8 nodes, all under one leaf switch) will outperform the same job spread across 8 racks (1 node per rack, crossing spine switches for every AllReduce). The performance difference stems directly from the communication analysis in Chapter 6: ring AllReduce latency scales with the slowest link in the ring, and cross-rack hops introduce both higher latency and increased contention risk. For synchronous training, every iteration waits for the slowest worker to complete its AllReduce, so even occasional cross-rack congestion events create persistent throughput degradation.

The scheduler must therefore model the cluster not as a flat list of compute slots, but as a hierarchical graph with distinct bandwidth cliffs at each level (GPU, Node, Rack, and Pod) corresponding to NVLink, InfiniBand, and spine-switch boundaries. Each boundary crossing introduces an order-of-magnitude bandwidth reduction, and at the rack and pod levels, network oversubscription (often 2:1 or 4:1 at the spine switches) further restricts bisection bandwidth beyond what the raw link speed suggests. A topology-aware scheduler explicitly optimizes for these constraints, treating "locality" as a first-class resource. For a distributed training job requiring 1,024 GPUs, a fragmented placement that scatters workers across random racks can degrade end-to-end training throughput by 15 to 30 percent compared to a compact placement within a single pod.

Figure 8.3 contrasts optimal and scattered placement for a 4-GPU tensor parallel group, showing how the communication path determines whether GPUs use a single leaf switch or must traverse slower leaf-spine-leaf hops.

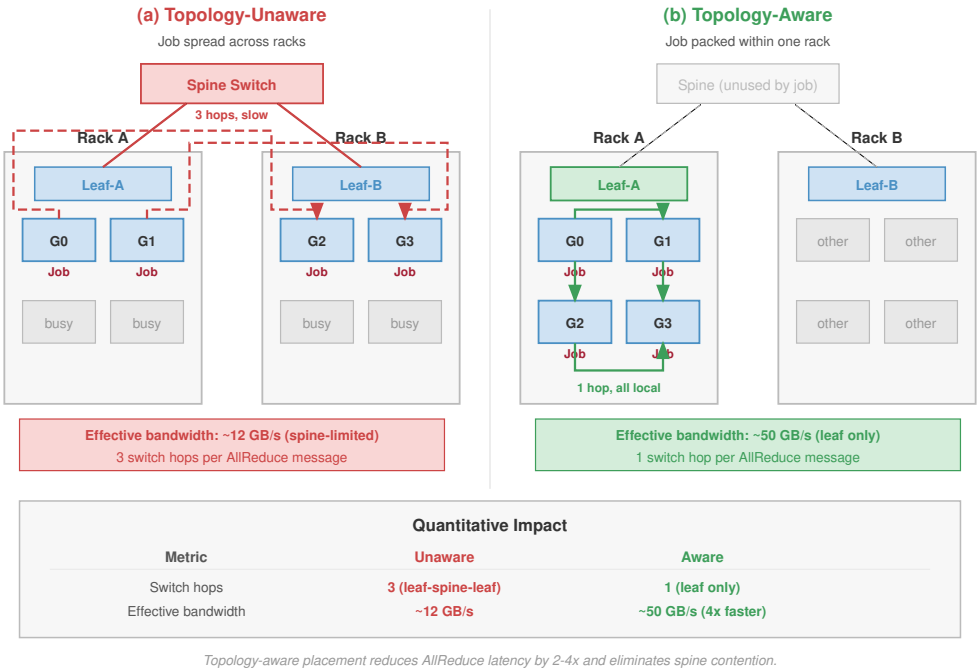


Figure 8.3: Topology-Unaware vs. Topology-Aware Placement: A 4-GPU job placed across two racks (left, topology-unaware) must traverse leaf-spine-leaf hops: 3 switch hops per AllReduce message at about 12 GB/s effective bandwidth. The same 4-GPU job packed within one rack (right, topology-aware) uses a single leaf switch: 1 hop at about 50 GB/s, roughly 4× faster. A quantitative summary beneath the figure reports switch-hop counts and effective bandwidth.

8.3.2 Placement algorithms

The bandwidth gap shown in figure 8.3 is the key constraint: the roughly 4× reduction from intra-rack leaf paths to cross-rack leaf-spine-leaf paths means that scattered placement forces tensor parallel groups to spend more time communicating than computing, a penalty that compounds at every transformer layer in the model. Topology-aware placement algorithms assign jobs to nodes that minimize communication cost, transforming the abstract bin packing problem into a graph optimization problem on the cluster's physical topology. The simplest approach uses a **locality cost**, defined in equation 8.2, that penalizes allocations spanning multiple topology domains:

$$\text{Cost}_{\text{locality}} = \sum_{i<j} w(d(g_i, g_j)) \tag{8.2}$$

where g_i and g_j are GPUs assigned to the job, $d(g_i, g_j)$ is their topological distance (0 for same node, 1 for same rack, 2 for different racks), and $w(\cdot)$ is a weighting function that increases with distance. The scheduler minimizes this score when selecting an allocation from the set of feasible placements. The weighting function encodes the performance impact of each topology level: $w(0) = 0$ (same node, NVLink, no penalty), $w(1) = 1$ (same rack, one switch hop), $w(2) = 10$ (different racks, multiple switch hops). The tenfold weight difference between same-rack and cross-rack placements reflects the disproportionate performance impact of crossing spine switches.

More sophisticated algorithms distinguish between parallelism strategies, recognizing that different communication patterns have different bandwidth and latency requirements. Tensor parallelism requires the highest bandwidth and lowest latency because it exchanges activation tensors at every layer; it should be confined to NVLink domains within a single node. Pipeline parallelism tolerates moderate latency because it sends activations between stages less frequently (once per microbatch per stage boundary rather than at every layer); it can span nodes within a rack where InfiniBand provides sufficient bandwidth. Data parallelism, with its bulk gradient synchronization

that can overlap with computation, operates efficiently across racks because the communication is less latency-sensitive and can use the available bandwidth effectively.

Algorithm 8.1 combines these pieces into a single decision: the all-or-nothing gang constraint of section 8.1.5, the locality cost of equation 8.2, and the parallelism-to-topology mapping just described. The scheduler enumerates the allocations that fit, scores each by communication cost, and takes the cheapest, leaving the job queued rather than stranding accelerators when none does.

Algorithm 8.1 Topology-aware placement

Require: job requesting N accelerators with tensor-, pipeline-, and data-parallel groups; cluster topology with per-level link costs $w(\cdot)$
Ensure: an all-or-nothing placement of least communication cost, or the job stays queued
1: enumerate the feasible slots: sets of N free accelerators that satisfy every resource dimension ▷ *gang: all or nothing*
2: **if** no feasible slot exists **then**
3: leave the job queued without holding any accelerators ▷ *prevents hold-and-wait deadlock*
4: **end if**
5: **for** each feasible slot **do**
6: map tensor-parallel ranks within an NVLink domain, pipeline stages within a rack, data-parallel replicas across racks
7: score the slot by $\text{Cost}_{\text{locality}} = \sum_{i<j} w(d(g_i, g_j))$
8: **end for**
9: **return** the feasible slot of least $\text{Cost}_{\text{locality}}$ ▷ *topology-optimal, atomic*

Scattered placement runs AllReduce 4.8× slower than a topology-aware slot, a gap figure 8.4 breaks out across four placement strategies from random to topology-optimal.

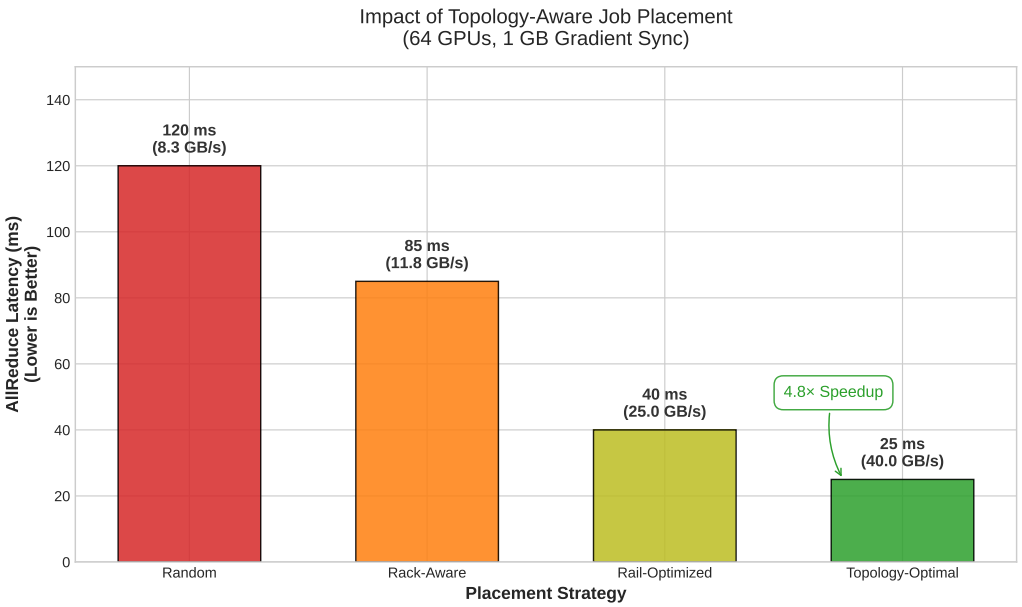


Figure 8.4: Topology Placement Impact: Reducing AllReduce latency by 4.8× through topology-aware scheduling (rack/rail optimization) compared to random placement for a 1 GB gradient sync on 64 GPUs.

A **hierarchical placement** algorithm assigns resources in layers that match this parallelism hierarchy. First, it assigns tensor parallel groups to individual nodes, ensuring that all GPUs in a tensor parallel group share NVLink connectivity. Second, it groups those nodes into pipeline stages within racks, placing consecutive pipeline stages on nodes connected through the same leaf switch.

Third, it distributes data parallel replicas across racks, ensuring that each data parallel group has access to the cluster's full bisection bandwidth for AllReduce operations. This three-level placement mirrors the three-level topology and the three-level parallelism strategy, creating a natural alignment between logical and physical structure.

Napkin Math 8.2: The topology placement impact

The physical location of GPUs on the network switch fabric dramatically impacts collective performance. Consider an AllReduce operation across 64 GPUs:

- **Random placement**: Spread across many racks → Traffic traverses core switches → 120 ms latency.
- **Rack-aware**: Confined to a single rack (Top-of-Rack switch) → 85 ms.
- **Rail-optimized**: Aligned to specialized "rails" (dedicated standard IB switches) → 40 ms.
- **Topology-optimal**: All GPUs on the same leaf switch → 25 ms. Proper scheduling yields a 4.8× speedup purely by respecting the network topology, with zero changes to the model code.

The computational cost of topology-aware placement is nontrivial. Evaluating all possible placements for a 256-GPU job across a 10,000-GPU cluster is combinatorially intractable. Production schedulers use heuristic approaches: greedy algorithms that build placements bottom-up (first fill NVLink domains, then fill racks, then spread across racks), tree-search algorithms with pruning that explore the most promising placements first, or scoring-based approaches that evaluate a sample of feasible placements and select the best. The scheduling latency introduced by topology-aware algorithms (milliseconds to seconds, depending on cluster size and algorithm complexity) is negligible compared to job runtimes of hours to weeks, making the throughput improvement worth the scheduling overhead.

Napkin Math 8.3: Placement impact on 3D-parallel training

Scenario: Consider a 256-GPU training job using 3D parallelism: 8-way tensor parallel, 4-way pipeline parallel, 8-way data parallel.

Good placement (topology-aware):

- Tensor parallel groups: 32 groups of 8 GPUs, each confined to one 8-GPU node (NVLink: 900 GB/s)
- Pipeline stages: 4 consecutive nodes within the same rack (1 switch hop, InfiniBand: 50 GB/s)
- Data parallel replicas: 8 racks (2 switch hops for cross-rack AllReduce)

Bad placement (topology-unaware):

- Tensor parallel groups split across 2 nodes (InfiniBand instead of NVLink)
- Pipeline stages scattered across racks (2 switch hops instead of 1)
- Data parallel replicas randomly distributed

Math: The tensor parallel communication alone illustrates the impact. With NVLink delivering 450 GB/s per direction, a 1 GB activation transfer takes approximately 2.2 ms. Over InfiniBand at 50 GB/s, the same transfer takes approximately 20 ms, a 9× slowdown.

Systems insight: The 4.8× AllReduce-latency gap in figure 8.4 is a communication-only measurement; it does not become a 4.8× end-to-end penalty because synchronous training overlaps much of that communication with the next iteration's compute. Since tensor parallelism communicates at every transformer layer (typically 80+ layers for large models), the residual

exposed communication compounds to 15 to 30 percent total training throughput degradation, the figure to carry forward for 3D-parallel jobs.

8.3.3 Rail-optimized scheduling

Large GPU clusters increasingly use **rail-optimized topologies** where each GPU in a node connects to a different network switch, creating parallel "rails" through the network fabric. This design, analyzed in section 3.5.3, provides balanced bandwidth across all GPUs by distributing traffic across independent switch hierarchies. However, it also creates a scheduling constraint: AllReduce operations are most efficient when communicating GPUs are on the same rail (connected to the same switch chain), because same-rail communication avoids cross-rail traffic that could create congestion at shared switch ports.

Rail-aware scheduling assigns data parallel replicas such that corresponding GPUs across nodes share the same rail. For an 8-GPU-per-node cluster with 8 rails, GPU 0 on every node connects to rail 0, GPU 1 to rail 1, and so on. The scheduler places data parallel rank r on GPU $r \bmod 8$ across selected nodes, ensuring that AllReduce for each data parallel group traverses only a single rail switch hierarchy rather than crossing between rails. This alignment means that AllReduce traffic is confined to independent switch trees, eliminating contention between data parallel groups and providing each group with the full bandwidth of its dedicated rail.

The interaction between rail-aware scheduling and topology-aware placement creates a multi-dimensional optimization problem. The scheduler must simultaneously optimize for intra-node NVLink locality (tensor parallelism requires GPUs on the same node), intra-rack switch locality (pipeline parallelism benefits from minimal switch hops between stages), and rail alignment (data parallelism benefits from same-rail communication). These three objectives are partially conflicting: constraining data parallel replicas to specific GPU positions within each node (for rail alignment) reduces the scheduler's flexibility to choose nodes based on rack locality, and requiring full-node allocations for NVLink locality prevents sharing nodes between jobs that could improve packing efficiency.

Production schedulers resolve these conflicts using weighted scoring functions that prioritize constraints based on workload characteristics. For large training jobs using 3D parallelism, tensor parallel locality receives the highest weight (because NVLink-to-InfiniBand performance degradation is the largest), followed by rail alignment (because it eliminates congestion for the most frequent communication pattern), followed by rack locality (because the performance benefit is smaller and can be partially compensated by compute-communication overlap). For pure data parallel jobs, rail alignment receives the highest weight. For single-node jobs, topology constraints are irrelevant and the scheduler optimizes purely for packing efficiency.

8.3.4 Topology discovery and dynamic adaptation

Static configuration files like Slurm's `topology.conf` or Kubernetes node labels provide a baseline map of the cluster, but they represent the *intended* state, not the *effective* state. True **topology discovery** requires runtime introspection to validate that the physical hardware matches the architectural diagram. Libraries like NCCL bypass static configurations entirely, parsing the Linux virtual filesystem at `/sys/class/infiniband` to map the PCIe bus hierarchy, enumerate NVLink connections, and verify NIC proximity. This creates a ground-truth graph where edges represent verified, operational links rather than theoretical cables.

Cluster topology is an operating state, not a fixed schematic. Hardware failures, maintenance windows, and thermal throttling constantly reshape the effective topology available to the scheduler. A single spine switch failure in a Clos network does not sever connectivity entirely, but it drastically reduces the bisection bandwidth available to jobs spanning the affected racks. If a spine switch serving our 175B parameter model's data-parallel AllReduce group fails, cross-rack bandwidth might drop by 50 percent. The scheduler faces a critical optimization decision: continue training at reduced throughput, effectively wasting 25 percent of the expensive GPU cycles waiting for straggling gradients, or initiate a topology-aware migration.

Migration is not free; it requires a coordinated checkpoint, a job kill, a rescheduling event on healthy nodes, and a warmup phase. This process typically consumes 5 to 15 minutes of idle time.

However, for a multi-week training run, the math often favors migration. If the network degradation creates a 20 percent throughput penalty, the break-even point for a 10-minute migration is less than an hour of training. Sophisticated orchestrators continuously monitor effective bandwidth metrics reported by the collective communication library. When the delta between the assigned topology score and the realized throughput exceeds a threshold, the scheduler triggers a drain-and-migrate workflow, treating network degradation with the same severity as a GPU hardware fault.

Example 8.2: The limping switch

Scenario: A foundation-model training job shows a large throughput regression. The GPUs are healthy, the code is unchanged, and the static topology map says the job has full bisection bandwidth.

Failure mode: A single switch or transceiver can remain technically "up" while producing high CRC error rates, packet drops, or retransmissions. Binary health checks report green; collective communication performance reports red.

Consequence: If the scheduler relies on a static topology map, it continues to schedule communication-intensive jobs onto a degraded rack, assuming full bandwidth that no longer exists.

Systems insight: Binary health checks (up/down) are insufficient. Schedulers need performance-aware topology monitoring that incorporates switch counters, retransmission rates, and collective bandwidth tests. Nodes connected to a "limping" switch should be dynamically marked ineligible for communication-intensive jobs until the hardware is repaired.

The general scheduling idea is quarantine: when runtime evidence says a node, rack, or link is degraded, the scheduler should stop placing sensitive work there even if the nominal resource count still looks available. Kubernetes implements this pattern through **taints and tolerations**[12], while Slurm clusters commonly use drain states, node features, or partition rules. The complementary problem is how jobs adapt when available resources change dynamically. Rather than treating resource allocation as fixed for a job's lifetime, elastic training allows jobs to grow, shrink, and recover without full restarts.

12 **Taints and Tolerations**: Kubernetes primitives for restricting pod placement. A taint on a node repels all pods that do not explicitly tolerate it. For ML fleets, this is the primary mechanism for isolating specialized hardware: by tainting A100 nodes, the orchestrator prevents general-purpose web services from "stealing" expensive GPU capacity, ensuring those nodes are reserved for training jobs that tolerate the taint.

8.4 Elastic Scheduling

When cluster demand fluctuates, rigid allocation creates two categories of waste. During off-peak hours, a job running on 256 GPUs cannot absorb idle resources to speed up training. During peak demand, the same job cannot release excess resources to accommodate higher-priority work without being fully preempted. The scheduler's only options are "let it keep all its resources" or "kill it entirely," with nothing in between.

Elastic scheduling eliminates this binary choice. The recovery mechanics of elastic training developed in section 7.8 show how a distributed job can adjust its worker count, recalibrate batch size and learning rate, and reconstruct its communication group after a failure. From the scheduler's perspective, the same capability solves a resource-allocation problem under fluctuating demand. A training job might start with 128 GPUs (the minimum needed for reasonable throughput), scale up to 256 when resources become available, and scale back to 192 when higher-priority work arrives, all while maintaining continuous training progress. This flexibility transforms the scheduler's action space from a *binary* {continue, preempt} to a *continuous* spectrum of resource allocation levels.

8.4.1 The scheduler-framework contract

Elastic scheduling requires a well-defined contract between the scheduler and the training framework. The scheduler guarantees that resource changes are communicated through defined signals, with sufficient advance notice for graceful adaptation. The framework guarantees that it can handle resource changes without corrupting model state, using the batch size adjustment, learning rate recalibration, and communication group reconstruction mechanisms developed in section 7.8.1.

From the scheduler's side, the contract has three parameters:

- **Elastic range**: $[N_{\text{min}}, N_{\text{max}}]$ specifies the worker counts between which the job can operate.

- **Transition cost**: The rendezvous cycle, typically 10 to 60 seconds, determines how frequently the scheduler can resize without the overhead negating the throughput gain.
- **Scaling efficiency curve**: The throughput gained from each added worker determines whether assigning idle GPUs to this job is more productive than holding them for a pending gang-scheduled job.

The contract gives the scheduler enough information to resize a job without treating elasticity as free capacity.

The scheduler must also account for the constraint that elastic scaling changes the effective batch size ($B_{\text{effective}} = B_{\text{per-worker}} \times N$). The two strategies for handling this (constant global batch size vs. adaptive learning rate) impose different limits on how aggressively the scheduler can resize: constant global batch size preserves convergence but may violate per-GPU memory limits at low worker counts, while adaptive learning rate simplifies worker management but introduces convergence risk during rapid scaling.

The scheduler interface to frameworks like TorchElastic exposes these parameters directly, as listing 8.2 illustrates.

Listing 8.2: TorchElastic Launch Configuration: The `--nnodes=4:16` range tells the scheduler that this job can operate with 32 to 128 GPUs. The scheduler places the job at any point within this range based on current cluster load and resizes it as demand changes.

```
# Launch elastic training with min 32 to max 128 workers
torchrun \
  --nnodes=4:16 \
  --nproc_per_node=8 \
  --rdzv_backend=c10d \
  --rdzv_endpoint=head-node:29500 \
  --max_restarts=3 \
  train.py
```

The `--nnodes=4:16` specification declares that the job can operate with anywhere from 4 to 16 nodes (32 to 128 GPUs with 8 GPUs per node). This range is the scheduler's action space: it can place the job at any point within this range based on current cluster load, and resize it as demand changes.



Systems Perspective 8.4: Elastic vs. rigid scheduling

Consider a congested cluster where a 128-GPU job waits 8 hours for a full gang allocation.

Rigid scheduling: Wait 8 hours, then train for 24 hours at full throughput. Total wall-clock: 32 hours.

Elastic scheduling (min 32, max 128):

- Start immediately with 32 GPUs (throughput = 25 percent of full)
- After 2 hours, scale to 64 (throughput = 50 percent)
- After 4 hours, scale to 128 (throughput = 100 percent)
- Training progresses during the entire wait period

Approximate work completed during the 8-hour "queue wait":

- Hours 0 to 2: 25% throughput $\times$ 2 hours = 0.5 hours equivalent
- Hours 2 to 4: 50% throughput $\times$ 2 hours = 1.0 hours equivalent
- Hours 4 to 8: 100% throughput $\times$ 4 hours = 4.0 hours equivalent
- Total: 5.5 hours of equivalent full-scale work

The elastic job completes in approximately 26.5 hours (8 hours of scaled-up training plus 18.5 hours at full scale), saving 5.5 hours compared to rigid scheduling. This 17 percent improvement

> comes entirely from using resources productively during what would otherwise be idle queue time.

The scheduler cannot treat every job as elastic. The most significant constraint is incompatibility with model parallelism. Elastic scaling is fundamentally a data-parallel operation: the scheduler adds or removes replicas of the model, changing the global batch size but keeping the model architecture constant. For a 175B-parameter job that relies on 8-way tensor parallelism within each node, elasticity operates strictly at the *node* level. The job can scale between 128 nodes (1,024 GPUs) and 32 nodes (256 GPUs) in whole-node steps, changing the data-parallel degree. It *cannot* elastically remove a single GPU from a tensor-parallel group; doing so would require re-sharding the model weights, equivalent to a full checkpoint-and-restart cycle. Pipeline parallelism permits limited elasticity in principle, but adding or removing stages forces a recalculation of the pipeline schedule that often negates the benefit.

The transition cost also constrains the scheduler's resize frequency. Each scaling event incurs 10 to 60 seconds of reduced throughput while the framework reconstructs its communication group. The scheduler must amortize this cost over the subsequent run duration, so small resize increments are only worthwhile if the job will run at the new scale for significantly longer than the transition window; section 8.4.3 works through the thrashing regime this rule rules out.

8.4.2 Scheduler integration

Elastic training's value depends critically on scheduler integration. Without scheduler awareness, elastic training is merely a *fault tolerance mechanism* (replacing failed workers). With scheduler integration, it becomes a *scheduling optimization* that fundamentally changes the economics of cluster utilization. The scheduler must understand that elastic jobs can operate within a range of resource allocations, specified as $[N_{\text{min}}, N_{\text{max}}]$ workers, enabling three key scheduling optimizations.

The most immediately valuable benefit is faster job start. An elastic job requesting 32 to 128 GPUs can start as soon as 32 GPUs are available, rather than waiting in the gang scheduling queue for 128 contiguous GPUs. In congested clusters where large-job queue times can be hours to days, starting immediately at reduced scale often completes the job sooner than waiting for full-scale allocation, a trade-off the elastic scaling decision below works through quantitatively. The scheduler can compute it dynamically, choosing the start time and initial scale that minimizes expected total time (wait time plus execution time).

Opportunistic scaling enables the scheduler to use elastic jobs as flexible capacity absorbers. When resources become available (other jobs complete, spot instances are provisioned, preempted jobs release resources), the scheduler can expand running elastic jobs to improve their throughput. When resources are needed (higher-priority job arrives, reserved capacity is reclaimed), the scheduler can shrink elastic jobs without preempting them entirely. This bidirectional flexibility converts elastic jobs into a buffer that absorbs utilization variance, smoothing the gap between allocated and productive capacity.

Graceful preemption transforms the scheduler's response to resource pressure. Instead of killing a training job outright to reclaim resources, the scheduler can request a scale-down, reducing the job's resource allocation to release capacity for higher-priority work. The job continues with fewer workers, maintaining progress at reduced throughput rather than losing all progress since the last checkpoint and incurring restart overhead. This provides the scheduler with a continuous spectrum between "full resources" and "fully preempted," rather than the binary choice of traditional scheduling. Graceful preemption composes well with checkpoint-based preemption: if the scheduler needs to reclaim all resources, it first scales the job down to its minimum size, then initiates a checkpoint-and-preempt sequence, minimizing the amount of lost work.

The trade-off for all of these benefits is complexity. Elastic training requires framework support (not all training frameworks support it), introduces potential convergence concerns from batch size changes (requiring validation that elastic scaling does not degrade model quality), and complicates performance modeling (a job's throughput is no longer constant, making capacity planning harder). Not all workloads benefit equally from elastic scaling: jobs with high communication-to-computation ratios see diminishing returns from additional workers (because communication overhead grows

with worker count) and thus benefit less from scale-up, while compute-bound jobs with low communication overhead scale more linearly and benefit more from opportunistic scaling. The scheduler must model these scaling characteristics per-job to make optimal elastic scaling decisions.

8.4.3 Elastic scaling policies

Adding resources to a running job does not always accelerate time-to-convergence. The naive assumption that throughput scales linearly with worker count collapses under the weight of communication overhead, requiring the scheduler to enforce elastic scaling policies based on empirical efficiency curves rather than resource availability alone. A job's scaling efficiency $\eta_{\text{scaling}}(k)$, defined as the ratio of observed throughput at k workers to the ideal linear speedup, typically exhibits three distinct phases: a linear phase where compute dominates communication, a sublinear phase where gradient synchronization latency begins to mask compute, and a saturation phase where the AllReduce ring becomes the bottleneck. A scheduler that blindly assigns available GPUs to a job in the saturation phase wastes cluster capacity that could have cleared the queue; conversely, running a job below its **minimum viable scale** forces it to hold onto memory and network leases for disproportionately low progress.

The decision to scale is therefore a function of marginal utility, not vacancy alone. For our 175B parameter model, the efficiency curve is derived from profiling: it scales linearly from 64 to 512 GPUs, becomes sublinear between 512 and 1,024, and hits diminishing returns beyond 1,024. Consequently, the scheduler enforces an elastic range of $[256, 1024]$. If the cluster has free resources but the job is already at 1,024 GPUs, the scheduler withholds the extra nodes, predicting that the 5 percent marginal throughput gain does not justify the re-sharding overhead or the opportunity cost of starving a pending job. Similarly, if only 128 GPUs are available, the scheduler may choose to queue the job rather than run it, as the training intensity per GPU would be too low to hide the communication latency.



Elastic scaling's benefit flattens into saturation.

This logic extends to the temporal dimension, where the scheduler must weigh the immediate progress of a smaller allocation against the deferred gratification of waiting for a larger one. If a 512-GPU job can start immediately with 128 GPUs, or wait two hours for the full request, the optimal choice depends on the integration of throughput over time. The preemption overhead (the cost of stopping, checkpointing, and restarting to resize) must be amortized over the subsequent run duration. A policy of "opportunistic scaling" that resizes a job every time a single node becomes free often yields net negative progress due to the constant thrashing of the distributed process group.

Napkin Math 8.4: The elastic scaling decision

Consider a large language model training job with a target batch size that requires 512 A100s for peak efficiency. Three scheduling strategies are evaluated over a window of 24 hours.

Given: Throughput at 128 GPUs is 1 epoch/hour (baseline). Throughput at 256 GPUs is 1.8 epochs/hour (90 percent scaling efficiency). Throughput at 512 GPUs is 3.2 epochs/hour (80 percent scaling efficiency). Re-scaling cost is 10 minutes (0.17 hours) to checkpoint, re-shard, and restart.

Strategy 1 – immediate start (no scaling): The job starts immediately with 128 GPUs and runs for the full 24 hours. Total work: $24 \times 1.0 = 24$ epochs.

Strategy 2 – wait for capacity (gang scheduling): The job waits in the queue for 4 hours until 512 GPUs become available, then runs for 20 hours. Total work: $20 \times 3.2 = 64$ epochs.

Strategy 3 – elastic scaling (step-up): The job starts immediately with 128 GPUs. After 4 hours, 256 GPUs become available. The job pauses, scales up, and resumes. Phase 1: $4 \times 1.0 = 4$ epochs. Phase 2: $(24 - 4 - 0.17) \times 1.8 \approx 35.7$ epochs. Total work: 4 + 35.7 = 39.7 epochs.

The elastic strategy completes nearly 1.7× the epoch-equivalent work of the immediate small-scale run, but still trails gang scheduling by about 38 percent. That gap shrinks when queue waits grow or peak capacity stays scarce; re-scaling overhead is only 10 minutes, and if that tax exceeded 2 hours (due to slow checkpoint transfer or compilation), Strategy 3 would fall further behind Strategy 2.

The placement and elasticity mechanisms discussed so far address individual job efficiency, optimizing how each job uses its allocated resources. Fleet economics shifts the objective to the collective level: an allocation policy must maximize useful work across all jobs when every placement, preemption, and reservation creates opportunity cost.

8.5 Cost Optimization

The utilization-to-dollars relationship established in section 8.1 now becomes the orchestrator's objective function. A 10,000-GPU cluster represents a capital investment of roughly $300 million and consumes millions more in monthly power and cooling. If poor scheduling leaves 20 percent of those GPUs idling in fragmentation gaps or waiting for data, roughly $60 million in deployed capital sits unused, and at cloud-equivalent rates the same idleness burns approximately $35 million per year in operating cost. Cost optimization forces the fleet orchestrator to treat the cluster as a massive financial asset whose return on investment (ROI) must be maximized.

Figure 8.5 makes this waste concrete by plotting annual wasted cost as a function of cluster size and utilization level. At 10,000 GPUs with 50 percent utilization, the annual waste exceeds $87 million under the chapter's $2/GPU-hour accounting scenario. Published measurements from production clusters show that underutilization is a real operational problem: Microsoft's Philly cluster study (Jeon et al. 2019) measured mean GPU utilization of approximately 52 percent, and Li et al. (2023) found that 50 percent of jobs on NERSC's Perlmutter supercomputer used less than 25 percent of allocated GPU memory. The 30 to 70 percent shaded band in the figure is an illustrative operating range for the cost model, not a universal empirical claim about all production clusters. Moving from 50 percent to 70 percent utilization on a 10,000-GPU cluster saves over $35 million annually in this scenario without purchasing a single additional GPU.

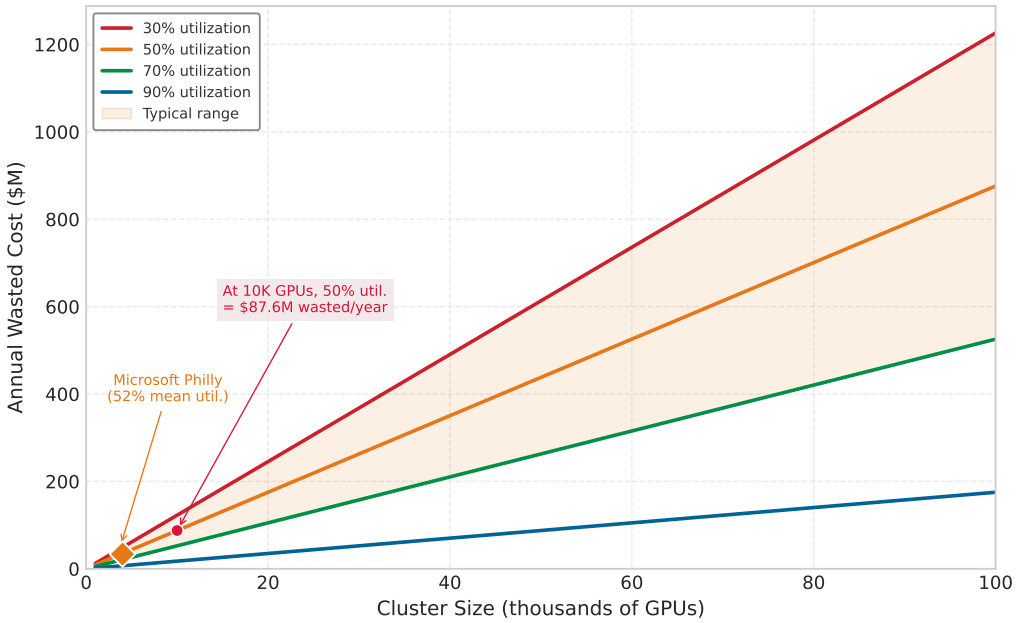


Figure 8.5: The Cost of Idle GPUs: Annual wasted cost as a function of cluster size for different GPU utilization levels, assuming $2/GPU-hour (cloud equivalent). The shaded region between 30 percent and 70 percent is an illustrative operating range for the scenario analysis. At 10,000 GPUs with 50 percent utilization, the model yields $87.6M in annual wasted capacity. The Microsoft Philly data point marks the published mean GPU utilization of 52 percent. Data sources: (Jeon et al. 2019; Li et al. 2023).

The idle-capacity figure establishes the economic target; the remaining cost controls explain how the scheduler moves toward it. Interruptible capacity lowers the price of tolerant work, reservation and quota policy decide which jobs deserve scarce guaranteed capacity, and accountability mechanisms make idle GPUs visible to the teams that create demand.

8.5.1 Spot instances and preemptible VMs

Cloud providers offer **spot instances**[13] (AWS) or **preemptible VMs** (GCP) at 60 to 90 percent discounts compared to on-demand pricing, with the caveat that instances can be reclaimed with as little as 30 seconds to 2 minutes notice when the provider needs the capacity for on-demand customers. At \$0.70/GPU-hour instead of \$2/GPU-hour, spot instances transform the economics of large-scale training, potentially reducing the cost of a multi-week pretraining run from hundreds of thousands of dollars to under one hundred thousand.

The viability of spot instances for ML training depends on two factors that determine whether the discount translates into actual savings: the cost of each interruption (checkpoint overhead plus restart time plus lost work since the last checkpoint) and the frequency of interruption (which varies by instance type, region, time of day, and overall cloud demand). Equation 8.3 gives the **effective cost** of spot training:

$$C_{\text{effective}} = C_{\text{spot}} \times \frac{T_{\text{total}}}{T_{\text{productive}}} \tag{8.3}$$

where T_{total} includes productive training time plus checkpoint overhead plus restart overhead after interruptions, and $T_{\text{productive}}$ is the time spent advancing training. If interruptions are rare (less than once per day) and checkpointing is efficient (less than 5 percent overhead), effective cost remains close to the spot price and the savings are nearly the full discount. If interruptions are frequent (every few hours) and restarts are expensive (large model, slow checkpoint loading, cold GPU cache), effective cost can approach or even exceed on-demand pricing, turning the apparent discount into a hidden premium.

The relationship between spot savings and fault tolerance infrastructure is circular in an important way. The fault tolerance mechanisms from Chapter 7 (frequent asynchronous checkpointing, fast checkpoint loading, elastic recovery) directly enable spot instance usage by minimizing the cost of each interruption. Elastic training allows jobs to continue with reduced worker count when some spot instances are reclaimed, avoiding full restart entirely. Organizations that invested in fault tolerance for reliability reasons discover that the same infrastructure unlocks significant cost savings through spot instance utilization, providing a return on their fault tolerance investment that goes far beyond avoiding lost work from hardware failures.

The strategic insight is that fault tolerance infrastructure has a double return: it both protects against involuntary losses (hardware failure) and enables voluntary cost savings (spot instances). This double return often justifies fault tolerance investments that would be hard to justify on reliability grounds alone, because the cost savings from spot utilization are concrete and measurable while the cost avoidance from hardware failure protection is statistical and harder to quantify.

8.5.2 Spot instance strategies for ML training

To maximize spot instance utility while minimizing disruption, sophisticated schedulers employ diversification and predictive strategies that go beyond simple "retry on failure" logic. Availability zone diversification reduces the probability of simultaneous fleet-wide reclamation. Cloud providers typically manage spot pools independently per availability zone (AZ). If a training job requires 1,024 GPUs, allocating them as a single block in one zone exposes the job to a complete stop if that specific zone's spot pool is reclaimed. Spreading the job across multiple AZs ensures that a reclamation event in one zone affects only a fraction of the fleet. The scheduler still must model correlation carefully: independent per-instance interruptions make large simultaneous losses unlikely, but pool-level or AZ-level reclamations can remove hundreds of GPUs at once and must be treated as elastic-shrink or failover events rather than ordinary isolated failures.

Instance type diversification exploits independent spot markets across GPU SKUs. A job that strictly requests one instance type competes in a single, crowded market. A job that accepts multiple equivalent instance types dramatically increases its scheduling probability. The scheduler should maintain a priority-ordered list of acceptable instance types, falling back to larger-memory instances when the primary type is unavailable. While this mix requires careful peer-to-peer bandwidth management, it effectively uses the "upgrade" to maintain progress at a blended cost still far below on-demand rates.

[13] **Spot Instances**: Named after commodity trading's "spot price" (the current market price for immediate delivery), spot instances sell spare cloud capacity at 60 to 90 percent discounts with the provider retaining reclamation rights. The critical asymmetry for ML training is the interruption notice window: AWS provides 2 minutes, GCP provides 30 seconds. A 175B-parameter model requires 3 to 5 minutes to checkpoint, so GCP's 30-second notice forces either continuous checkpointing overhead or acceptance of lost work on every interruption.

Spot interruption prediction uses the data that cloud providers expose about reclamation likelihood, such as the AWS Spot Placement Score or GCP preemptibility data. Advanced schedulers ingest this feed to estimate the expected interruptions per day for a given instance type and region. If the expected interruption rate rises above a threshold where the overhead of restarts exceeds the cost savings (typically more than 4 interruptions per day for large models), the scheduler should automatically migrate the job to on-demand capacity or a different region, preventing thrashing where a job spends more time recovering than training.

Checkpoint frequency optimization for spot instances differs from the standard reliability logic. The Young-Daly formula developed in section 7.1.2 optimizes for unpredicted hardware failures. Spot interruptions, however, come with a warning: 2 minutes on AWS, 30 seconds on GCP. The optimal strategy is a **two-tier checkpointing** approach: a periodic checkpoint at the Young-Daly interval to protect against hard crashes, combined with an immediate checkpoint triggered by the termination notice. This "panic checkpoint" captures the exact state of training seconds before the node vanishes, reducing lost work to near zero and transforming preemption from a rollback event into a pause-and-resume event.

The same need for contingency planning becomes acute when spot interruptions are correlated rather than isolated.



Systems Perspective 8.5: Correlated spot risk

Context: Spot capacity is cheap because it is reclaimable. That risk is not independent across instances: cloud regions, instance types, product launches, and deadline cycles can create correlated demand.

Failure mode: A fleet that concentrates all interruptible training in one region and one instance type can lose a large fraction of workers at once, even if each individual worker appears restartable.

Consequence: Checkpointing handles isolated interruptions; it does not solve fleet-wide preemption when every retry targets the same exhausted pool.

Systems insight: A robust scheduler must diversify across regions and instance types, monitor interruption risk, and maintain a "break glass" path to on-demand capacity when the spot market becomes toxic.

8.5.3 Capacity reservation strategies

Organizations that rely on cloud infrastructure (or that think about on-premise infrastructure in economic terms) balance three tiers of compute capacity, each offering a different trade-off between cost, availability, and commitment. Table 8.4 compares those tiers across cost, availability, interruption risk, and workload fit.

Table 8.4: Cloud Capacity Reservation Tiers: Capacity tiers trade cost and commitment against availability guarantees and interruption risk, giving the scheduler a price ladder for routing workloads by urgency and restart tolerance.

Capacity tier	Cost and commitment	Availability and interruption risk	Best-fit workloads
Reserved capacity	One- to three-year commitments, with 30–60% discounts versus on-demand.	Guaranteed availability for the committed baseline.	Predictable continuous workloads: 24/7 inference, scheduled training, CI and regression testing, SLA-bound jobs.
On-demand capacity	Full price with no long-term commitment.	Immediate provisioning when queuing or interruption is unacceptable.	Urgent retraining, interactive debugging, and short-lived jobs that do not justify spot management.
Spot capacity	Discounted interruptible capacity.	Availability varies with cloud demand, and workers can be reclaimed.	Restart-tolerant jobs: hyperparameter sweeps, ablations, checkpointed pretraining, and exploratory research.

The optimal mix depends on workload characteristics and organizational priorities. An organization with 60 percent steady-state utilization and 40 percent burst demand might reserve capacity for 60 percent of peak need, use on-demand for latency-sensitive bursts that require immediate provisioning, and use spot for the remainder. The scheduling system must understand these capacity tiers and route workloads accordingly: inference workloads to reserved capacity for guaranteed availability, large training jobs to a mix of reserved (for the base allocation) and spot (for additional workers via elastic scaling), and exploratory work to spot exclusively where the cost savings are maximized and the interruption tolerance is highest.

Systems Perspective 8.6: Spot vs. on-demand training

Consider training a 70B parameter model requiring 512 GPUs for 14 days.

On-demand: 512 GPUs $\times$ 336 hours $\times$ \$2/GPU-hour = \$344,064

Spot (65 percent discount, 5 percent checkpoint overhead, 14 interruptions/day with 30 minutes restart):

- Base cost: 512 GPUs $\times$ 336 hours $\times$ \$0.70/GPU-hour = \$120,422
- Checkpoint overhead: $5\% \times$ 336 hours = 16.8 hours
- Restart overhead: 196 interruptions $\times$ 0.5 hours = 98 hours
- Total time: 336 hours + 16.8 hours + 98 hours = 450.8 hours
- Total cost: 512 GPUs $\times$ 450.8 hours $\times$ \$0.70/GPU-hour = \$161,567

Savings: \$182,497 (53.0 percent), at the cost of approximately 34.2 percent longer wall-clock time.

The economics are compelling when fault tolerance infrastructure is already in place. Without checkpointing, a single interruption forces restart from scratch, potentially wasting days of compute and negating all savings.

8.5.4 Scheduling for cost efficiency

Cost-aware scheduling extends the scheduler's optimization objective beyond utilization and fairness to include financial efficiency. Traditional schedulers minimize a single objective (for example, average job completion time or maximum wait time), but cost-aware schedulers must navigate multi-objective trade-offs where time, money, and reliability are all decision variables. The scheduler must routinely choose among three decision classes:

- **Wait vs. run now**: Run a job on reserved A100s now, or wait 2 hours for spot H100s that would complete the job 3$\times$ faster at half the cost.
- **Preempt vs. pay more**: Preempt a low-priority hyperparameter sweep to make room for a high-priority training run on reserved capacity, or place the training run on more expensive on-demand instances.
- **Fallback vs. elastic shrink**: When spot capacity is reclaimed, migrate the job to on-demand to maintain throughput at higher cost, or scale down elastically to maintain low cost at reduced throughput.

Each decision prices time, money, and reliability differently, so cost-aware scheduling is policy design rather than simple bin packing.

These decisions require the scheduler to model both the time-value of computation (the organizational cost of a 2-hour delay in researcher productivity, product launch schedules, or competitive position) and the financial cost of different resource allocations. The time-value varies enormously by workload type: delaying a production model retrain by 2 hours might violate a service level agreement (SLA) and cost thousands in penalty fees, while delaying an exploratory experiment by 2 hours costs nothing but researcher patience.

Production systems typically define **cost policies** per workload class that encode these trade-offs as configuration rather than requiring per-decision human judgment. Inference workloads are pinned to reserved capacity for guaranteed availability and predictable latency, since the cost of

an inference outage (lost revenue, degraded user experience) far exceeds the premium of reserved pricing. Large training runs use a mix of reserved capacity (for the minimum viable allocation) and spot capacity (for elastic expansion), with automatic fallback to on-demand if spot is reclaimed and the job is within a configurable deadline. Exploratory workloads are restricted to spot to minimize cost, with the understanding that they may be interrupted and must tolerate variable completion times.

Technical optimization must be paired with financial accountability through **ML FinOps**. In many organizations, GPU costs are pooled into a generic "infrastructure" bucket, creating a tragedy of the commons where teams lack visibility into their resource consumption and have no incentive to optimize. Effective governance requires granular tagging to implement **showback** (reporting costs to teams) or **chargeback** (billing teams' budgets). Metrics must evolve from "GPU hours" to business-aligned units: "Cost per Model Version," "Cost per Experiment," and "Cost per 1M Predictions." This shift transforms cost from an opaque infrastructure expense into a measurable input to product decisions, enabling teams to make informed trade-offs between model quality, training speed, and infrastructure cost. Section 12.7.8 develops the platform machinery that makes these costs visible and attributable; the concern here is narrower, namely how that price signal reshapes the scheduling and quota behavior of the teams competing for the fleet.

8.5.5 The total cost of ownership

Optimizing for spot instance availability or scheduling density addresses only the visible tip of the infrastructure iceberg. The full cost of a machine learning fleet extends far beyond GPU-hours: in high-performance clusters the compute silicon typically accounts for only 40 to 60 percent of total system cost. The remainder is consumed by the supporting infrastructure required to keep that silicon fed and cool: high-bandwidth networking (InfiniBand switches, optics, and cabling), parallel file systems for checkpointing, and specialized power and cooling delivery. A single rack of H100 nodes can draw upwards of 40 kW, ten times the density of a standard web server rack, forcing facilities to deploy liquid cooling or rear-door heat exchangers that reshape the facility's capital requirements. A scheduler that optimizes GPU-hours alone is therefore blind to most of what the fleet actually costs.

Whether that fixed capacity is worth owning is a question of utilization. Section 12.3 develops the full build-versus-buy analysis, amortizing three years of capital and operational expenditure against cloud rental to find the break-even utilization that decides between them. The consequence for the scheduler is direct: owned hardware only amortizes when it stays busy, so every point of utilization the scheduler reclaims moves the fleet toward the break-even point that justifies its capital, while a cluster left idle by poor packing or fragmentation makes owned infrastructure more expensive than the cloud it was meant to undercut.

> Checkpoint 8.2: Cost-aware scheduling trade-offs
>
> Before moving to ML-specific schedulers, review how the cost-optimization mechanisms interact:
>
> - ☐ **Spot break-even**: Determine the checkpoint frequency under which spot training becomes more expensive than on-demand for a 512-GPU, 14-day training job with a 65 percent spot discount, 1 interruption per day, and 30-minute restart overhead.
> - ☐ **Double return**: Explain why fault tolerance infrastructure has a "double return" for reliability and cost savings, and identify the minimum fault tolerance investment needed to make spot instances viable.
> - ☐ **Reservation sizing**: Determine whether increasing reserved capacity from 40 percent to 60 percent would save or waste money for an organization.

Spot pricing and reservation strategy squeeze the cost of running a job, but they leave a second source of efficiency untouched: the learning dynamics inside the training loop. The cost models so far optimize time-to-completion, yet the metric that matters to a research organization is time-to-accuracy, the wall-clock time until a run reaches a target validation metric. A scheduler blind to

convergence cannot tell whether a run is in its steep early phase or its diminishing-returns tail, so it cannot redirect resources to where they buy the most progress. Closing that gap is the job of the ML-aware schedulers examined next.

8.6 Custom ML Schedulers

General-purpose schedulers treat training jobs as opaque boxes, so they cannot tell whether a model is in the steep early phase of learning or the final diminishing-returns phase. Custom ML schedulers trade implementation complexity for that visibility: they peer inside the training loop to optimize time-to-accuracy rather than only time-to-completion. The selection question is which internal signal addresses the fleet's binding bottleneck, and the four systems differ less by scheduler sophistication than by the signal they ask the platform to trust: consumed service, iteration boundaries, completion value, or measured goodput.

8.6.1 Tiresias: Duration-agnostic scheduling

Tiresias (Gu et al. 2019) addresses the fundamental problem identified in section 8.1.5: ML job durations are unpredictable, and scheduling policies that rely on exact duration estimates can make poor decisions when those estimates are unavailable or unreliable. Rather than fighting this information asymmetry, Tiresias eliminates the requirement for duration estimates entirely. It uses a **two-dimensional attained service**[14] scheduler that makes priority decisions based on what a job has *already consumed*, not what it *claims it will consume*. Jobs accumulate "service" based on GPU-time consumed, with priority decreasing as service increases. The two dimensions are time (how long the job has been running) and resources (how many GPUs the job uses), capturing both elapsed duration and resource intensity.

A discretized version groups jobs into service bins (for example, less than 1 GPU-hour, 1 to 10 GPU-hours, 10 to 100 GPU-hours, greater than 100 GPU-hours), promoting jobs in lower bins to the front of the queue. This bin structure means that all short jobs (the majority of submitted work) receive high priority and run quickly, while the few long jobs that consume most cluster resources gradually lose priority and are deprioritized relative to new arrivals. The key insight is that this behavior approximates the theoretically optimal Shortest Remaining Processing Time (SRPT) policy without requiring the future knowledge that SRPT demands: jobs that have consumed little service are statistically likely to be short jobs, so prioritizing them is a good heuristic for minimizing average completion time.

Experiments on production cluster traces from Microsoft and Alibaba show 40 to 60 percent reduction in average job completion time compared to FIFO scheduling, with the largest improvements for short jobs that previously waited behind long-running training runs. The improvement is not free: long-running training jobs experience increased completion times because they are systematically deprioritized. However, the aggregate benefit is positive because there are many more short jobs than long ones, and the short-job improvement exceeds the long-job penalty in total.

14 **Attained Service Scheduling**: A discipline from OS queuing theory where priority decreases with cumulative resource consumption, approximating the theoretically optimal Shortest Remaining Processing Time (SRPT) policy without requiring the future knowledge that SRPT demands. In ML clusters, this approximation is particularly effective because job durations are heavy-tailed: the majority of submitted jobs are short experiments, so deprioritizing high-service jobs correctly identifies the long-running outliers without requiring users to provide runtime estimates they cannot accurately compute.

8.6.2 Gandiva: Iteration-aware scheduling

Gandiva (Xiao et al. 2018) exploits a characteristic that general-purpose schedulers completely ignore: the iterative nature of deep learning training. Each training iteration follows a predictable, repeating pattern: a GPU-intensive forward pass, a GPU-intensive backward pass, a communication-intensive gradient synchronization, and then a CPU-intensive data loading and preprocessing phase before the next iteration. During the data loading phase, the GPU sits partially idle, computing at less than full utilization while waiting for the next batch of data to be prepared.

Gandiva uses **iteration-boundary time slicing**, enabling higher utilization through controlled oversubscription. A cluster with 100 GPUs might support 120 concurrent jobs if each spends 20 percent of its iteration time waiting for data, because the scheduler can interleave data loading from one job with GPU computation from another on the same device. The critical insight that makes this practical is that iteration boundaries provide natural preemption points where GPU state is minimal. Between iterations, only model weights and optimizer state reside on the GPU; the intermediate activations from forward and backward passes have been freed. This minimal state makes context switching cheap (seconds rather than minutes), unlike preempting mid-iteration when the full activation memory is in use.

Gandiva also implements **grow-shrink elasticity** at a finer granularity than the framework-level elastic training discussed in section 8.4. Gandiva automatically adjusts data parallelism degree based on real-time resource availability, using profiled iteration times to predict the throughput impact of adding or removing workers. When a high-priority job arrives, Gandiva shrinks lower-priority jobs by reducing their worker count rather than killing them entirely, preserving their progress while freeing resources. When resources become available again, it grows jobs back to their preferred parallelism degree. This fine-grained elasticity, informed by actual iteration-level profiling data, enables scheduling decisions that general-purpose systems cannot make because they lack visibility into job internals.

8.6.3 Themis: Finish-time fairness

Traditional fair-share scheduling treats all GPU-seconds equally: a job consuming 100 GPU-hours is treated the same whether those hours represent the first 10 percent of a long training run or the last 10 percent of a nearly complete one. From a resource accounting perspective, 100 GPU-hours is 100 GPU-hours regardless of context. **Themis** (Mahajan et al. 2020) argues this resource-centric view is fundamentally unfair because it ignores the sunk cost of work already performed.

Themis defines a **finish-time fairness** metric that allocates resources to minimize the maximum slowdown any job experiences relative to exclusive access (the hypothetical scenario where the job has the entire cluster to itself). Under this metric, a job that is 90 percent complete and needs only 10 more GPU-hours to finish receives higher priority than a job that is 10 percent complete and needs 90 more GPU-hours. The reasoning is economic: delaying the nearly-complete job by an hour wastes the 900 GPU-hours already invested (because those hours only produce value when the job completes), while delaying the early-stage job by an hour has a proportionally smaller impact on the total investment's return.

This approach benefits shorter jobs and nearly-complete jobs without excessive penalty to longer ones, and it aligns scheduling decisions with the economic value of completing work rather than the simple accounting of resource consumption. Themis implements this metric through an auction mechanism where jobs bid for resources based on their current marginal value (how much their completion time improves per GPU-hour allocated), creating a market-like dynamic that naturally directs resources to their highest-value use.

8.6.4 Pollux: Adaptive resource allocation

Pollux (Qiao et al. 2021) takes the most aggressive approach of the four research schedulers by jointly optimizing resource allocation and training hyperparameters. Where Tiresias, Gandiva, and Themis make scheduling decisions *about* jobs (when to run, how to share), Pollux makes decisions *within* jobs (how many GPUs and what batch size). The key observation is that the optimal number of GPUs for a training job depends on the current batch size, learning rate, and gradient noise level, all of which change during training as the model moves through different phases of convergence.

Pollux dynamically adjusts each job's GPU allocation and batch size to maximize **cluster-wide goodput**, defined as the rate of useful training progress across all jobs simultaneously. The goodput metric folds both sides of the trade-off into one number: statistical efficiency (how much each gradient step advances convergence, which depends on batch size) and system throughput (how many gradient steps per second, which depends on GPU count and communication overhead). A job experiencing diminishing returns from its current GPU count (because communication overhead is growing superlinearly) may have GPUs reassigned to a job that would benefit more (because it is in a phase of training where larger batches improve statistical efficiency).

This co-optimization of scheduling and hyperparameters achieves 37 to 50 percent improvement in average job completion time compared to scheduling with fixed resource allocations. The improvement comes from two sources: better resource allocation (GPUs go to jobs where they produce the most progress) and better batch size tuning (each job runs at a batch size that balances statistical and computational efficiency for its current training phase). The combined effect is greater than either optimization alone.

8.6.5 Scheduler comparison framework

Custom ML schedulers represent a progressive trade-off between implementation complexity and scheduling precision, so the comparison in table 8.5 matters as a deployment filter rather than a

ranking. General-purpose schedulers see declared resources and coarse runtime metadata; these research systems add ML-specific signals such as iteration structure, elasticity, convergence phase, and measured goodput.

Table 8.5: Custom ML Scheduler Comparison: A taxonomy of research schedulers based on the specific workload characteristic they exploit. Tiresias and Themis optimize for queuing metrics (JCT, fairness) using external signals, while Gandiva and Pollux optimize for efficiency using internal profiling data.

Sched-uler	Key Insight	Scheduling Signal	Optimization Target	Overhead	Production Readiness
Tiresias	Attained service predicts remaining time	GPU-hours consumed	Minimize avg JCT	Low (no profiling)	Medium
Gandiva	Iteration boundaries are preemption points	Iteration timing	Maximize utilization	Medium (profiling)	Medium
Themis	Sunk cost matters for fairness	Completion percentage	Minimize max slowdown	Low (job metadata)	Low (auction complexity)
Pollux	Batch size and allocation are coupled	Goodput (stat + sys)	Maximize cluster goodput	High (continuous profiling)	Low (framework integration)

As table 8.5 summarizes, this design space reveals a fundamental tension: scheduler complexity vs. scheduling quality. Tiresias operates with the least information, essentially guessing that "young" jobs will finish soon, yet achieves significant gains simply by preventing large jobs from blocking small ones. It is robust because it requires no cooperation from the user or the training framework. At the other extreme, Pollux requires deep bidirectional integration: the scheduler must know the job's scaling curve and gradient noise scale, and the job must accept commands to change its batch size and learning rate on the fly. When this integration works, it produces the highest cluster-wide throughput; when the assumptions are violated (for example, a model with unstable convergence dynamics that cannot tolerate batch size changes), the optimization collapses.

Failure modes generally track this complexity gradient. Tiresias fails gracefully: if its assumption (that past usage predicts future usage) is wrong, it simply degrades to a standard fair-share policy. Gandiva's failure mode is performance jitter: if the profiling phase misestimates iteration variance, it may pack incompatible jobs onto the same node, causing interference. Pollux has the most brittle failure mode because it modifies the training semantics themselves; an incorrect goodput model could theoretically harm model convergence, a risk that most production teams are unwilling to take for a 15 percent efficiency gain.

For our 175B parameter model, these trade-offs dictate a hybrid lifecycle. Pollux offers the highest potential improvement during the early, unstable phase of training where batch sizes are small and the job is elastic; it could dynamically resize the job to fit available holes in the cluster. However, effectively using Pollux requires modifying the training loop to be elasticity-aware. Tiresias offers the simplest deployment path: it would treat the 175B model as a "heavy" job and deprioritize it relative to small experiments, ensuring the cluster remains responsive to researchers without requiring any changes to the 175B model's code. In practice, most infrastructure teams start with Tiresias-like policies to solve the "blocked queue" problem before attempting the deep integration required by Pollux.

The consistent theme across all four systems is that exploiting domain-specific knowledge about ML workloads (predictable iteration times, diminishing returns from parallel scaling, and inherent checkpoint-restart capability) enables dramatically better scheduling outcomes than static resource requests alone. A generic scheduler sees a job requesting 64 GPUs for 3 days; an ML-aware scheduler sees a stochastic gradient descent process that converges nonlinearly, releases resources if performance plateaus, and can be preempted with minimal loss. The decision framework for selecting among these schedulers depends on the primary bottleneck: for exploratory clusters with high churn, Gandiva's time-slicing minimizes queuing delay; for production training where job completion time is the primary SLA, Tiresias's age-based prioritization prevents starvation; for large-scale training where resource efficiency is paramount, Pollux's adaptive scaling reclaims the "elasticity gap" that static allocation leaves on the table.

The challenge for production systems is incorporating these insights while maintaining the operational reliability, policy flexibility, and organizational governance that production environments

require. Production platforms usually adopt scheduler ideas selectively because reliability, governance, heterogeneous hardware, and user policy constrain pure research designs.

8.6.6 From research to production

Research schedulers become useful in production only when their core signal survives contact with a heterogeneous data center. In research, the cluster is often assumed to be a homogeneous pool of accelerators with uniform interconnect topology, and jobs are well-behaved entities that declare their resource needs accurately. In production, the "cluster" is a geological formation of hardware generations: V100s alongside A100s and H100s, connected by a patchwork of InfiniBand and Ethernet, running jobs that frequently crash, hang, or misrepresent their memory requirements. Consequently, no major hyperscaler simply "installs" a research scheduler; instead, they selectively transplant specific algorithms into standard orchestrators via Kubernetes operators or Slurm plugins.

Consider our 175B-parameter model training run. A pure implementation of Pollux might aggressively resize the job based on global cluster load, scaling the number of GPUs up and down to maximize goodput. However, changing the worker count for a synchronous training job requires a checkpoint-restart cycle, which for a model of this size involves writing terabytes of optimizer state to persistent storage. If the re-scaling interval is too short, the I/O overhead of checkpointing negates the throughput gains. In practice, production teams adopt a hybrid approach: they use Pollux-style adaptive allocation primarily during the volatile early phase of training or for hyperparameter sweeps, but lock the 175B model into a static, topology-aware allocation once the learning rate warmup completes and the job enters its months-long steady state.

The build-vs.-buy decision for a custom scheduler is a substantial capital investment. Developing a robust, fault-tolerant scheduler that outperforms the default bin-packing behavior of Kubernetes requires a specialized team of 3 to 5 systems engineers working for 6 to 12 months. The ROI math is what justifies the effort: if the team can improve overall cluster utilization by 10 percent on a fleet of 10,000 H100 GPUs (approximately $25,000 per hour in capital depreciation and power), the savings amount to roughly $22 million per year. This stark return on investment drives the development of internal scheduling platforms at every major AI lab, transforming the scheduler from a mere utility into a primary lever for operational efficiency.

The risk is not only implementation cost; scheduler behavior can also fail to match the execution semantics of distributed training.

War Story 8.1: OpenAI's gang-scheduling deadlock (2021)

Context: OpenAI scaled Kubernetes clusters to 7,500 nodes for workloads including GPT-3, CLIP, and DALL-E. Their large training jobs used MPI-style execution, where all workers in a group had to be scheduled simultaneously before training could make progress (Sigler 2021).

Failure mode: Default Kubernetes scheduling did not guarantee that all of one job's workers would be placed before scheduling started filling another job. If two experiments each requested the whole cluster, Kubernetes could schedule half of each job instead of all of one job, leaving both jobs unable to run.

Consequence: The cluster could appear busy while no experiment made useful training progress: a classic gang-scheduling deadlock caused by partial allocation.

Systems lesson: Distributed ML scheduling must encode the job's execution semantics. For MPI-style training, "some pods scheduled" is not partial progress; it is wasted allocation. Production schedulers need gang scheduling, quota controls, and workload-aware policies rather than generic pod placement alone.

A scheduler designed for general-purpose pod placement will mishandle workloads that require gang semantics, and adding gang awareness alone does not resolve priority conflicts among competing training jobs. Each of the four research schedulers examined above addresses one facet of this problem (runtime estimation, iteration-boundary preemption, fairness across finish times, or joint resource-hyperparameter tuning), but none addresses all facets simultaneously. The design space is therefore a set of explicit trade-offs rather than a single optimal policy.

Checkpoint 8.3: Custom scheduler design space

Consider the trade-offs between the four research schedulers:

- ☐ **Estimate-free scheduling**: Identify the information Tiresias sacrifices by eliminating runtime estimates, and describe when that sacrifice hurts scheduling quality.
- ☐ **Time-slicing limits**: Identify workloads for which Gandiva's iteration-boundary time-slicing fails, and explain why.
- ☐ **Job starvation**: Assess whether Themis's preference for nearly complete jobs can starve long-running pretraining jobs indefinitely.
- ☐ **Goodput-model risk**: Describe the failure mode when Pollux's goodput model is wrong.

Custom schedulers maximize throughput for long-running training jobs, but the operational reality changes drastically when models are deployed for inference. The objective function flips from maximizing cluster utilization to guaranteeing millisecond-level responsiveness. Serving resource management must handle bursty, unpredictable traffic patterns while adhering to strict Service Level Objectives.

8.7 Serving Resource Management

Training jobs are predictable, batch-oriented workloads that run for weeks; serving workloads are volatile, user-facing services that must respond in milliseconds to unpredictable traffic spikes. When an e-commerce platform launches a major sale, the serving fleet must instantly autoscale to handle thousands of requests per second. Serving resource management forces the orchestrator to balance these aggressive latency constraints against raw hardware utilization.

For orchestration, the important serving fact is that inference capacity is a live envelope around user traffic rather than a batch allocation around a training job. The fleet orchestrator manages that envelope by scaling replica counts up and down with demand, isolating serving workloads from interference, and balancing inference resource needs against training's claims on the same shared infrastructure.

8.7.1 Autoscaling for inference

Kubernetes **Horizontal Pod Autoscaling (HPA)** adjusts replica counts based on observed metrics, adding instances when load increases and removing them when load decreases. The default autoscaling metric for general cloud workloads is CPU utilization, with targets of 50 to 70 percent. This default poorly reflects GPU inference workloads for two reasons. First, GPU utilization is a noisy metric that can be high even when the system is not processing user requests (background maintenance, model warmup). Second, the relationship between GPU utilization and inference latency is highly nonlinear: latency can spike dramatically when utilization crosses a threshold (typically 70 to 80 percent for GPU inference), making utilization-based scaling reactive rather than predictive. The metric choice is therefore the first scheduling decision for serving: table 8.6 lists custom inference metrics that predict user-facing degradation before it occurs.

Table 8.6: Inference Autoscaling Metrics: Custom metrics for GPU inference workloads capture the relationship between load and user-facing performance more accurately than default CPU utilization. Queue depth provides a leading indicator of latency degradation, while pending tokens captures outstanding autoregressive decode work; KV-cache residency and context-length distribution then determine whether that work is also memory-pressure risk.

Metric	Target Range	Considerations
GPU utilization	60 to 80%	Varies by model batch efficiency
Request queue depth	10 to 50 requests	Prevents latency spikes before they manifest in P99 metrics
P99 latency	Below SLO target	Reactive metric that lags demand changes by seconds to minutes
Pending tokens	Model-specific	Tokens queued or currently being decoded across active requests

Beyond the HPA-oriented metrics in table 8.6, **Vertical Pod Autoscaling (VPA)** operates on a different axis, adjusting resource requests and limits for individual pods. Where HPA changes how many instances run, VPA changes how much resource each instance receives. For inference, VPA can right-size memory allocations based on observed usage patterns, preventing over-provisioning of CPU memory and host resources that reduces the number of inference pods that fit on each node. GPU resources cannot be vertically scaled without pod restart (the CUDA context must be reinitialized), limiting VPA's utility for accelerated workloads where model loading takes minutes. However, VPA is valuable for the CPU components of inference pipelines, such as preprocessing, postprocessing, and tokenization, where resource requirements may differ significantly from initial estimates and can be adjusted with a brief pod restart.

Large language model (LLM) inference requires specialized scaling considerations that neither HPA nor VPA fully address, due to the **key-value (KV) cache**[15] memory growth pattern. A 70B parameter model serving long-context requests may require more than 80 GB of GPU memory for KV cache alone, even when the serving engine stores that cache in paged, noncontiguous blocks. The GPU memory bottleneck for LLM serving is therefore not the model weights (which are fixed) but the KV cache (which grows with request count and context length). Scaling decisions must account for both request rate and context length distribution, not just computational load. A sudden increase in requests with long contexts (for example, a shift from short chatbot queries to document summarization workloads) can exhaust GPU memory even at moderate request rates, requiring scale-out despite low GPU compute utilization. Conversely, a burst of many short-context requests may stress compute without threatening memory limits.

This dual resource dimension (compute and memory) suggests that effective LLM autoscaling should monitor both GPU compute utilization and GPU memory pressure, scaling out when either approaches critical thresholds. Some production systems define a composite scaling metric that combines these dimensions, triggering scale-out when the maximum of normalized compute utilization and normalized memory pressure exceeds a threshold. This composite approach prevents the system from being surprised by either type of resource exhaustion.

Autoscaling for inference must also handle **cold start latency**, which creates a tension between cost efficiency and responsiveness that has no simple resolution. The cold-start problem in serving is quantitatively more severe than in standard web services due to the sheer mass of model weights. A 70B-parameter model in FP16 occupies approximately 140 GB, and all of it must reach GPU memory before the first request can be served, a transfer bound by PCIe or NVLink bandwidth. Even at the per-direction PCIe Gen4 x16 speed of 32 GB/s, the raw transfer takes over 4 seconds, and in practice model initialization, graph compilation, and safety checks extend this to 60–120 seconds. Loading from Non-Volatile Memory Express (NVMe) SSDs at 7 GB/s read bandwidth takes roughly 20 seconds; from network storage, it can exceed 2 minutes.

Aggressive scale-down policies that terminate replicas during brief traffic lulls create painful cold starts when traffic returns minutes later, producing latency spikes that violate SLOs and degrade user experience. The fundamental problem is that inference traffic exhibits burstiness at multiple timescales (seconds, minutes, hours), and short-term lulls do not reliably predict sustained low demand. A five-minute quiet period might be followed by a traffic spike, and the cost of a cold start during that spike (degraded latency for all requests while the model loads) can exceed the cost savings from the few minutes of freed GPU capacity.

Production systems address cold start through three complementary strategies:

- **Minimum replica count**: Keeping at least one warm replica above zero eliminates cold starts for the first burst of traffic but incurs a baseline cost even during truly idle periods.
- **Predictive scaling**: Scaling from historical traffic patterns, such as diurnal cycles, day-of-week patterns, and known events, adds capacity before expected peaks rather than reacting after load arrives.
- **Prewarming**: Loading model replicas into GPU memory on standby GPUs reduces activation from minutes of model loading to seconds of process initialization, but standby replicas consume GPU memory that cannot serve other workloads.

15 **Key-Value (KV) Cache**: Stores precomputed attention key and value tensors to avoid recomputing attention over the entire sequence for each new token. KV cache grows linearly with sequence length and batch size; for a 70B model with 128K context, the cache can exceed the model weights in memory consumption. This growth pattern creates a scheduling failure mode invisible to compute-based autoscalers: GPU memory exhaustion at low compute utilization, causing requests to stall even when the GPU appears idle.

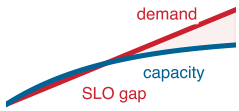
Organizations with multiple models competing for the same GPU pool must balance the pre-warming overhead against the cold-start penalty, choosing which models to keep warm based on their traffic patterns and latency requirements.

8.7.2 Predictive autoscaling and SLO management

Reactive autoscaling (like Kubernetes HPA) is inherently backward-looking: it observes a metric breach (for example, GPU utilization exceeding 80 percent), waits for a stabilization window, and then triggers a scale-up event. For a container that starts in milliseconds, this lag is negligible. For a 175B parameter model that takes 3 minutes to load weights from disk to GPU memory, this lag is fatal to SLOs. If traffic doubles in 30 seconds, a realistic scenario during a product launch or viral event, the reactive scaler will provision new replicas only after the surge has already saturated the existing fleet, causing a 3-minute window of degraded latency and dropped requests.

During this cold-start gap, continuous batching (also called in-flight batching) functions as a shock absorber. Serving engines that implement continuous batching, such as vLLM or TensorRT-LLM, do not wait for a fixed batch to fill before beginning generation; instead, they interleave new incoming requests into ongoing decode steps whenever a sequence slot becomes free. When load spikes and new replicas are still loading, the engine responds by increasing the live batch size toward the KV-cache memory limit, accepting more concurrent sequences per iteration. This absorbs additional requests without rejecting them, sustaining acceptable latency at the cost of higher per-request memory pressure. The relief is bounded: once the KV cache is fully saturated—meaning active sequences occupy every available memory page—the engine must begin queuing or shedding load regardless of batch elasticity. The scheduler therefore has a narrow window, typically from the moment the autoscaler fires until KV-cache saturation, in which continuous batching buys time. Tuning minimum replica counts and scale-up thresholds to keep that window open is the principal lever for preventing SLO violations during traffic ramps that exceed cold-start duration.

Predictive autoscaling decouples scaling actions from current load. By analyzing historical traffic patterns (diurnal cycles, day-of-week seasonality) and incorporating real-time leading indicators (for example, a surge in login requests often precedes a surge in inference queries), the scheduler can preprovision capacity before the demand arrives. Serving our 175B model with a 500 ms P99 SLO requires this anticipation. If the model takes 3 minutes to become ready, the predictive scaler must issue the scale-up command at least 4 minutes before the expected traffic ramp. This transforms the scaling problem from a *control theory problem* (reacting to error) to a *forecasting problem* (predicting the future).



Reactive autoscaling opens an SLO gap during cold start.

Effective scaling policies must be **SLO-driven** rather than utilization-driven. Targeting a fixed utilization (for example, "keep GPU at 70 percent") is a proxy that often fails: a model might hit its latency SLO at 60 percent utilization due to memory bandwidth contention, or it might safely run at 90 percent utilization if the requests are compute-bound and uniform. An SLO-driven policy explicitly targets the metric that matters: "Scale up when P99 latency exceeds 400 ms (80 percent of the 500 ms target)." This approach automatically adapts to changes in workload characteristics. If a new model version is less efficient, the latency metric will rise faster, and the scaler will provision more replicas to maintain the SLO, without requiring manual tuning of utilization thresholds.

To prevent **oscillation**, where the scaler rapidly adds and removes replicas during noisy traffic, production systems implement **hysteresis** and **cooldown periods**. A typical policy scales up aggressively (no stabilization window) to protect the SLO, but scales down conservatively (15-minute cooldown) to avoid thrashing. This asymmetry acknowledges that the cost of an unnecessary scale-up (wasted compute for 15 minutes) is far lower than the cost of a missed scale-up (SLO violation and user churn).

> Napkin Math 8.5: The autoscaling lag
>
> Consider a serving fleet with 10 replicas, each capable of 5 QPS while meeting the 500 ms SLO. Total capacity: 50 QPS.
>
> **Scenario**: Traffic ramps from 40 QPS to 80 QPS linearly over 60 seconds. Model load time: 3 minutes (180 seconds).

Reactive scaling: HPA detects overload at $t = 15$ s (when traffic hits 50 QPS). It requests 10 new replicas. These replicas become ready at $t = 195$ s (15 + 180 seconds). From $t = 15$ s to $t = 195$ s, demand exceeds capacity. During the ramp, excess demand grows linearly from 0 to 30 QPS; after the ramp reaches 80 QPS, the full 30 QPS overload persists until the new replicas are ready. The area above capacity is approximately 4,725 requests, all of which are queued or dropped and violate the SLO.

Predictive scaling: The scaler forecasts the ramp and triggers scale-up at $t = -180$ s. The 10 new replicas come online at $t = 0$ s, pushing capacity to 100 QPS just as the ramp begins. Zero SLO violations.

8.7.3 Resource isolation

When multiple inference workloads share the same physical hardware, noisy neighbor problems arise when one workload's resource consumption degrades another's performance. On GPUs, interference manifests through four shared resource channels:

- **Memory bandwidth**: One workload's data movement can saturate the HBM interface.
- **L2 cache contention**: One workload's working set can evict another's cached data.
- **PCIe bottlenecks**: Concurrent host-device transfers can compete for bus bandwidth.
- **Thermal effects**: One workload's heat generation can cause thermal throttling that affects all workloads on the same GPU.

For latency-sensitive inference, even minor interference can push P99 latency above SLO thresholds, turning a seemingly well-provisioned system into an SLO-violating one.

MIG sits at the strong-isolation end of this design space. Table 8.2 already showed the fixed profiles; the operational implication is that hardware isolation removes most cross-tenant interference but turns partition choice into a placement constraint. MIG is available only on A100 and later GPUs, profiles cannot change without draining all workloads from the GPU, and fixed partition sizes may not match workload requirements. A workload that needs 15 GB of GPU memory wastes 5 GB on a 20 GB MIG instance or cannot fit on a 10 GB instance.

Software approaches to isolation provide more flexibility at the cost of weaker guarantees. Efficiently sharing GPUs for inference requires navigating a hierarchy of isolation mechanisms, each trading performance for security. At the simplest level, **CUDA time-slicing** allows multiple processes to share a GPU by context-switching the compute resources. While flexible, this incurs high latency penalties (often 10–20 ms) and offers no memory isolation. NVIDIA's **Multi-Process Service (MPS)**[16] improves this by allowing kernels from different processes to run concurrently on the same GPU, improving throughput for small batch inference while providing weaker isolation than MIG (Volta and newer give each client a separate GPU address space, but compute and memory bandwidth remain shared).

GPU memory isolation prevents one model from consuming memory needed by another through explicit memory limits enforced by the container runtime. Without such limits, a memory leak, an unexpectedly large batch (triggered by a request with unusually long context), or a misbehaving custom kernel can consume all available GPU memory and crash colocated workloads. MPS improves utilization for small workloads by eliminating the context-switching overhead of time-slicing, but it adds latency overhead of approximately 5 to 10 microseconds per kernel launch and provides limited protection against memory bandwidth interference.

Furthermore, long-running serving instances suffer from dynamic memory fragmentation. The key culprit is the KV cache in transformer inference, which grows and shrinks with request sequence length. Standard allocators struggle with these highly variable lifetimes, leaving holes in GPU memory too small for new requests but collectively wasting gigabytes. Serving engines increasingly use paged memory management to allocate KV cache in noncontiguous blocks, reducing fragmentation-induced out-of-memory failures; Chapter 10 develops the paged-allocation mechanism in full.

At the CPU level, core pinning assigns specific CPU cores to inference pods, preventing the Linux scheduler from migrating processes between cores in ways that invalidate processor caches. For latency-sensitive workloads, isolating cores using `isolcpus` kernel parameters and `taskset`

16 **CUDA MPS (Multi-Process Service)**: Enables concurrent kernel execution from multiple processes on a single GPU, eliminating the 10 to 20 ms context-switch penalty of time-slicing. MPS adds approximately 5 to 10 microseconds of overhead per kernel launch but provides no memory isolation: a process that exhausts GPU memory crashes all colocated workloads. This makes MPS suitable for trusted, same-team inference workloads but dangerous for multi-tenant environments where a single misbehaving model can take down its neighbors.

affinity removes OS scheduling jitter that manifests as random latency spikes. Combined with NUMA-aware[17] placement that ensures inference pods access memory through the nearest memory controller (avoiding remote NUMA access that adds 50 to 100 ns per access), this reduces P99 latency by 10 to 30 percent for sub-millisecond inference tasks. The principle is straightforward: inference latency is dominated by memory access patterns, and any source of memory access unpredictability, whether cache eviction from core migration, remote NUMA access, or TLB misses from address space switching, directly degrades tail latency.

The isolation techniques discussed here represent a spectrum from strong (MIG, hardware isolation) to flexible (MPS, software sharing), and the choice depends on the trust boundary between colocated workloads. Multi-tenant inference platforms serving different external customers typically require MIG for security isolation. Single-tenant platforms serving different internal models on the same GPU can use MPS or time-slicing where the isolation requirements are weaker.

8.7.4 GPU sharing economics

The decision to co-locate multiple models on a single GPU defines the efficiency frontier of an inference fleet. This choice operates on a spectrum: at one end, **exclusive access** guarantees isolation but strands capacity; at the other, **aggressive sharing** (via MPS or time-slicing) maximizes utilization but risks latency interference. The physics of this trade-off are dictated by memory fragmentation and compute contention.

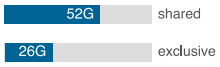
Consider an 80 GB A100 GPU serving a 7B parameter model. The model weights in FP16 consume approximately 14 GB. With a typical KV cache and activation overhead, the total runtime footprint is roughly 26 GB. Under an exclusive access policy, this single model leaves 54 GB (67 percent) of the GPU's high-bandwidth memory dark, a waste of silicon capital. Partitioning the GPU with MIG (for example, a `3g.40gb` profile) tightens the container, reducing the waste to roughly 35 percent within the partition, but still leaves the remainder of the GPU strictly segmented. Enabling MPS to pack two such models onto the same 80 GB device consumes 52 GB, dropping the aggregate memory waste to just 35 percent. For fleets running hundreds of small models, shifting from exclusive to shared hosting can reduce the required GPU count by a factor of 2–3×.

This density comes at the cost of **interference**. When two workloads share a GPU via MPS, they compete for Streaming Multiprocessors (SMs) and memory bandwidth. The degradation is workload-dependent: two compute-bound models (for example, large batch processing) fighting for ALUs often see 15 to 20 percent throughput degradation each. However, a compute-bound model co-located with a memory-bound model (for example, decoding-heavy generation) often coexist peacefully, seeing only 5 percent degradation because they bottleneck on different hardware resources. The scheduler's job thus becomes a multi-dimensional bin packing problem: finding complementary workloads that maximize density while respecting latency SLOs.

For a heterogeneous fleet, this logic dictates a bifurcated strategy. Our 175B parameter model requires about 350 GB just for FP16 weights; on 80 GB A100s that means at least five GPUs before KV cache, activation buffers, runtime workspace, and tensor-parallel granularity are included. In practice, an 8-GPU tensor-parallel replica is the cleaner scheduling unit and is a candidate for exclusive access. Conversely, the dozens of 1B to 7B parameter support models, such as toxicity classifiers, query embedding encoders, and auxiliary generation models, are prime candidates for sharing. By packing these smaller models onto shared "utility nodes," the orchestrator liberates high-performance clusters for the heavy lifting of foundation model inference.

> **Napkin Math 8.6: GPU sharing ROI**
>
> Consider a fleet serving 100 distinct 7B models (26 GB footprint each) and 10 distinct 175B models (350 GB footprint each).
>
> **Strategy A – exclusive access (one model per GPU)**: The 7B models require 100 A100 GPUs (80 GB each), achieving only 32.5 percent memory utilization (100 × 26 GB used out of 8000 GB total). The 175B models require 80 GPUs (10 models × 8 GPUs each for tensor parallelism). Total fleet: 180 GPUs.

[17] **NUMA (Non-Uniform Memory Access)**: Local memory access takes approximately 100 ns vs. 150 to 200 ns for remote-socket access, a 50 to 100 percent penalty. For ML inference preprocessing, placing CPU cores on the wrong NUMA domain relative to the GPU's PCIe connection adds this penalty to every host-device transfer, inflating P99 latency by 10 to 30 percent for sub-millisecond inference tasks. The fix is straightforward (CPU pinning and NUMA-aware pod scheduling) but often overlooked in Kubernetes deployments where the default scheduler ignores NUMA topology entirely.



Sharing packs two tenants, cutting wasted memory from 67 percent to 35 percent.

Strategy B – mixed sharing (MPS for small, exclusive for large): Pack 2 models per A100 for the 7B models (2×26 GB = 52 GB, well within 80 GB), requiring only 50 GPUs at 65 percent memory utilization. The 175B models remain unchanged at 80 GPUs. Total fleet: 130 GPUs. Sharing reduces the fleet size by 27.8 percent (50 fewer GPUs). At $2/GPU-hour, this saves approximately $876,000 annually in compute costs, purely through scheduling policy.

8.7.5 Model routing and traffic management

Once resources are isolated, the orchestrator faces the challenge of directing inference queries to the correct model replicas. For a monolithic web service, simple round-robin load balancing suffices. For a 175B parameter language model served across thousands of GPUs, routing becomes a complex distributed systems problem where the "unit of serving" is no longer a single container but a **tensor parallel group**, a collection of 8 GPUs that must function as a single logical entity. The load balancer cannot simply route a request to "GPU 12"; it must route to "TP Group 3," ensuring the request arrives at the exact moment the group is ready to process it.

Request routing strategies determine cluster-wide throughput. Naive **round-robin** distribution fails for generative workloads because request processing times vary by orders of magnitude based on output token length. A round-robin scheduler inevitably sends new requests to replicas backed up with long-generation tasks, causing tail latency to spike. **Least-outstanding-requests** (LOR) routing improves this by directing traffic to the idlest replicas. However, for large language models, **memory-aware routing** is superior. By tracking the KV cache occupancy of each replica, the router can send long-context requests to replicas with ample free memory and short requests to those near capacity, preventing fragmentation-induced out-of-memory errors and increasing effective batch sizes by 30 to 40 percent compared to blind routing.

Traffic shifting mechanisms like canary deployments and A/B testing are essential for safe model updates. When deploying Model v2, the orchestrator does not simply replace Model v1. Instead, it spins up v2 replicas alongside v1, creating a shadow fleet. The traffic manager routes 1 percent of live requests to v2 (often in "shadow mode," where the response is computed but discarded) to validate latency and error rates against v1. Only after statistical verification does the orchestrator gradually shift weights (5 percent, 25 percent, 100 percent), draining v1 replicas only as v2 proves stable.

Multi-model serving complicates this further. A single cluster often hosts diverse models: a product recommendation model (10 ms P99 SLO, high throughput) alongside a coding assistant model (2,000 ms P99 SLO, bursty usage). Placing these on the same node invites interference; placing them on separate clusters strands capacity. The optimal strategy uses **latency-aware bin packing**: the orchestrator co-locates latency-critical models with batch-processing jobs (like embedding generation) that can be throttled instantly, ensuring high utilization without violating strict SLOs.

8.7.6 The training-serving resource boundary

The deepest utilization divide in the ML fleet is the wall between training and serving. Most organizations manage these as separate fiefdoms: a "Production Serving" cluster that must handle peak traffic (and sits 50 percent idle at night) and a "Research Training" cluster that is perpetually backlogged. This **static partitioning** is safe but economically inefficient. **Dynamic sharing** unifies these pools, but it introduces a fundamental friction: the mode-switching cost.

Repurposing a GPU from training to serving is not instantaneous. The orchestrator must checkpoint the training job (1 to 5 minutes), kill the container, launch the inference server, load the model weights from storage (2 to 10 minutes), and run warmup queries (1 to 5 minutes) to populate compilation caches. This 5-to-20-minute switching latency means GPUs cannot react to second-by-second traffic bursts. Instead, they must follow diurnal patterns.

Inference traffic typically follows a "human curve": low at 3 AM, ramping up at 8 AM, peaking at 2 PM, and tapering off at 10 PM. A smart orchestrator forecasts this curve 30 minutes in advance. At 7:30 AM, it preempts low-priority training jobs (like hyperparameter sweeps) to warm up inference replicas. At 10:30 PM, as traffic drops, it drains inference replicas and releases GPUs back to the training scheduler.

Napkin Math 8.7: The diurnal GPU shift

Consider a fleet of 1,024 GPUs managed under two regimes:

Scenario A – static partitioning: The serving pool reserves 400 GPUs to handle peak demand, but average usage is only 150 GPUs (37.5 percent utilization). The training pool reserves 618 GPUs, and backlog ensures 100 percent utilization. Result: 150 GPUs + 618 GPUs = 768 active GPUs. Fleet utilization: 75 percent.

Scenario B – dynamic sharing: Serving claims GPUs strictly as needed (average 150 GPUs). Training reclaims unused serving GPUs (average 250 GPUs), minus a 117-GPU safety buffer. The orchestrator shifts approximately 133 GPUs between roles twice daily based on diurnal forecasts. Result: serving (150 GPUs) + training (618 GPUs + 133 GPUs) = 901 active GPUs (the second training term is reclaimed serving capacity). Fleet utilization: 88 percent.

Dynamic sharing recovers 133 GPUs of effective capacity. At $2/GPU-hour, this creates approximately $2.3M per year in value without purchasing a single additional accelerator.

Optimizing isolation for a single deployment is straightforward, but maintaining strict guarantees becomes volatile when thousands of training and serving workloads coexist on the same infrastructure. The collision of high-priority inference SLAs with resource-hungry training jobs creates a noisy neighbor problem that simple containerization cannot solve. Multi-tenancy and quotas become the governance layer that arbitrates fair access and prevents tragedy-of-the-commons scenarios in shared fleets.

8.8 Multi-Tenancy and Quotas

When the computer vision team hoards 500 GPUs for a week while the natural language processing (NLP) team sits blocked trying to push a critical bug fix to production, shared clusters devolve into a tragedy of the commons. Teams with the most aggressive job submission rates, the largest jobs, or the most persistent resubmission scripts consume disproportionate resources, while teams with more modest or intermittent needs find the cluster perpetually occupied. This creates organizational friction, political escalation to management, and underinvestment in slower-moving but potentially higher-value projects whose teams lack the engineering effort to compete for resources.

Multi-tenancy and quotas provide the organizational and technical firewalls needed to prevent this degradation. The quota systems discussed in this section establish formal policies for resource allocation, borrowing, and reclamation, so fair access and high overall utilization become enforceable scheduler properties rather than managerial negotiation.

8.8.1 Hierarchical fair-share

GPU quota allocation typically operates at the namespace or project level, defining how much of the cluster each team is entitled to use. The simplest approach allocates fixed GPU counts per team: Team A gets 500 GPUs, Team B gets 300 GPUs, Team C gets 200 GPUs. This approach is easy to understand and implement but leads to systematic underutilization when teams have variable workloads. If Team A's 500-GPU allocation sits 50 percent idle during a quiet period while Team B's queue overflows with urgent work, the cluster wastes 250 GPUs of capacity that rigid quotas forbid Team B from using. The alternative, allocating less quota to each team, creates a different problem: teams are blocked from running legitimate work during peak periods even when the cluster has overall spare capacity.

Hierarchical quotas resolve this tension by enabling departmental limits with sub-team flexibility and cross-team borrowing. Equation 8.4 bounds the effective quota for any team by both its own allocation and the parent department's capacity that sibling teams have not already claimed, where $U_{\text{allocated}}$ is the resources currently committed to each of the other teams in the department:

$$Q_{\text{effective}} = \min\left(Q_{\text{team}},\ Q_{\text{department}} - \sum_{\text{other teams}} U_{\text{allocated}}\right) \tag{8.4}$$

When aggregate demand exceeds capacity, each team receives resources proportional to its share allocation, ensuring that no team is systematically disadvantaged. Unused capacity borrows down

the hierarchy: if Team A uses only 60 percent of its 500-GPU allocation, the remaining 200 GPUs become available to other teams within the same department. These borrowed resources carry lower preemption priority than owned resources: when Team A later submits jobs that need its full allocation, the scheduler reclaims borrowed capacity by preempting jobs running on those resources, returning capacity to its rightful owner.

This borrowing mechanism is essential for achieving high utilization in multi-tenant clusters. Without borrowing, organizations face a painful choice between allocating enough quota to handle each team's peak demand (guaranteeing low utilization when demand is uneven, since peaks rarely coincide across all teams) or allocating less (guaranteeing that some team is always quota-blocked during their peak periods). Borrowing resolves this dilemma by allowing peak-level allocations as *guarantees* while permitting idle capacity to *flow* to where it is needed. A team that is guaranteed 500 GPUs can always get 500 GPUs when needed (through preemption of borrowed jobs), but it does not waste the cluster's capacity when its actual demand is lower.

The preemption dynamics of borrowing require careful integration with the checkpoint infrastructure from Chapter 7. Jobs running on borrowed capacity must checkpoint frequently enough that preemption (when the owning team reclaims resources) does not waste significant work. Shared clusters often distinguish between "guaranteed" and "opportunistic" scheduling tiers: guaranteed jobs run on the team's owned quota and face preemption only from higher-priority work, while opportunistic jobs run on borrowed capacity and accept that they may be preempted when the owning team needs its resources back. Training frameworks that support elastic scaling can respond to borrowed-capacity reclamation by shrinking rather than terminating, further reducing the cost of the borrowing mechanism.

8.8.2 Burst capacity and over-subscription

In a shared GPU cluster, static quotas inevitably lead to the "burst capacity" paradox: Team A has a quota of 64 GPUs but needs 256 for a weekend-long training run, while Team B's allocated 64 GPUs sit idle. The solution requires decoupling "guaranteed quota" from "limit quota." A team is guaranteed 64 GPUs, which they can access instantly, but can burst up to 512 GPUs if the cluster has slack capacity. This "over-subscription" model relies on preemption: if Team B wakes up and reclaims their guaranteed share, the scheduler must immediately terminate Team A's burst jobs.

Burst capacity handling enables teams to temporarily exceed their quotas when cluster-wide resources are available, providing a more aggressive sharing mechanism than quota borrowing. Where borrowing reassigns idle capacity from one team to another, burst capacity allows teams to exceed the total allocated capacity by exploiting the gap between *requested* resources and *actual* utilization. Over-commitment ratios such as 1.2–1.5$\times$ are plausible when admission controllers track actual vs. requested resources and intervene when contention materializes. When contention occurs, jobs using burst capacity face preemption first, protecting guaranteed allocations within each team's owned quota.

To manage this safely, clusters must implement strict priority classes:

- **Production serving**: Priority 0 workloads are nonpreemptible.
- **Production training**: Priority 1 workloads handle core model updates and are only preemptible by serving outages.
- **Interactive/debug**: Priority 2 workloads get high scheduling precedence but short time limits.
- **Batch/research**: Priority 3 workloads are fully preemptible and fault-tolerant.

This hierarchy ensures that a massive hyperparameter sweep fills the cluster's cracks but evaporates instantly when a critical retraining job arrives.

The appropriate over-commitment ratio depends on workload characteristics and requires careful observation of actual resource utilization patterns. If most jobs request 100 percent GPU utilization but actually achieve 70 percent (due to data loading phases, communication overhead, memory allocation gaps, or suboptimal kernel scheduling), a 1.3$\times$ over-commitment ratio improves cluster-wide utilization without significant contention because the total actual demand (70 percent $\times$ 1.3 = 91 percent) remains below physical capacity. If jobs are genuinely compute-bound and sustain near-100 percent GPU utilization, over-commitment leads to resource contention and performance degradation for all colocated workloads. The over-commitment ratio should therefore be calibrated

empirically based on cluster-wide utilization telemetry, not set based on theoretical assumptions about workload behavior.

8.8.3 Resource accounting and observability

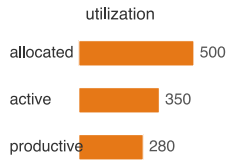
The 64 GPUs that sat idle for a weekend because a researcher forgot to cancel a reservation represent invisible waste in a multi-tenant cloud fleet. In a dedicated cluster, such waste is at least visible: the lights are on, but the fans are quiet. In a shared environment, the cost is hidden and compounds across thousands of jobs. Effective governance requires moving beyond simple "allocation" metrics to a tiered accounting model that distinguishes between what was requested, what was used, and what was useful.

Three tiers of measurement expose this gap:

- **Allocated capacity**: The resources the scheduler has reserved for a job are unavailable to others.
- **Compute utilization**: The percentage of time GPU kernels are active measures hardware busyness.
- **Productive utilization**: The fraction of time the GPU advances model state excludes data loading pauses, communication overhead, and checkpointing.

The distinction is financial, not merely technical. A team might be "allocated" 500 GPUs but only using 350 (70 percent compute utilization) and only "productively using" 280 (56 percent of allocation). If the organization pays for allocation but measures success by training progress, this 44 percent gap represents pure burn.

Attribution in a shared fleet presents a forensic challenge. A single "training" job might be a shared experiment between three teams, or a platform test run by an SRE. Without granular tagging at the job level, costs default to the "infrastructure" bucket, creating a tragedy of the commons where no one owns the bill. Kubernetes labels and Slurm accounts provide the mechanism for attribution, but the organizational discipline to apply them consistently is the harder problem. Mature organizations enforce "no tag, no schedule" policies, rejecting untagged jobs at the admission controller level.



Allocated, active, and productive utilization drift far apart.

This data feeds the utilization dashboard, the cluster operator's primary instrument for fleet health. This view must synthesize per-team utilization against quotas, queue depth by priority class, and the **fragmentation index**, a measure of free GPUs that cannot be allocated due to topology constraints. Crucially, it must track the "cost of chaos": preemption rates, spot instance interruption frequencies, and the recovery overhead they induce. When these metrics are visible, they drive behavior. A 64-GPU fine-tuning run for our 175B model held for 816 h consumes about $130,560 of capacity (64 GPUs $\times$ 816 h $\times$ $2.50/GPU-hour). Without proper accounting, this cost is invisible to the team that requested it. With chargeback, the team must justify this expenditure against the model's expected business value.

Automated anomaly detection turns these metrics into actionable signals. A sudden spike in a team's consumption to 3$\times$ their historical norm might indicate a runaway script spawning infinite jobs. A queue that grows faster than it drains signals a capacity shortfall that auto-scaling must address. Conversely, a sudden drop in cluster-wide power consumption typically precedes a mass job failure event, often due to a shared dependency like a storage outage. These alerts allow operators to intervene before the budget is drained or the queue becomes unmanageable.

> Systems Perspective 8.7: The three utilization metrics
>
> To debug cluster efficiency, measure utilization at three distinct layers. **Allocated utilization** is the percentage of physical GPUs reserved by the scheduler; low values indicate a lack of demand or overly restrictive quotas. **Compute utilization** is the percentage of time allocated GPUs are executing kernels (reported by `nvidia-smi`); low values indicate inefficient code, I/O bottlenecks, or communication stalls. **Productive utilization** is the percentage of allocated time spent effectively training (excluding overhead); low values indicate poor distributed scaling, excessive checkpointing, or frequent restarts.

8.8.4 Security and namespace isolation

Multi-tenancy requires not only fair resource allocation but also security isolation that prevents teams from interfering with or observing each other's workloads. The stakes are real: ML models represent significant intellectual property, training data may contain sensitive information subject to privacy regulations, and the models themselves may encode proprietary business logic. Without proper isolation, a compromised or misconfigured workload in one team's namespace could access another team's model weights, training data, or inference traffic.

In Kubernetes environments, namespace separation provides the fundamental isolation boundary. Each team operates within dedicated namespaces with role-based access control (RBAC) limiting visibility to their own resources. RBAC policies control who can submit jobs, view logs, access model artifacts, and modify scheduling policies within each namespace, providing organizational governance over cluster usage.

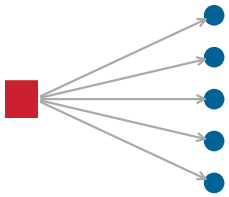
Network policies extend isolation to the network layer, preventing cross-namespace communication except through explicitly permitted services. Network policies for ML workloads must balance isolation against the communication requirements of distributed training. A practical policy might allow unrestricted all-to-all communication within a namespace (necessary for ring AllReduce as analyzed in Chapter 6, where every worker must communicate with every other worker) while blocking all ingress from other namespaces (preventing external workloads from intercepting gradient traffic or model updates). Egress policies can prevent training jobs from accessing external networks, reducing data exfiltration risk from compromised training code or poisoned dependencies.

Table 8.7 compares GPU virtualization options across isolation strength, resource efficiency, and workload fit.

Table 8.7: GPU Virtualization Isolation Spectrum: GPU sharing mechanisms trade isolation against packing efficiency, so the right mechanism depends on both the trust boundary and the workload's tolerance for interference.

Option	Isolation strength	Efficiency and flexibility	Suitable workload or trust boundary
Time-slicing	Low; workloads share a GPU through software scheduling.	High flexibility and packing efficiency.	Trusted workloads from the same team sharing a GPU for cost efficiency.
MIG	Strong hardware partitioning with fixed GPU slices.	Predictable isolation but less flexible placement.	Multi-tenant inference where different customers share the same physical GPU.
Full device passthrough	Complete isolation through exclusive access to one or more GPUs.	Lowest packing efficiency.	Training jobs that saturate resources or cannot tolerate interference from colocated work.

The choice depends on the trust boundary between colocated workloads and the performance sensitivity of each workload type. Security in multi-tenant environments extends beyond simple resource fairness. Side-channel attacks on shared GPUs are a documented vulnerability; by monitoring contention on shared caches or memory controllers, a malicious tenant can infer the architecture or even data properties of a co-resident model. In highly sensitive environments, hardware isolation mechanisms like MIG or strictly dedicating entire GPU nodes to single tenants become mandatory requirements.



One eviction ripples to many: the preemption cascade.

8.8.5 Priority preemption cascades

When a high-priority job enters a saturated cluster, the scheduler must decide which running workloads to terminate to free up resources. This decision is rarely isolated. In tightly packed clusters, evicting a medium-priority job to accommodate a high-priority request often triggers a **preemption cascade**, where the evicted job immediately attempts to reschedule itself by displacing lower-priority workloads. Without dampening controls, a single urgent inference service deployment can ripple through the queue, destabilizing dozens of training jobs and forcing a storm of checkpoint reloads that saturate storage bandwidth.

The engineering cost of preemption extends far beyond the scheduling latency. The **preemption tax** is the sum of lost computation since the last checkpoint, the overhead of persisting state, and

the cold-start penalty upon resumption. For large-scale distributed training, this tax is nonlinear. Consider the 64-GPU fine-tuning run for our 175B parameter model. If preempted, the job loses the work performed since the last snapshot (average 15 minutes). Upon restarting, it requires 20 minutes to reload the massive optimizer state from distributed storage and another 10 minutes of warmup time as data pipelines refill prefetch buffers, just-in-time (JIT) compilers re-optimize kernels, and GPU caches re-populate.

A further hidden cost compounds this tax: reconstructing the state of the distributed data pipeline. Preempting a large-scale training job does not merely interrupt gradient computation—it discards the deterministic state of the data loader, including the random seed used for shuffling, the shuffle order applied to the epoch, and the byte offset into the dataset shards that records exactly which samples have been consumed. Without restoring this state precisely, the restarted job either re-reads samples already processed (poisoning gradient statistics by amplifying certain examples) or uses a different shuffle order (disrupting the statistical independence between epochs). Restoring a petabyte-scale distributed data pipeline to its pre-preemption state typically requires replaying metadata logs across all data loader workers, a process that can extend the effective warmup window well beyond the time consumed by optimizer state reloading alone. Frameworks that store a compact data loader checkpoint—recording the seed, the sampled permutation index, and the current shard offset—reduce this cost significantly, but the majority of production training pipelines require a manual warmup period to approximate the original data ordering before gradient quality returns to baseline.

During this 45-minute recovery window, the GPUs are active but effectively unproductive. At $2/GPU-hour, a single preemption event costs $96 in wasted compute capital. If the scheduler allows unlimited preemption, 12 preemptions per day waste 9 hours per day of that job's allocation, or 576 GPU-hours per day for this 64-GPU run. To mitigate this, schedulers enforce **preemption budgets**. These rate limits constrain disruption frequency, such as preventing a job from being preempted more than once per hour or capping total cluster preemption churn to 5 percent of capacity in any 10-minute window. This forces the scheduler to wait for natural job completions rather than triggering a cascade, trading slightly higher pending times for significantly higher aggregate throughput.

8.8.6 Quota governance and organizational dynamics

Schedulers manage the second-by-second allocation of silicon, but **quota governance** manages the month-by-month allocation of organizational intent. Quotas are the interface between engineering constraints and business priorities. In mature ML organizations, these are rarely static; they are adjusted through periodic **quota review cycles** where allocations are recalibrated based on realized utilization rather than forecasted demand. A team that consistently uses only 40 percent of their 128-GPU allocation is effectively blocking other teams from launching experiments, creating a phantom scarcity that drives up queue times despite ample physical capacity.

The central tension in quota management is the **hoarding problem**. Engineering teams, rational in their desire to minimize wait times, often request maximum capacity "just in case" a project accelerates. This leads to low average utilization and high burst demand. Two ML-specific patterns amplify this behavior beyond generic resource anxiety. The first is the reinforcement learning from human feedback (RLHF) evaluation phase, in which the GPU nodes allocated to the policy model sit largely idle while the team awaits scores from human annotators. The team retains the reservation to avoid queuing again once annotation batches arrive, but the actual GPU occupancy during the annotation window can fall below 10 percent. The second pattern occurs when a team reserves a GPU block for a training run whose preprocessing pipeline has not yet completed. CPU-heavy data tokenization and vocabulary normalization for large corpora routinely takes longer than the team estimated; the GPUs sit reserved and idle while the tokenizer processes the dataset, because releasing the reservation risks losing it to a competing team. Both scenarios are rational from the team's perspective but represent systematic waste at the cluster level. To counter this, platform teams implement **utilization-based reclamation**: if a quota pool remains underutilized for a set period, the system automatically reduces the allocation, returning resources to the general pool. This technical enforcement shifts the burden of proof back to the team to justify their reservation.

Financial accountability further aligns incentives. Organizations typically begin with showback, a reporting mechanism that exposes the dollar cost of GPU usage to engineering managers without

directly impacting their budgets. This fosters awareness but lacks teeth. As spending scales, organizations transition to chargeback, where compute costs are deducted from team budgets. Chargeback creates immediate pressure to optimize code and release unused quota, though it risks discouraging speculative research if the internal pricing model is too aggressive.

A concrete quota scenario shows how quickly idle reservations become expensive.

Example 8.3: Quota hoarding

Scenario: A team reserves a block of 500 GPUs for a "critical" urgent model refresh.

Failure mode: Upstream data delays postpone the training jobs, but the team retains the reservation to ensure availability "when the data arrives."

Consequence: The GPUs sit idle for three weeks, a capital waste of roughly $504,000, while other teams' queues grow.

Systems insight: The infrastructure team implemented a "use it or lose it" policy. Any reserved quota group operating below 20 percent utilization for 72 consecutive hours is automatically reclaimed and converted to spot capacity for the general pool. Reclaiming the quota requires VP-level approval, effectively eliminating "parking" on idle hardware.

The "use it or lose it" reclamation policy addresses the most visible symptom of quota hoarding, but it raises a second-order question: when idle capacity is lent to another team, the system must distinguish borrowed capacity (preemptible on short notice) from guaranteed reservations (protected by SLOs). Defining borrowing, preemption, and guaranteed tiers as first-class policy primitives is the core multi-tenancy design decision.

Checkpoint 8.4: Multi-tenancy design decisions

Consider a 2,000-GPU cluster shared between a research team (60 percent allocation) and a production team (40 percent allocation):

- ☐ **Idle reclamation**: Decide whether the production team can use the idle 30 percent of the cluster, and describe the preemption behavior when the research team submits a large job.
- ☐ **Priority classes**: Configure priority classes for a production inference workload that needs 100 GPUs with guaranteed latency SLOs and a research training job that can tolerate preemption.
- ☐ **Over-commitment ratio**: Determine the appropriate over-commitment ratio when research jobs average 65 percent GPU utilization and production jobs average 85 percent.

Even with sophisticated multi-tenancy and quota policies in place, the cluster can still fall victim to complex, systemic bottlenecks that defy simple monitoring. When utilization drops inexplicably despite full queues, operators must move from dashboards to systematic utilization debugging.

8.9 Debugging Cluster Utilization

A cluster dashboard shows average GPU utilization stuck at 60 percent, yet developers complain about three-day queue times for small debugging jobs. Hundreds of GPUs sit idle, but no jobs are starting. Bridging this gap between theoretical capacity and production reality requires methodical forensic analysis to untangle the interactions between hardware topology, scheduler policies, and user behavior. As figure 8.6 illustrates, the relationship between utilization and wait time is highly nonlinear: operating above 80 percent causes wait times to explode.

The debugging method is to compare scheduler state with physical capability, inspect job requests against the actual hardware pools, and then test for a policy-infrastructure mismatch. A full queue and idle accelerators can coexist when the scheduler is honoring constraints that the dashboard has

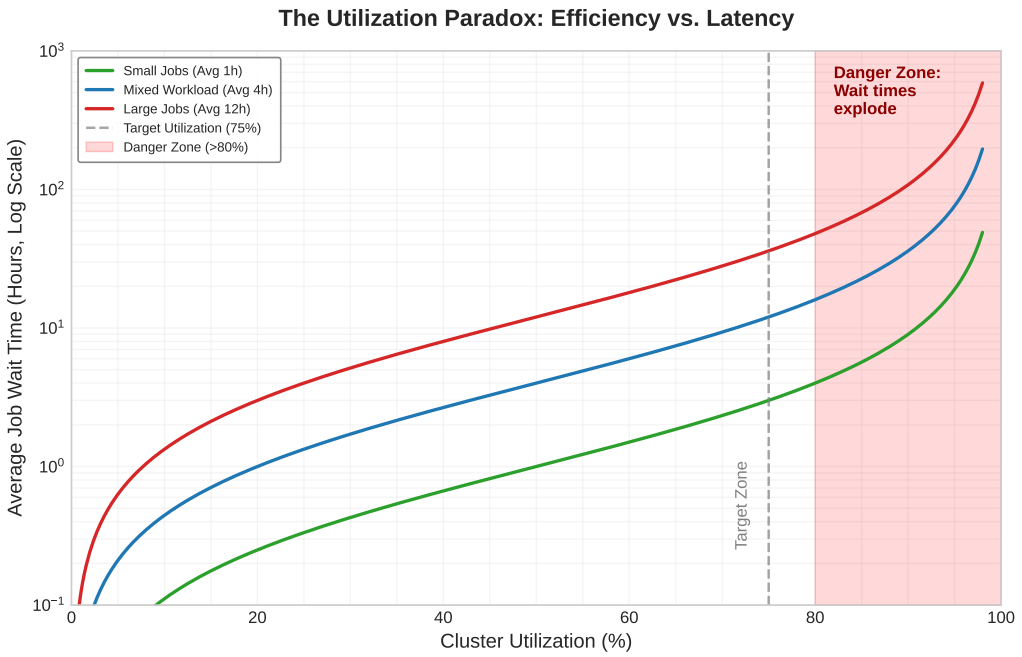


Figure 8.6: The Utilization Paradox: As cluster utilization approaches saturation (100 percent), job wait times increase hyperbolically and diverge near saturation, not linearly. Operating above 80 percent utilization (the Danger Zone) causes unpredictable scheduling delays, especially for large jobs, illustrating why 100 percent utilization is an anti-pattern for interactive research clusters. A dashed line shows target utilization at 75 percent.

aggregated away. The cluster at the center of the worked example below runs the heterogeneous hardware in table 8.8, accumulated across three procurement cycles.

Table 8.8: Cluster hardware inventory: Three procurement generations create distinct resource pools with different capabilities. A100 nodes support tensor parallelism via NVLink, while V100 nodes are limited to data parallelism.

GPU Type	Count	Memory	Interconnect	Nodes
A100-80 GB	400 GPUs	80 GB HBM2e	NVLink (600 GB/s)	50 nodes × 8 GPUs
A100-40 GB	424 GPUs	40 GB HBM2e	NVLink (600 GB/s)	53 nodes × 8 GPUs
V100-32 GB	200 GPUs	32 GB HBM2	PCIe Gen3 (15.8 GB/s)	50 nodes × 4 GPUs

Example 8.4: Debugging low GPU utilization

Problem: A 1,024-GPU cluster reports 60 percent average utilization, below the 80 percent operating target, despite a full queue with over 50 pending jobs. Dashboards show idle GPUs, but users still wait. Identify the binding constraint and the diagnostic path.

The fleet stack framework in figure 1.13 structures the diagnosis: inspect hardware capability first, then scheduling policy, then the mismatch between them.

The first diagnostic step is infrastructure-layer analysis. The hardware inventory in table 8.8 separates physical capability before the scheduler policy is examined: A100 nodes can run tensor-parallel jobs over NVLink, while the V100 pool is confined to data parallelism over PCIe.

The second diagnostic step is distribution-layer analysis. The Slurm scheduler implements gang scheduling with strict resource type matching. Examining the job specifications reveals the demand pattern:

- 15 large training jobs requesting at least 64 GPUs of A100-80 GB capacity, with aggregate demand of 1,200 GPUs.
- 8 medium jobs requesting 32 GPUs of A100-40 GB capacity, with aggregate demand of 256 GPUs.
- 0 jobs explicitly request V100 resources.

The scheduler's allocation log shows A100-80 GB GPUs nearly saturated by a few running jobs, while the remaining large jobs stay pending because no full 64 GPUs A100-80 GB allocation can be formed. The result is not a standard Slurm partial-allocation deadlock; it is a policy-induced utilization gap where strict resource requests leave A100-40 GB and V100 pools idle while A100-80 GB jobs wait.

Analysis: Multiple pathologies compound to create low utilization:

1. **Over-specification**: Model memory analysis reveals that most large jobs actually require only 35 GB peak memory per GPU, well within A100-40 GB capacity. Users copied job templates specifying A100-80 GB without recalculating requirements.
2. **Pool fragmentation**: The strict homogeneity requirement means a 64 GPUs job requesting "A100-80 GB" cannot use any A100-40 GB GPUs, even when 153 GPUs sit idle.
3. **Stranded resources**: No jobs target V100 hardware because researchers perceive it as "legacy." The 200 GPUs contribute zero productive work despite consuming power and cooling.
4. **Atomic-allocation starvation**: Large jobs remain pending until a full compatible allocation appears, while other resource pools sit idle because policy prevents compatible substitutions or backfill.

Diagnosis: Job templates and organizational practices evolved for a homogeneous A100-80 GB cluster. When infrastructure expanded with heterogeneous hardware, the Distribution Layer policies were never updated. The scheduler correctly implements its configured policy; the policy itself creates the utilization gap.

Solution: Implement tiered resource matching with topology awareness. The policy changes are durable: express capability requirements instead of exact GPU SKUs, reserve NVLink-connected nodes for jobs that need tensor parallelism, backfill compatible smaller jobs into otherwise idle pools, seed appropriate workloads onto older accelerators with lower cost accounting, and make the correct resource request easier than the over-specified template.

Impact: After implementing tiered matching and backfill scheduling, a two-week validation run recovered idle capacity across every pool, with the per-pool before-and-after allocation reported in table 8.9.

Systems lesson: Surface-level diagnosis suggested a scheduling algorithm problem. Deeper analysis showed a policy-infrastructure mismatch: the scheduler correctly implemented policies written for a homogeneous A100-80 GB cluster, but the infrastructure had become heterogeneous. Many utilization problems have this shape. The fix is not always a new scheduler; often it is updating the operational policy so it matches the fleet that actually exists.

Table 8.9: Utilization recovery results: The distinction between "allocated" and "effective" utilization captures the policy-induced utilization gap: before the fix, A100-80 GB GPUs showed high allocation in Slurm but lower actual compute utilization, while strict job templates stranded other capable hardware pools. The A100-80 GB "effective" values are measured utilization of allocated GPUs; cluster-wide utilization is computed over the full heterogeneous fleet.

Pool	Before	After	Change
A100-80 GB	95% allocated, 72% effective	89% allocated, 94% effective	+22 pp
A100-40 GB	64% allocated	88% allocated	+24 pp
V100	0% allocated	71% allocated	+71 pp
Cluster-wide	**60%**	**84%**	**+24 pp**

As table 8.9 shows, this utilization gain adds approximately 246 fully utilized GPUs of productive work. That is equivalent to buying 410 additional raw GPUs if those GPUs ran at the old 60 percent utilization. At $2/GPU-hour, the equivalent raw capacity represents approximately $7M in annual recovered value, achieved through policy changes requiring zero additional hardware investment.

8.9.1 Common utilization anti-patterns

High-level metrics often mask deep inefficiencies. A cluster dashboard may report 85 percent allocation, but productive utilization, the share of cycles actually advancing model state, can be significantly lower. Debugging these gaps requires identifying specific signatures in the GPU telemetry that reveal the underlying bottleneck.

The zombie job represents the most egregious waste: a training process that has crashed or deadlocked but continues to hold GPU resources. In a distributed run training a 175B parameter model, a single rank failure can leave 128 GPUs allocated but idling if the process cleanup fails. The signature is high memory allocation (greater than 300 GB) paired with near-zero SM utilization (less than 5 percent) persisting for longer than a heartbeat timeout (typically 10 minutes). This state often results from CUDA context corruption or a driver-level deadlock that prevents the container runtime from successfully reaping the process, requiring forceful node-level remediation to reclaim the device.

The data starvation pattern manifests as a high-frequency sawtooth wave in GPU utilization, where the device oscillates between 100 percent load (forward/backward pass) and 0 percent load (waiting for the next batch). While the average utilization might appear acceptable at 60 to 70 percent, the GPU is effectively idling for a third of its life. This occurs when the CPU-bound data pipeline cannot sustain the throughput required by the GPU, a common scenario in computer vision where JPEG decoding and augmentation saturate host CPUs. A utilization variance exceeding 30 percent standard deviation is the primary diagnostic indicator, signaling the need for active prefetching, increased loader parallelism, or local NVMe caching to saturate the accelerator.

The communication bottleneck reveals itself through periodic, synchronized drops in SM utilization across all ranks in a distributed group. For the 175B model, synchronizing gradients for 350 GB of parameters requires massive bandwidth; if the network is undersized or congested, the GPUs spend more time waiting for AllReduce to complete than performing computation. The diagnostic metric is the step time ratio: if the total iteration time exceeds $1.5\times$ the compute-only time (forward plus backward passes in isolation), the network has become the dominant constraint. Remediation requires topology-aware placement to keep traffic on high-bandwidth switch tiers or algorithmic changes like gradient accumulation to increase the compute-to-communication ratio.

The memory fragmentation trap is an insidious failure mode in long-running inference servers. A 175B model serving diverse request lengths can end up with 80 percent of its HBM allocated to the KV cache, yet be unable to accept a new request because the free memory is scattered in small, noncontiguous blocks. The GPU appears full to the scheduler but underutilized in terms of throughput. This resembles the classic heap fragmentation problem but with higher stakes: a single restart to defragment dumps hundreds of gigabytes of state. Monitoring the ratio of requested token slots to allocated memory bytes often exposes this gap, which paged cache allocators address by

decoupling logical sequence contiguity from physical memory layout, recovering usable memory and boosting serving throughput by 2–4× without adding hardware.

The debugging example illustrates a broader principle that applies to every system discussed in this chapter: scheduling systems are only as effective as the policies they implement, and policies must evolve alongside infrastructure. When infrastructure changes (new hardware generations are added, network topologies are upgraded, workload mixes shift from training-dominant to serving-dominant), the policies encoded in scheduler configuration, job templates, and organizational practices must be re-evaluated and updated. The most expensive scheduling bug is often not a software defect but a policy that was correct for the old infrastructure and was never updated for the new one.

This principle connects scheduling mechanics to fleet governance. Technical policies (GPU type matching, gang scheduling timeouts, preemption grace periods) interact with organizational policies (team quotas, priority hierarchies, cost allocation) and human behavior (job template reuse, resource request habits, queue submission patterns). Effective fleet orchestration requires attending to all three layers simultaneously.

8.10 Fallacies and Pitfalls

Those policy-infrastructure mismatches recur as a small set of misconceptions that can make an apparently healthy scheduler waste expensive fleet capacity. It is tempting to look at a cluster running at 98 percent utilization and declare victory, only to discover that researchers have stopped submitting jobs entirely because wait times have stretched into weeks. Cluster orchestration is riddled with counterintuitive traps like this, where optimizing a single metric destroys overall system value.

Fallacy: *More sophisticated scheduling algorithms always improve utilization.*

Engineers facing low utilization often reach for more advanced schedulers, assuming the algorithm is the bottleneck. As the debugging example in section 8.9 demonstrates, the root cause is frequently policy misconfiguration, not algorithmic limitation. A simple FIFO scheduler with correct policies (capability-based matching, backfill with timeouts, appropriate gang scheduling constraints) often outperforms a sophisticated scheduler with incorrect policies. Before upgrading the scheduler, audit the policies: confirm that users are not requesting resources they do not need, that heterogeneous resources are properly exposed, and that gang-scheduling timeouts are configured.

Pitfall: *Treating all GPU-hours as equal when measuring utilization.*

Cluster dashboards commonly report "GPU utilization" as a single percentage, averaging across all GPUs and all workloads. This metric hides critical information. A cluster might report 80 percent utilization where 60 percent is productive training, 15 percent is idle GPUs allocated to jobs waiting for gang completion, and 5 percent is GPUs running data loading with no active computation. Effective monitoring distinguishes between allocated utilization (GPUs assigned to jobs), compute utilization (GPUs executing kernels), and productive utilization (GPUs advancing useful training or serving requests). Only the last metric correlates with actual value delivered.

Fallacy: *Gang scheduling is always necessary for distributed training.*

Gang scheduling prevents deadlock for synchronous training, but not all distributed training is synchronous. Asynchronous training methods tolerate worker arrivals and departures, and elastic training frameworks handle variable worker counts. For workloads that can operate asynchronously or elastically, relaxing the gang scheduling requirement dramatically improves scheduling flexibility and reduces queue wait times. The trade-off is potential convergence degradation from stale gradients or batch size variability, but for many practical workloads (hyperparameter sweeps, fine-tuning, pretraining with adaptive batch size), this trade-off is favorable.

Pitfall: *Setting static quotas based on peak demand.*

Organizations commonly set team quotas to handle worst-case demand, reasoning that teams need guaranteed access during crunch periods. If Team A's peak demand is 500 GPUs and Team B's is 300 GPUs, the cluster needs 800 GPUs with static quotas. In practice, peaks rarely coincide. Hierarchical fair-share with borrowing can serve both teams' peak demands with a 600-GPU cluster, because when Team A peaks, Team B is typically at moderate demand and can yield borrowed capacity. Static quotas at peak levels waste 25 to 40 percent of cluster capacity on guaranteed-but-unused allocations.

Fallacy: *Spot instances are always cheaper for training.*

The 60 to 90 percent discount on spot instances creates the impression of automatic savings. The effective cost depends on interruption frequency, checkpoint overhead, and restart time. For a large model with 15-minute checkpoint times and 30-minute restart times, a spot interruption costs 45 minutes of productive time. If interruptions occur every 2 hours, the effective cost rises enough that the nominal 65 percent discount shrinks to about 52 percent savings. For 100B+ parameter models with slow checkpointing on clusters with frequent spot interruptions, on-demand instances can actually be cheaper when accounting for all overhead. The decision requires quantitative analysis of the specific workload and spot market conditions, not blanket assumptions about cost savings.

Pitfall: *Ignoring topology when scheduling distributed training.*

Schedulers that treat GPUs as interchangeable units create allocations where tensor parallel groups span nodes (forcing NVLink-speed operations over InfiniBand) or AllReduce groups cross spine switches (adding latency and congestion). The 15 to 30 percent throughput penalty from poor topology placement accumulates over multi-week training runs, potentially wasting more resources than would be lost by waiting for a topology-aware allocation. Topology-aware scheduling increases scheduling complexity and may reduce packing efficiency, but for large distributed training jobs, the throughput improvement often justifies the trade-off.

Fallacy: *GPU utilization alone is a sufficient autoscaling signal.*

GPU utilization is a poor proxy for inference health. A GPU can be 90 percent utilized but still violating latency SLOs because the utilization comes from processing a backlog of queued requests. Conversely, a GPU at 40 percent utilization might be perfectly healthy if it is serving low-latency requests with headroom for bursts. The right metrics are queue depth, P99 latency relative to SLO, and KV cache memory pressure for LLM workloads. Relying on utilization creates a lagging indicator that only triggers scaling *after* performance has already degraded, whereas queue depth and SLO margin provide leading indicators that allow the fleet to scale *before* the user experience suffers. Furthermore, for LLMs, memory exhaustion from KV cache growth can occur at low compute utilization, causing requests to stall or fail. An autoscaler watching only compute utilization will miss this failure mode entirely, leaving the service degraded despite apparent capacity.

Pitfall: *Using elastic training to avoid gang-scheduling decisions.*

Elastic training complements but does not replace gang scheduling. While it allows jobs to resize, the resizing steps themselves often have atomic requirements. Elastic training works for data parallelism but not tensor parallelism (which requires a fixed number of GPUs per group). A 175B model using 8-way tensor parallelism needs exactly 8 GPUs per tensor-parallel group to function; this is a gang constraint that elastic training cannot relax. Elastic training adjusts the number of data-parallel replicas, not the intra-replica parallelism configuration, meaning the scheduler must still enforce gang constraints for the base unit of the model. If a scheduler naively places these 8 GPUs across different racks or fails to allocate them atomically, the tensor parallel group will either fail to initialize or suffer catastrophic performance degradation.

Fallacy: *The scheduler needs no workload-specific validation.*

Scheduler configuration requires continuous tuning as workloads evolve. Default configurations in Slurm or Kubernetes are optimized for generic workloads, often prioritizing simple fairness over throughput. ML-specific tuning (gang scheduling timeouts, topology-aware placement weights, preemption grace periods, and fair-share decay rates) can improve utilization by 15 to 25 percent. Organizations that deploy a scheduler and never revisit its configuration leave significant value on the table. A default preemption grace period might be too short for a large model to checkpoint, causing wasted work, while an untuned topology weight might fragment the cluster unnecessarily. As the cluster grows and the workload mix shifts (for example, from training-heavy to inference-heavy), the effective scheduling parameters shift with it. Treating the scheduler as a set-and-forget appliance lets the fleet drift into inefficiency.

Pitfall: *Treating high utilization as proof of cluster health.*

A cluster running at 95 percent utilization sounds efficient but may indicate a problem: insufficient capacity that is choking experimentation velocity. When utilization consistently exceeds 85 to 90 percent, queue times grow rapidly following the well-known M/M/1 queuing result from stochastic process theory. In this model, average wait time is proportional to $\rho/(1-\rho)$ where ρ is utilization; at 90 percent utilization, average wait is $9\times$ the service time, and at 95 percent, it is $19\times$. Researchers

waiting hours or days for resources conduct fewer experiments, test fewer hypotheses, and iterate more slowly on model designs. This reduced experimentation velocity has real costs that are harder to quantify than GPU-hours but may be larger in aggregate. The effective utilization target balances resource efficiency against researcher productivity, often landing between 70 and 85 percent for training clusters where queue responsiveness directly affects research output. Inference clusters, where requests are served immediately or not at all, often target even lower utilization (50 to 70 percent) to maintain latency headroom for traffic spikes.

Recognizing these fallacies prevents platform teams from chasing false optimizations that look good on dashboards but degrade developer velocity. The core orchestration principles that successfully navigate these trade-offs share a common thread: they treat scheduling as an engineering discipline, not an administrative function.

8.11 Summary

Fleet orchestration transforms raw data center capacity into productive ML infrastructure. The scheduling algorithms, placement strategies, and resource management policies examined in this chapter determine whether thousands of isolated GPUs function as one coherent computing platform or as a fragmented collection of expensive servers. The difference between those outcomes is not hardware capability alone, but the scheduler's ability to maintain useful state under partial failures, network partitions, stale resource views, and competing consistency and availability requirements. Slurm and Kubernetes embody different trade-offs along that spectrum, and large fleets may use both because batch training and self-healing service deployment stress the scheduler in different ways.

The chapter's central technical lesson is that placement is performance. Topology-aware scheduling exploits the communication hierarchy of GPU clusters, improving training throughput by 15 to 30 percent when tensor-parallel, pipeline-parallel, and data-parallel groups are placed on the fabric tiers that match their traffic. Elastic training, preemption-aware checkpointing, and spot-instance policies add economic flexibility, but only when the job can survive changing worker counts or interruptions. Research schedulers such as Tiresias, Gandiva, Themis, and Pollux make the same point algorithmically: ML jobs have observable structure, including iterative progress, heavy-tailed durations, sunk-cost economics, and convergence-dependent scaling, so schedulers that exploit that structure can improve average completion time by 40 to 60 percent over generic policies.

The objective function changes again at serving time. Training clusters can trade queue wait against utilization, but inference fleets must protect tail latency under bursty demand. Autoscaling therefore needs queue depth, P99 latency, and KV-cache pressure rather than GPU utilization alone; multi-tenancy needs MIG, MPS, temporal slicing, quotas, borrowing, and chargeback policies that balance isolation against fleet-wide efficiency. The same scheduler that looks efficient on a utilization dashboard can still be mismanaging the fleet if researchers wait days for experiments, inference SLOs degrade under burst load, or fragmented GPUs cannot form the gang allocations that training jobs require.

Mastery of fleet orchestration is therefore diagnostic and economic as much as algorithmic. Zombie jobs, data starvation, topology mismatch, and resource fragmentation are symptoms of scheduler policy failing to match infrastructure reality; adding more nodes often hides the symptom without fixing the cause. With compute nodes defined, networks connected, storage configured, fault tolerance in place, and schedulers running, the infrastructure layers of the fleet stack now form a coherent platform rather than a rack of independent machines.

Key Takeaways: Scheduling is systems engineering

- **Distributed scheduling is fundamentally hard**: Cluster scheduling faces challenges (partial failures, network partitions, state inconsistency) that single-machine schedulers never encounter. The CAP theorem forces trade-offs between consistency and availability that shape every scheduling system's design.
- **Gang scheduling prevents deadlock but reduces flexibility**: Gang scheduling gives distributed training atomic resource allocation to prevent hold-and-wait deadlocks, but

rigid gang scheduling wastes resources when combined with backfill timeouts and elastic training alternatives.

- **Topology determines performance**: Where GPUs are placed within the cluster hierarchy matters as much as how many GPUs a job receives. NVLink vs. InfiniBand placement decisions can create 15 to 30 percent throughput differences for the same job on the same hardware.
- **ML-specific scheduling outperforms generic approaches**: Exploiting workload characteristics (predictable resource needs, iterative computation, diminishing returns) enables 40 to 60 percent improvements in job completion time compared to general-purpose FIFO or fair-share policies.
- **Utilization is a first-order economic driver**: Improving cluster utilization from 50 percent to 80 percent on a large cluster effectively adds thousands of GPUs worth of capacity. Closing this gap through policy and algorithm work pays back faster than buying additional accelerators.
- **Policies must evolve with infrastructure**: Scheduling algorithms are only as effective as the policies they implement. When infrastructure changes (heterogeneous hardware additions, topology upgrades, workload mix shifts), policies must be re-evaluated and updated.

The fleet now has every physical part it needs: compute, communication, storage, and the resilience to keep them running. None of it produces anything until something decides what runs where, and that decision is pure coordination. A scheduler that chases peak utilization will deadlock the first time two large jobs need the same nodes, so a scheduler that guarantees progress must leave some capacity deliberately unused. That gap, between the throughput a fleet could reach and the throughput it can safely sustain, is the price of coordination, and orchestration is the discipline of spending it where it buys the most finished work. At fleet scale the binding constraint moves up the stack again, from the speed of the hardware to the quality of the policy that hands it out.

What's Next: From orchestration to optimization

We have organized the operational layer of the Machine Learning Fleet, the "who gets what, when, and where" of resource management. Yet the fleet's purpose is not to run training jobs; it is to produce models that serve users. The next challenge is extracting maximum throughput from the hardware once it has been orchestrated. Chapter 9 examines the techniques that do so: operator fusion, precision engineering, graph compilation, and execution scheduling. These optimizations determine whether the fleet's expensive silicon delivers its theoretical potential or wastes cycles on inefficient execution. The orchestration layer ensures the right resources are allocated; performance engineering ensures those resources are used effectively.

Research Questions: For further inquiry

- How should a scheduler trade utilization, fairness, topology locality, and deadlock prevention for distributed training jobs?
- How should placement policies incorporate collective costs and fault-domain boundaries?
- When do preemption, spot capacity, and elastic training reduce total cost rather than increase lost work and variance?
- Which signals beyond accelerator utilization should drive autoscaling and cluster health decisions?

III

DEPLOYMENT AT SCALE

Part III

Deployment-at-Scale Principles

Parts I and II built the Fleet and trained the model. Production deployment now becomes the test of operational accountability. Part III shifts the focus from the single, massive training run to the global inference fleet: the thousands of geographically distributed servers and billions of edge devices that serve model outputs to users in real time. This transition from training to production fundamentally changes the engineering constraints: from maximizing throughput on a static dataset to minimizing latency on a dynamic stream of requests, all within the physics of global serving.

In this regime, engineering becomes a negotiation with the user's experience. The system can optimize for high batch throughput in the data center, but at the risk of violating the user's latency budget. Deploying a model to the edge eliminates network latency, but introduces the tight memory and power envelopes of mobile and Internet of Things (IoT) hardware. These principles govern the performance engineering and operational economics of deploying intelligence at scale.

The first constraint is imposed by the mechanics of generation itself.



Principle 14: The Autoregressive Bottleneck

Invariant: In low-batch dense transformer decode, throughput is memory-bandwidth bound because the model weights must be streamed across the device for *every generated token*. Higher batch sizes amortize weight streaming, and key-value (KV) cache traffic shifts the binding constraint at long context.

Implication: Throughput initially improves with batch size by amortizing weight reads across requests, until per-device latency budgets, KV cache capacity, or sustained compute become the binding ceiling. Serving systems therefore need batching, cache-management, and memory-layout policies that keep bandwidth useful without violating latency SLOs.

The same accounting extends from device bandwidth to fleet economics.



Principle 15: The Serving Cost Dominance Law

Invariant: For high-traffic production models, cumulative inference cost can exceed the one-time training cost by 10–100$\times$ over the model's lifetime.

Implication: Inference efficiency often becomes the dominant lifecycle optimization target. Optimization efforts (quantization, distillation, sparsity) should be focused on the inference path when traffic volume, product lifetime, and retraining cadence make serving cost dominate.

At scale, routing policy becomes another lever for protecting latency.

Principle 16: The Power of Two Choices (P2C)

Invariant: Under an idealized balls-into-bins model with R replicas, querying just two random replicas and selecting the least-loaded one reduces the maximum load on any replica from $\Theta(\log R / \log \log R)$ to $\Theta(\log \log R)$.

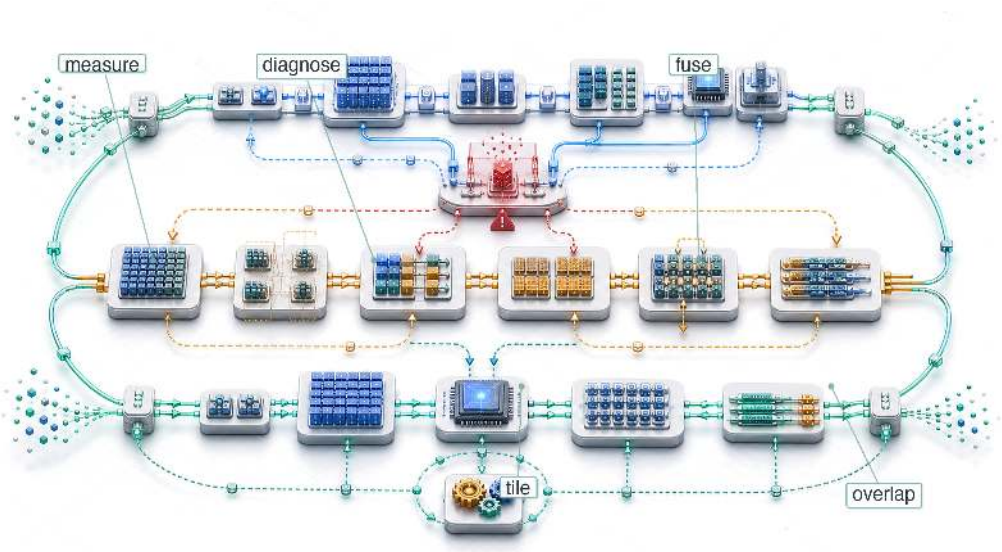
$$\Theta(\log R / \log \log R) \rightarrow \Theta(\log \log R)$$

Implication: Simple randomized load-balancing algorithms produce dramatically lower maximum-load bounds with $\mathcal{O}(1)$ coordination overhead. Under serving queueing assumptions, the lower maximum load translates into reduced tail latency at scale.

Part III takes the trained model from the cluster to the world. The path begins with performance engineering, where precision choices, kernel behavior, and compilation determine how much of the hardware's theoretical capacity the system can actually use. It then moves to inference at scale, where request batching, cache management, routing, and sharding turn single-device efficiency into fleet-level latency control. Edge intelligence pushes the same discipline into devices with much smaller memory, power, and connectivity budgets, while operations at scale closes the loop by monitoring the deployed fleet, coordinating platform ownership, and responding when production behavior diverges from design assumptions. Together, these chapters develop the operational discipline that turns a trained model into a durable production system.

9

Performance Engineering

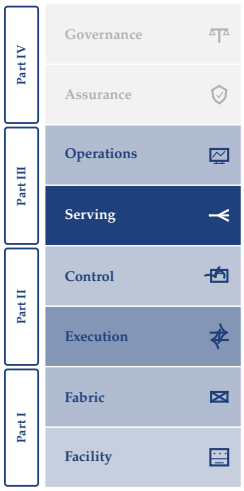


9.1 The Memory Wall and Roofline Diagnosis
9.2 Serving Regimes: Batch Size and Prefill-Decode
9.3 Operator Fusion and Kernel Engineering
9.4 Precision Engineering
9.5 Graph Compilation
9.6 Algorithmic Performance Transformations
9.7 Communication-Computation Overlap
9.8 System Profiling
9.9 Measurement at Scale
9.10 The Optimization Playbook: A 70B LLM Case Study
9.11 Fallacies and Pitfalls
9.12 Summary

Purpose

How do we make billion-parameter models run on millisecond timescales?

Model compression reduces the size of the model's computation. Performance engineering reshapes that computation to match the physics of the hardware. The distinction matters: a quantized model loaded naively into an accelerator kernel that reads every weight from off-chip memory wastes the very bandwidth savings that quantization was designed to provide. Real performance comes from understanding the full path a tensor travels, from registers through on-chip memory to high-bandwidth memory and back, and then engineering each step to eliminate wasted movement. This chapter develops the system-level optimization techniques that bridge the gap between a theoretically efficient model and a production artifact that saturates hardware. The levers are operator fusion and tiling strategies that keep data in fast local memory, precision formats that double effective bandwidth, compilation frameworks that automate kernel selection, and algorithmic innovations like speculative decoding and sparse expert routing that change the performance equation. Together, they transform a model that should be fast into one that is fast, often by an order of magnitude. That order of magnitude comes entirely from the compute term of the C^3 taxonomy: performance engineering extracts more useful work from the cycles the fleet already owns.



Learning Objectives

- Apply roofline analysis to classify ML kernels as compute-bound, memory-bound, or launch-limited on target accelerators
- Analyze prefill, decode, and batch-size regimes to predict latency-throughput behavior in LLM serving
- Design fusion, tiling, and CUDA graph strategies that reduce HBM traffic and launch overhead
- Select precision, compilation, and runtime optimizations from bandwidth, quality, and deployment constraints
- Evaluate speculative decoding and MoE routing using acceptance rates, batching limits, and AllToAll costs
- Diagnose serving bottlenecks with profilers, roofline plots, and fleet-efficiency metrics
- Synthesize a 70B serving optimization plan across compute, communication, coordination, and quality trade-offs

9.1 The Memory Wall and Roofline Diagnosis

An H100 GPU capable of 989 TFLOP/s of FP16 compute can still show single-digit compute utilization during small-batch language-model decode. The processor is not short on arithmetic; it is starving for data. Performance engineering operates inside that constraint: the memory wall, where moving bytes from memory to compute units can cap throughput long before Tensor Cores reach their advertised peak.

Placement, synchronization, recovery, and scheduling can put the workload on the right hardware and keep it alive. The remaining problem is local execution: expensive silicon can still sit idle after work arrives. In the fleet stack shown in figure 1.13, performance engineering is the optimization discipline within the Serving Layer, reaching down into the Distribution and Infrastructure layers when kernels, memory hierarchy, interconnects, or framework overhead determine achieved throughput. The answer is usually data movement, so performance engineering begins with the memory hierarchy and then follows the consequences through fusion, precision, compilation, and algorithmic changes that move fewer bytes, launch less overhead, or do different work entirely.

9.1.1 The iron law of ML performance

The memory wall is one term in a larger budget. Equation 9.1 states the iron law of ML system performance, decomposing execution time into three competing costs:

$$T = \frac{D_{\text{vol}}}{\text{BW}} + \frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}} + L_{\text{lat}} \tag{9.1}$$

In overlapped execution, the roofline-style simplification replaces the sum of compute and data movement with the slower exposed term:

$$T \approx \max\left(\frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}}, \frac{D_{\text{vol}}}{\text{BW}}\right) + L_{\text{lat}}$$

The inherited iron law decomposes execution time into three terms. The compute fraction represents the total floating-point operations divided by realized hardware throughput. The data fraction represents total bytes transferred divided by memory bandwidth. The roofline approximation then asks which exposed term dominates at a given operating point: increasing compute throughput for a memory-bound workload, for example, does not materially improve performance until the memory term is reduced. The overhead term captures everything else: kernel launch latency, synchronization, communication, and software stack inefficiency. PyTorch training loops expose a particularly large slice of this term: every kernel launch must traverse the Python Global Interpreter Lock (GIL) and the framework's CPU dispatcher before reaching the GPU command queue, spending tens

of microseconds in Python dispatch per operation before any GPU work begins. This is precisely why ML framework developers prioritized `torch.compile(mode="reduce-overhead")` and CUDA Graphs: both trace away the Python dispatch path entirely, converting repeated GPU submissions into a single native replay that bypasses the GIL and dispatcher on every step.

Standard model compression (pruning, quantization, distillation) shrinks the model's intrinsic work, performing fewer operations on smaller data. System optimization addresses the complementary problem: shrinking the gap between that work and the hardware's peak by moving the same terms through implementation rather than model change, keeping data in fast memory, packing each transfer more densely, and removing software overhead.

D C L

Autoregressive transformer decode is often memory-bound: the data-movement term dominates the iron law.

Several levers map directly to those terms. When memory traffic dominates, operator fusion and tiling reduce the exposed data-volume term by eliminating intermediate HBM round-trips. A fused sequence that keeps its intermediates in SRAM can shrink the exposed memory-access term dramatically, often by 10–30× for attention computation. Precision engineering attacks the same numerator from a different angle: FP8, INT4, and KV-cache compression (storing the per-token attention keys and values in fewer bytes) represent each value in fewer bytes, so the same physical bandwidth carries more useful model state.

Other levers change the overhead or compute terms. Graph compilation, including torch.compile, Accelerated Linear Algebra (XLA), and TensorRT, reduces overhead by eliminating kernel launch gaps, fusing operations, and optimizing memory allocation across the graph. Communication-computation overlap makes distributed communication concurrent with useful work, removing it from the critical path when $T_{\text{comm}}(N) - T_{\text{overlap}} \leq T_{\text{compute}}/N$ (that is, communication finishes before the next layer's computation completes). This inequality is the exposed-time test, not a new law: overlap helps only when useful computation is large enough to cover the communication. Algorithmic innovations such as speculative decoding and mixture-of-experts (MoE) change the compute term itself by making the model perform a different computation that preserves the output contract at lower exposed cost.

Each technique attacks a different term, and this taxonomy guides optimization strategy: diagnose which term dominates (using the roofline model from section 9.1.5), then apply the technique targeting that term. Applying a technique that targets the nondominant term wastes engineering effort.

Before using this diagnostic process, check how each term in the iron law maps to a practical optimization lever.

> **Checkpoint 9.1: The iron law of performance**
>
> Verify your understanding of system-level performance diagnosis:
>
> - ☐ **Memory-bound limit**: Determine whether increasing the GPU's clock frequency (increasing FLOP/s) improves a memory-bound workload.
> - ☐ **Quantization term**: Identify which term in equation 9.1 changes when quantization moves the representation from FP16 to INT4.
> - ☐ **Overhead term**: Define the "Overhead" term in the iron law and explain why graph compilation targets it specifically.
> - ☐ **Compute-bound bandwidth**: Evaluate the claim that memory bandwidth is irrelevant to the performance of a compute-bound system.
> - ☐ **Iron-law decomposition**: For a step that is 60 percent data movement, 30 percent compute, and 10 percent latency, estimate which single optimization buys the most wall-clock time, and what the iron law predicts once you halve that dominant term.

The same diagnostic process can be codified as a decision flowchart, mapping each bottleneck to its corresponding optimization technique. The flowchart in figure 9.1 makes the sequencing explicit: rule out I/O, CPU, and communication overheads before classifying the remaining workload as compute-bound or memory-bound.

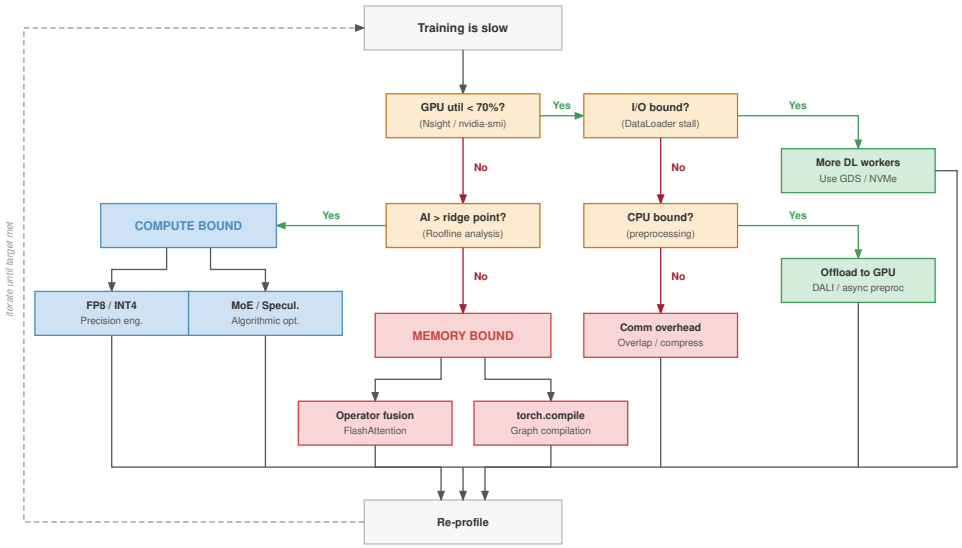


Figure 9.1: Iron Law Diagnostic Flowchart: A training-slow-down troubleshooting flow that starts from observed symptoms (slow training, low GPU utilization), branches through I/O, CPU, and communication-overhead checks, and classifies the workload as compute-bound or memory-bound at the leaves. Compute-bound paths point to precision engineering and algorithmic change; memory-bound paths point to operator fusion and tiling. Overhead-bound symptoms (graph dispatch, communication) are surfaced earlier in the flow and addressed before the final compute/memory classification.

The central lesson of figure 9.1 is that profiling must precede optimization: applying operator fusion to a compute-bound workload, or precision engineering to an overhead-bound one, yields zero improvement regardless of implementation quality. Misdiagnosis is not only wasted effort; a performance change shipped blind can move the system to the wrong point on its operating boundary.

That boundary is the **efficiency frontier**, the Pareto-optimal curve of model quality vs. system throughput. A model on the frontier cannot improve throughput without sacrificing quality, or vice versa. Thompson et al. (2021) show why this frontier matters for deep learning: quality improvements have required rapidly increasing compute, making efficiency gains central to continued progress. Performance engineering pushes the frontier outward by making each quality level achievable at higher throughput, or equivalently, by making each throughput level achievable at higher quality. An organization's goal is not merely to reach the frontier but to find the point on it that best matches their latency, throughput, cost, and quality requirements.

The multi-dimensional nature of this frontier makes optimization challenging. Table 9.1 identifies the five dimensions that performance engineers must balance before choosing an optimization target.

Table 9.1: Efficiency Frontier Dimensions: Performance optimization balances measurable throughput, latency, cost, quality, and memory constraints rather than a single hardware-utilization number.

Dimension	How it is measured	What it constrains	Typical tension
Throughput	Tokens/second or requests/second	How much work the system completes per unit time	Larger batches improve throughput but often degrade latency
Latency	Time-to-first-token and inter-token latency	How quickly the system responds to individual requests	Lower latency can require smaller batches and higher per-token cost
Cost	Dollars per million tokens	Economic efficiency of the system	Cheapest configurations may miss latency or quality targets

Dimension	How it is measured	What it constrains	Typical tension
Quality	Perplexity, benchmark accuracy, or human preference	Accuracy and usefulness of model outputs	Precision reduction and speculation require quality guardrails
Memory	Peak GPU memory	Feasible batch size and sequence length	Larger contexts and batches consume capacity needed for model state

The dimensions in table 9.1 interact in nonobvious ways. Increasing batch size improves throughput and cost efficiency but degrades latency. Reducing precision improves throughput and memory but may degrade quality. Speculative decoding improves latency but may increase per-token cost. The performance engineer's task is to navigate these trade-offs guided by the application's specific requirements.

A real-time chatbot prioritizes latency (time-to-first-token under 200 ms, inter-token latency under 50 ms) and may tolerate higher per-token cost. A batch processing pipeline for document summarization prioritizes throughput and cost, tolerating seconds of latency. A medical diagnostic system prioritizes quality above all else, accepting lower throughput and higher cost. Each application maps to a different optimal point on the efficiency frontier, and the performance-engineering toolbox provides the methods for reaching that point. To make this concrete, consider two deployment configurations for the same 70B large language model (LLM).

Configuration A is latency-optimized: FP16 weights, batch size 1, speculative decoding enabled. The model replica produces approximately 50 tokens/second with 20 ms inter-token latency while occupying 8 H100 GPUs for a single user stream. Cost: approximately $0.12 per 1,000 output tokens.

Configuration B is throughput-optimized: INT4 weights, batch size 64, no speculation. Each H100 serves approximately 4,000 tokens/second aggregate throughput across all batched requests, with 120 ms inter-token latency per request. Cost: 4 GPUs serving 64 concurrent users, approximately $0.002 per 1,000 output tokens.

Configuration B achieves 60× lower cost per token than Configuration A, but at 6× higher latency. Neither configuration is objectively "better"; they represent different points on the efficiency frontier, optimized for different applications. Performance engineering is the discipline of navigating between these points.

9.1.2 The memory wall

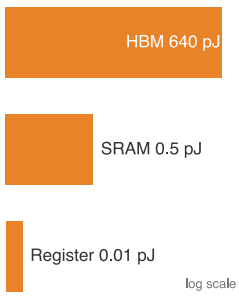
The efficiency frontier establishes what we are optimizing toward. The physics of memory bandwidth determines where we start. Many accelerator-based ML performance problems begin with the same observation: memory bandwidth, not compute, is the bottleneck. Consider a single autoregressive decoding step in a large language model. The model reads its full weight matrix from High Bandwidth Memory (HBM)[1] to generate a single token, performing only one or two multiply-accumulate operations per weight loaded.

An NVIDIA H100 delivers 1979 TFLOP/s of FP8 compute but only 3.35 TB/s of memory bandwidth. If every byte loaded from memory does fewer than 590.7 FLOP/byte arithmetic operations, the compute units sit idle, starved for data. This gap between compute capability and memory delivery rate is the memory wall, and it defines the landscape within which all performance engineering operates.

The memory wall represents a fundamental physical constraint rather than a temporary engineering limitation. Moving data costs energy proportional to distance. Accessing a value from on-chip SRAM (L1 cache) costs approximately 0.5 pJ, while fetching the same value from off-chip HBM costs roughly 640 pJ, a ratio of 1280×. Manufacturing constraints limit the amount of SRAM that can sit close to the compute units. HBM provides capacity (the H100 offers 80 GB) but at physically greater distance, requiring the data to traverse longer wires. The fundamental tension is that models need gigabytes of parameters and state, but physics dictates that only kilobytes of data can be near the compute units at any given moment.

The capacity-bandwidth tension shapes the optimization space. Operator fusion reduces the number of trips to HBM by combining operations so that intermediate results stay in SRAM. Precision engineering reduces the number of bytes per trip by representing values in FP8 or INT4 instead

1 **HBM (High Bandwidth Memory)**: Achieves its bandwidth by vertically stacking DRAM dies connected through thousands of through-silicon vias (TSVs), a 3D packaging technique first commercialized by SK Hynix in 2013. Despite the "high bandwidth" label, HBM's 3.35 TB/s on the H100 is still 200–600× slower than on-chip SRAM access, making the memory hierarchy gap the central constraint of performance optimization.



Access energy climbs from registers out to HBM.

of FP16. Tiling strategies restructure algorithms to maximize data reuse within SRAM. Graph compilers automate these transformations. Each technique attacks a different term in the same fundamental equation: minimize the ratio of bytes moved to operations performed.

9.1.3 The GPU memory hierarchy

To understand why the memory wall exists, consider the physical structure of a GPU memory system. Table 9.2 summarizes the four levels by scale, access cost, and the optimization constraint each one imposes.

Table 9.2: GPU Memory Hierarchy Constraints: Faster GPU storage is smaller, closer to the compute units, and more explicitly managed, while HBM supplies capacity at much higher latency and energy cost.

Level	Scale on H100	Access cost	Optimization constraint
Registers	256 KB per SM across 132 SMs, about 33 MB total	One clock cycle and ~0.01 pJ per access	Private to each thread; FP32 accumulators can force register spilling to L1/shared memory at 20–30 clock cycles per access
Shared memory (SRAM)	Up to 228 KB configurable shared memory per SM	20–30 clock cycles (~20 ns) and ~0.5 pJ per access	Shared within a thread block; operator fusion is profitable when intermediates fit here instead of returning to HBM
L2 cache	50 MB on-chip buffer	About 200 clock cycles (~130 ns)	Captures reuse automatically across SMs but cannot be explicitly managed by kernel authors
High Bandwidth Memory (HBM)	80 GB at 3.35 TB/s bandwidth	About 300 ns and 640 pJ per access	Supplies model and activation capacity, but reading the full device takes about 24 ms, far longer than real-time inference latency targets

The table makes register pressure a first-class design constraint in custom Triton and CUDA kernels: tile size controls both arithmetic intensity and register demand. It also explains why shared memory and L2 reuse matter so much for attention. If KV cache entries or fused intermediates remain on chip, the kernel avoids the slow, high-energy HBM round trip that dominates low-arithmetic-intensity operations.

The energy cost of data movement has a direct economic consequence at data center scale. Consider a training cluster of 1,000 H100 GPUs, each performing approximately 10^{12} memory accesses per second during a memory-bound workload. If each access reads from HBM at 640 pJ, the memory subsystem alone consumes approximately 640 W per GPU, a significant fraction of the H100's 700 W TDP. If operator fusion moves half of those accesses from HBM to SRAM (at 0.5 pJ each), the per-GPU memory power drops by approximately 320 W. Across 1,000 GPUs, this saves 320 kW, equivalent to powering roughly 250 homes. This is not a secondary consideration; at cloud electricity prices, the annual cost difference is substantial, and it scales linearly with cluster size. The physics of data movement is not merely a performance constraint; it is an economic one.

The performance engineering challenge reduces to a data placement problem: keep the data that the compute units need in the fastest memory that can hold it. When a kernel reads a tensor from HBM, processes it, and writes the result back to HBM, the HBM round-trip dominates execution time for any operation with low arithmetic intensity. Every technique here shares the same goal: keeping data closer to compute for longer.

> Systems Perspective 9.1: Analogy: The scholar's library
>
> The GPU memory hierarchy parallels a scholar researching in a library:
>
> - **Registers** (33 MB) are *working memory*: instant access, but capacity is small enough to hold only a few values at once.
> - **Shared Memory (SRAM)** is a *desk*: very fast to reach, but capacity fits only a few open references.

- **L2 Cache** (50 MB) is a *book cart* beside the desk: a small access cost, holding a moderate working set.
- **HBM** (80 GB) is the *library basement*: holds everything that could be needed, but each round trip costs hundreds of nanoseconds.

Performance engineering is the art of minimizing trips to the basement.

9.1.4 The widening gap

The memory wall is not static; in the accelerator generations compared here, compute throughput has grown faster than off-chip memory bandwidth. Memory bandwidth improves more slowly because the physics of off-chip signaling and the economics of HBM manufacturing limit how fast data can leave the chip.

Table 9.3 quantifies how the hardware balance shifts across accelerator generations. The key column is the ridge point: as compute grows faster than bandwidth, more operators need higher arithmetic intensity just to remain compute-bound.

Table 9.3: The Widening Memory Wall: Compute throughput has increased 18× from V100 to B200 over seven years, while memory bandwidth has increased only 8.9×. The ridge point has roughly doubled overall (V100 to B200), pulling more workloads into the memory-bound regime over time, though the per-generation movement is not monotonic (B200 sits slightly below H100).

GPU	Year	Peak FP16 (TFLOP/s)	HBM BW (TB/s)	Ridge Point (FLOP/byte)
V100	2017	125	0.9	139
A100	2020	312	2.04	153
H100	2022	989	3.35	295
B200	2024	2,250	8	281

For the FP16/BF16 table as written, the ridge point increased from 139 FLOP/byte on the V100 to 281 FLOP/byte on the B200, about a 2× increase. An operation with arithmetic intensity of 200 FLOP/byte was compute-bound on the V100 and A100, but memory-bound on the H100 and B200. Performance engineering techniques targeting memory efficiency, fusion, precision, and tiling therefore become more important as the ridge point rises, not less. The systems lesson is not that one named kernel lasts forever, but that reducing exposed memory traffic becomes more valuable when compute grows faster than bandwidth.

9.1.5 The roofline model

Section 2.3.2 introduced the Roofline Model[2] (Williams et al. 2009) and arithmetic intensity as the framework for diagnosing whether a workload is compute-bound or memory-bound on a given accelerator, and computed the H100's ridge point. We recall it here only to push it to fleet scale: how the ridge point shifts across hardware generations, how FP8 moves it, where production ML workloads fall relative to it, and how batch size walks a workload across it. As a reminder, the model plots achievable performance as a function of arithmetic intensity, the ratio of floating-point operations to bytes transferred from memory, and the intersection of the two regimes is the ridge point[3].

Definition 9.1: Arithmetic intensity

Arithmetic Intensity (I) is the ML workload ratio of floating-point operations performed to the number of bytes transferred from memory (FLOP per byte).

1. **Significance**: It characterizes the computational density of a workload. It is the independent variable in the Roofline Model, determining whether a system operates in the bandwidth-bound (BW) or compute-bound (R_{peak}) regime.

2 **Roofline Model**: The original framing targets multicore CPUs, but the same ceiling diagram applies to accelerators: two numbers (peak FLOP/s and peak bandwidth) define the entire performance envelope. This same simplicity informs GPU purchasing decisions for ML inference, where the ridge point determines whether a workload benefits from faster compute or faster memory.

3 **Ridge Point**: The intersection of the memory-bound and compute-bound lines on a Roofline plot. It represents the minimum arithmetic intensity required to reach peak hardware performance (R_{peak}). For an H100 GPU (FP16), the ridge point is roughly 295 FLOP/byte; if an operator's intensity is below this "ridge," it will never saturate the Tensor Cores, regardless of how much compute is available.

2. **Distinction**: Unlike peak throughput (a hardware property), arithmetic intensity is an algorithmic property that measures how effectively a workload reuses data once it is loaded into the processor.
3. **Common pitfall**: A frequent misconception is that arithmetic intensity is fixed for a model. In reality, it varies by implementation: techniques like operator fusion increase arithmetic intensity by keeping data in local registers, while increasing batch size increases arithmetic intensity for layers with high parameter reuse.

For a given accelerator with peak compute R_{peak} (in FLOP/s) and peak memory bandwidth BW (in bytes/s), equation 9.2 gives the achievable performance of a workload with arithmetic intensity I (in FLOP/byte):

$$\text{Achievable FLOP/s} = \min(R_{\text{peak}}, \text{BW} \times I) \tag{9.2}$$

Equation 9.3 locates the ridge point where these two limits intersect:

$$I_{\text{ridge}} = \frac{R_{\text{peak}}}{\text{BW}} \tag{9.3}$$

Workloads with $I < I_{\text{ridge}}$ are memory-bound: their performance is limited by how fast data can be loaded, not how fast it can be processed. Workloads with $I > I_{\text{ridge}}$ are compute-bound: the arithmetic units are the bottleneck. Figure 9.2 illustrates this relationship graphically.

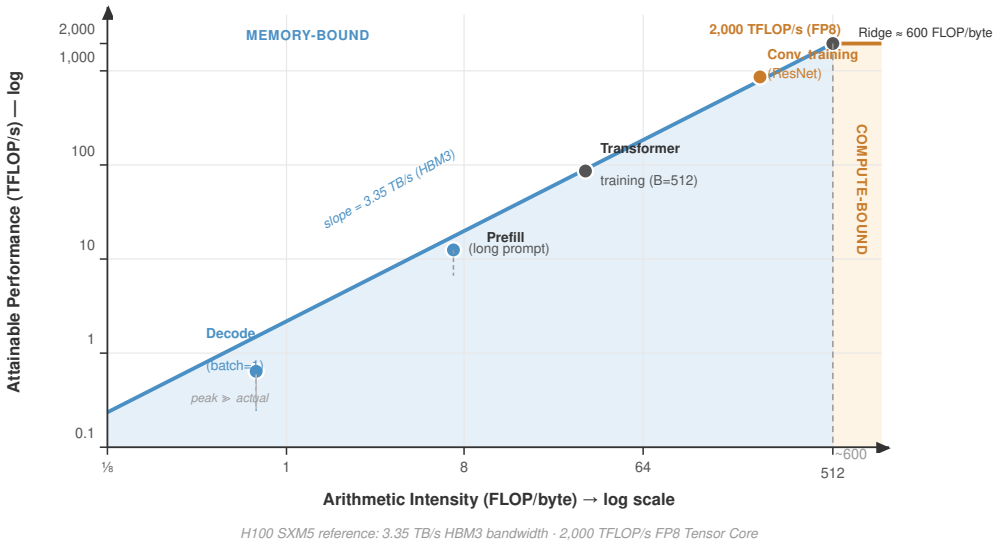


Figure 9.2: The Roofline Model: Achievable performance (y-axis) as a function of arithmetic intensity (x-axis) on a log-log plot. The sloped line represents the memory bandwidth ceiling; the flat line represents the compute ceiling at the labeled precision (FP8 tensor-core throughput on H100, approximately 1979 TFLOP/s); their intersection is the ridge point at roughly 590.7 FLOP/byte for FP8. Most transformer inference operations fall in the memory-bound region (left of the ridge point), while large batched GEMMs fall in the compute-bound region (right). The corresponding FP16 ridge point is roughly 295 FLOP/byte (989 TFLOP/s divided by 3.35 TB/s).

The ridge point of the NVIDIA H100 at FP16 precision follows directly from the same peak-compute and bandwidth values used in table 9.3:

$$I_{\text{ridge}}^{\text{H100, FP16}} = \frac{989 \text{ TFLOP/s}}{3.35 \text{ TB/s}} \approx 295 \text{ FLOP/byte}$$

Any operation performing fewer than 295.2 FLOP/byte floating-point operations per byte loaded is memory-bound on the H100 at FP16. At FP8 precision, where compute doubles to 1979 TFLOP/s and bandwidth remains 3.35 TB/s, the ridge point rises to approximately 590.7 FLOP/byte. The A100, with 312 TFLOP/s and 2.04 TB/s, has a lower ridge point of approximately 153 FLOP/byte at FP16. Across the generations in table 9.3, compute has outgrown bandwidth and the ridge point has roughly doubled (V100 to B200), so more workloads fall into the memory-bound regime over time; the per-generation movement is not monotonic, however, because each chip pairs its own compute and bandwidth (the B200 ridge sits slightly below the H100). The durable trend, not any single step, is what makes memory-efficiency techniques more valuable as hardware advances.

Figure 9.3 overlays the roofline models for four GPU generations on a single log-log plot, making the generational ridge-point shift tabulated earlier visible at a glance. An operation like naive self-attention, with an arithmetic intensity near 10 FLOP/byte, is memory-bound on every generation and falls progressively further below the ridge with each new chip. More critically, operations near 200 FLOP/byte, such as some matrix multiplications and fused blocks, can change regime as hardware changes. The same kernel can change performance regime across hardware generations, a fact that demands re-profiling whenever hardware is upgraded.

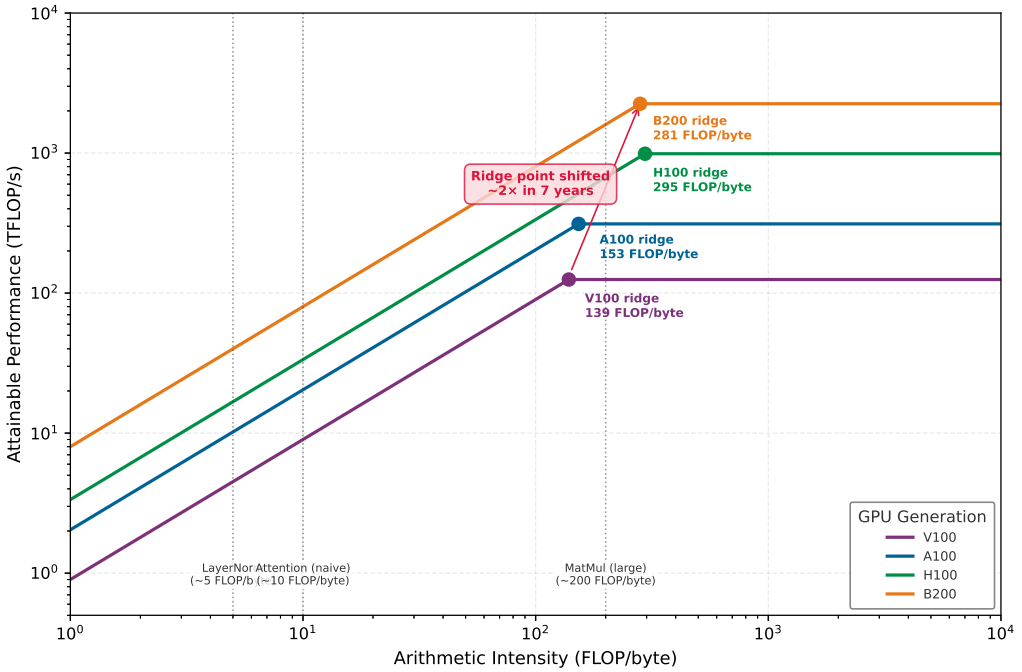


Figure 9.3: The Shifting Roofline Across GPU Generations: Overlaid FP16/BF16 roofline models for V100 through B200 show the ridge point growing from 139 to 281 FLOP/byte. Operations near 200 FLOP/byte can change regime as the roofline shifts. The same kernel can change performance regime across hardware generations.

9.1.6 Where ML workloads fall

ML operations span three orders of magnitude in arithmetic intensity, and the position of each operation on the roofline determines which optimization strategies apply.

Large general matrix multiply (GEMM) operations are the most compute-intensive operations in ML. A square matrix multiplication of dimension 4096×4096 in FP16 performs approximately 137.4 billion FLOPs while loading roughly 100.7 MB of data, yielding an arithmetic intensity of

approximately 1365.3 FLOP/byte. This sits well above the H100's ridge point, making large GEMMs firmly compute bound.

Element-wise operations tell the opposite story. A Gaussian Error Linear Unit (GELU) activation applied to a 4096×4096 tensor performs roughly 5 operations per element but must load and store each element, yielding an arithmetic intensity of approximately 1.2 FLOP/byte. The GPU spends almost all its time waiting for data transfers rather than computing, making these operations profoundly memory-bound.

Autoregressive LLM decoding at batch size one represents the extreme case. Each decoding step reads the entire weight matrix (gigabytes of data) to produce a single output token. With a hidden dimension of 4096 and batch size 1, the arithmetic intensity is approximately 1 FLOP/byte, deep in the memory-bound regime. The arithmetic intensity explains why LLM token generation achieves a tiny fraction of peak FLOP/s: the GPU spends nearly all its time reading weights, not multiplying them.

Table 9.4 reveals the central pattern behind these examples: batched GEMM can reach the compute-bound regime, but attention, element-wise work, and batch-1 decode sit below the ridge point and are governed by bytes moved rather than FLOPs advertised.

Table 9.4: Arithmetic Intensity of Common ML Operations: Many transformer inference operations, aside from large batched GEMMs, fall below the H100's ridge point and are therefore memory-bound. Performance engineering focuses on reducing the memory traffic of these operations.

Operation	Arithmetic Intensity	H100 FP16 Regime	Primary Bottleneck
GEMM (4096×4096)	~1,365 FLOP/byte	Compute-bound	Tensor core throughput
Self-Attention (seq=2048)	~50–200 FLOP/byte	Memory-bound	HBM bandwidth
Element-wise (GELU, LayerNorm)	~1–3 FLOP/byte	Memory-bound	HBM bandwidth
LLM Decode (batch=1)	~1–2 FLOP/byte	Memory-bound	HBM bandwidth

The majority of operations in a transformer inference pipeline are memory-bound. Training workloads with large batch sizes shift more operations into the compute-bound regime because GEMM dimensions scale with batch size. Inference, however, especially autoregressive generation, is dominated by memory-bound operations. Fusion, tiling, reduced precision, and algorithmic shortcuts all target the same fundamental problem: reducing bytes moved per operation.

The memory-bound nature of inference also explains a common source of confusion: GPU benchmarks reporting peak TFLOP/s often fail to predict real inference performance. Two GPUs with different TFLOP/s but identical memory bandwidth will achieve virtually identical LLM decode throughput at batch size 1, because decode is entirely memory bound. The correct metric for comparing GPUs for LLM inference is not FLOP/s but rather the combination of memory bandwidth and memory capacity. Bandwidth determines the token generation rate, and capacity determines the maximum batch size (and therefore throughput). Only at large batch sizes, where decode approaches the compute-bound regime, do the FLOP/s differences between GPUs translate into throughput differences.

9.2 Serving Regimes: Batch Size and Prefill-Decode

Diagnosis locates a workload on the roofline; the serving regime determines which knobs move it. Two structural features of LLM inference, the batch dimension and the split between prompt processing and token generation, decide whether decode stays pinned to the memory-bound slope or climbs toward the compute roof. Both are levers the rest of the chapter's techniques act through.

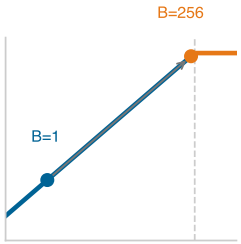
9.2.1 Batch size as the universal control knob

For memory-bound LLM serving, batch size is often the first performance lever to test, and also one of the most constrained. Increasing the batch size transforms the arithmetic intensity of every operation. For an LLM decode step, the arithmetic intensity scales linearly with batch size:

$$I_{\text{decode}}(B) = \frac{2PB}{Ps_{\text{param}} + Bd_{\text{model}}s_{\text{elem}}}$$

Here, P is parameter count, B is batch size, d_{model} is hidden width, s_{param} is bytes per stored parameter, and s_{elem} is bytes per activation element. At batch size 1, the denominator is dominated by the weight term (Ps_{param}), and $I \approx 2/s_{\text{param}} \approx 1$ FLOP/byte for FP16. At batch size 256, the weight term still dominates, but the same weight bytes are amortized across more requests, so $I \approx 2 \times 256/s_{\text{param}} \approx 256$ FLOP/byte, approaching the compute-bound regime.

At large batch sizes, the GPU transitions from memory-bound to compute-bound, and utilization increases dramatically. A single H100 achieving 5 percent utilization at batch size 1 may achieve 40 percent utilization at large batch sizes. The economic implication is stark: the cost per token decreases by roughly 8× as batching carries the workload from the memory-bound to the compute-bound regime.



Larger batches raise arithmetic intensity toward the ridge.

The constraint is memory: each additional request in the batch requires its own KV cache (the per-request store of attention keys and values for every token generated so far), and the total KV cache across all requests must fit in GPU memory alongside the model weights. The 70B-model-on-8-H100 deployment introduced here is the chapter's recurring example, revisited with full quantitative detail in the precision dividend (section 9.4.3) and the case study (section 9.10.3). A 70B model with 140 GB of weights in FP16 must be sharded across multiple GPUs; on an 8-GPU node that leaves about 62.5 GB per GPU for KV cache before overhead. The precision engineering techniques in this chapter address exactly this constraint: INT4 weight quantization reduces the per-GPU weight footprint to about 4.4 GB and frees roughly 13 GB per GPU for KV cache, enabling batch sizes that transform the economics of serving.

A serving scheduler can keep the effective batch full by adding new requests as older requests finish, but that policy only works when memory is available for the active requests. Performance engineering's role is to make that scheduler's job feasible by minimizing the per-request memory footprint, primarily through KV cache compression and weight quantization. Inference serving (Chapter 10) develops the full scheduler; the local point here is that memory optimization expands the batch sizes the scheduler can safely admit.

A critical enabler for large batch sizes is **PagedAttention**[4], vLLM's paged KV cache management technique (Kwon et al. 2023). Traditional KV cache implementations preallocate contiguous memory for each request's maximum possible sequence length.

If the maximum is 4,096 tokens but the average is 500, approximately 88 percent of the allocated memory is wasted. PagedAttention divides the KV cache into fixed-size blocks (pages), allocated on demand as the sequence grows. This eliminates memory fragmentation and enables near-100 percent utilization of the KV cache memory budget. The performance impact is indirect but substantial: by reducing memory waste, PagedAttention enables 2–4× larger effective batch sizes, which in turn improve throughput and GPU utilization through the batch size mechanism described earlier.

PagedAttention and KV cache quantization interact multiplicatively. PagedAttention reduces memory *waste* (from fragmentation), while quantization reduces memory *usage* (from precision). Together, they increase the effective batch size by 8–16× compared to a baseline system with preallocated FP16 KV caches, fundamentally changing the economics of LLM serving.

4 **PagedAttention**: Named by direct analogy to OS virtual memory paging, where the OS maps noncontiguous physical pages to contiguous virtual addresses. The vLLM paper presented this insight at SOSP 2023: the same mechanism eliminates internal fragmentation in KV caches, recovering the 60–80 percent of GPU memory wasted by worst-case preallocation (Kwon et al. 2023). This single abstraction transformed LLM serving economics by enabling 2–4× larger batch sizes without any change to model weights or precision.

9.2.2 The prefill-decode decomposition

Modern LLM serving systems decompose each request into two distinct phases with fundamentally different performance characteristics. The distinction between these phases drives system architecture and optimization strategy.

The **prefill phase** processes the entire input prompt in parallel. If the prompt contains S tokens, the prefill phase executes a single forward pass over all S tokens simultaneously. The GEMM operations have shape $[S, d_{\text{model}}] \times [d_{\text{model}}, d_{\text{model}}]$, making the batch dimension equal to S. For a prompt of 1024 tokens, this is arithmetically intensive: the arithmetic intensity is approximately $2 \times 1024/2 = 1024$ FLOP/byte for FP16 weights, well into the compute-bound regime. Prefill is therefore limited by Tensor Core throughput, not memory bandwidth.

The **decode phase** generates output tokens one at a time, autoregressively. Each step has a batch dimension of 1 (for a single request) or the number of concurrent requests (for batched serving). At batch size 1, decode is deeply memory-bound as analyzed in section 9.1.6.

The prefill-decode decomposition has direct implications for system design. A system optimized for prefill (maximizing FLOP/s utilization) would use large matrix sizes and high compute throughput. A system optimized for decode (maximizing bandwidth utilization) would use aggressive quantization and memory optimization. A real serving system must handle both phases, often simultaneously across different active requests.

Disaggregated serving addresses this mismatch by running prefill and decode on separate hardware pools. Here, disaggregation is evidence that prefill and decode have different bottlenecks; Chapter 10 develops the routing, admission-control, and serving-policy machinery. Prefill servers are optimized for compute (fewer, higher-FLOP/s GPUs), while decode servers are optimized for memory bandwidth and capacity (more memory per GPU, aggressive quantization). The KV cache computed during prefill is transferred to a decode server, which handles the subsequent autoregressive generation. This disaggregation allows each phase to use hardware and software configurations tuned for its specific bottleneck.

The performance characteristics of each phase determine which optimization techniques apply. FlashAttention provides its largest gains during prefill, where the quadratic attention computation dominates. KV cache quantization and speculative decoding apply exclusively to the decode phase. Precision engineering (FP8/INT4 weights) benefits both phases, but through different mechanisms: prefill benefits from doubled compute throughput (FP8 Tensor Cores), while decode benefits from doubled effective bandwidth (half the bytes per weight read).

A quick decode calculation shows why the memory wall dominates even when the accelerator has abundant unused FLOP/s.



Napkin Math 9.1: The Roofline diagnostic

Problem: A 70B parameter LLM is deployed on 8× H100 GPUs with tensor parallelism. At batch size 1, each GPU holds approximately 8.75B parameters in FP16 (17.5 GB of weights). Each decode step reads its weight shard to produce one token. What is the achieved arithmetic intensity, and what is the theoretical maximum token generation rate?

Math:

Step 1: Arithmetic intensity. Each decode step per GPU: FLOPs $= 2 \times 8.75 \times 10^9 = 17.5$B FLOPs. Bytes loaded $= 8.75 \times 10^9 \times 2$ bytes $= 17.5$ GB.

$I = \frac{1.75 \times 10^{10} \text{ FLOP}}{1.75 \times 10^{10} \text{ bytes}} = 1$ FLOP/byte

At 1 FLOP/byte, the operation sits far below the H100 ridge point of ~295 FLOP/byte: deeply memory-bound.

Step 2: Token rate. Since the operation is memory-bound, performance is limited by bandwidth, not compute:

$t_{\text{decode}} = \frac{17.5\text{GB}}{3.35\,\text{TB/s}} \approx 5.2\,\text{ms}$ per token

This yields approximately 191.4 tokens/s per GPU, or about 191.4 tokens/s for the model (since tensor parallelism does not multiply throughput for memory-bound decode). In practice, overheads from KV cache reads and NVLink synchronization reduce this substantially below the ideal bandwidth-only limit.

Systems insight: At batch size 1, only about 0.3 percent of the H100's FP16 FLOP/s are in use. The improvement paths all attack the same weight-streaming cost: larger batches amortize each weight read across more requests, quantization reduces the bytes read for each weight, and speculative decoding tries to obtain multiple accepted tokens from one target-model weight pass.

The roofline model establishes the physics that constrains all subsequent optimization. The first and most impactful strategy for breaking through the memory wall is keeping data in SRAM instead of round-tripping through HBM.

9.3 Operator Fusion and Kernel Engineering

Consider the simple sequence of operations $Y = \text{LayerNorm}(\text{GELU}(XW + b))$. In a naive implementation, the GPU writes the output of the matrix multiply back to main memory, reads it back for the GELU, writes it out again, and reads it one final time for the LayerNorm. This redundant data movement shatters performance. **Operator fusion** eliminates these intermediate round-trips by keeping results in ultra-fast registers, executing the entire sequence in a single trip to memory.

> Systems Perspective 9.2: Analogy: The short-order cook
>
> Imagine a kitchen where one chef chops vegetables, puts them in the fridge (HBM), then another chef takes them out to boil them, puts them back in the fridge, and a third chef takes them out to plate them. This is unfused execution: the bottleneck is not the cooking but the constant walking to the fridge.
>
> Operator fusion assigns the recipe to a single chef who keeps the ingredients on their cutting board (SRAM/Registers) and performs all three steps consecutively without ever returning to the fridge until the final dish is ready.

9.3.1 The kernel launch problem

Each GPU kernel launch involves overhead: the CPU must prepare launch parameters, dispatch to the GPU command queue, and the GPU must schedule thread blocks across its streaming multiprocessors (SMs). For a small element-wise operation on an accelerator such as an H100, this overhead can be 5–20 μs, a time during which a memory-bound kernel might have already completed its useful work. When a transformer layer comprises dozens of small operations (add, multiply, normalize, activate), the cumulative launch overhead becomes significant.

Each unfused kernel must also materialize its output in HBM. Consider a sequence of three operations: $Y = \text{LayerNorm}(\text{GELU}(XW + b))$. Without fusion, this requires three passes through HBM:

1. **GEMM kernel**: Read X and W from HBM, compute $XW + b$, write result Z_1 to HBM.
2. **GELU kernel**: Read Z_1 from HBM, compute $\text{GELU}(Z_1)$, write Z_2 to HBM.
3. **LayerNorm kernel**: Read Z_2 from HBM, compute $\text{LayerNorm}(Z_2)$, write Y to HBM.

Intermediate tensors Z_1 and Z_2 each occupy the same memory as the output Y. For a hidden dimension of 4096 and batch size of 2048 in FP16, each intermediate tensor is 16.8 MB. The unfused execution materializes 33.6 MB of intermediate tensors and performs 67.1 MB of intermediate HBM traffic, writing and then rereading each tensor. A fused kernel avoids that traffic entirely by holding Z_1 and Z_2 in registers or shared memory (SRAM) within the SM. Figure 9.4 contrasts these two execution paths, making the HBM traffic savings visible.

As figure 9.4 makes concrete, fusion reduces HBM round-trips from six to two per layer, a roughly 3× reduction in off-chip memory traffic for this layer block. In a naive implementation without operator fusion, executing one transformer layer requires roughly 50 separate kernel launches. If each launch incurs a 10-microsecond overhead, the system spends 500 microseconds purely on dispatch latency. If the actual arithmetic execution of the layer takes only 2 milliseconds, the launch overhead consumes 20 percent of the total wall-clock time, leaving the GPU compute units idle for one-fifth of the inference cycle. This "launch-bound" regime limits the benefits of faster hardware; doubling the GPU's FLOP/s does nothing to reduce the 500-microsecond fixed cost. Operator fusion addresses this by compiling these 50 discrete operations into a small handful of fused kernels, often reducing the count to 5–10 launches, thereby reclaiming the lost cycles and shifting the workload away from dispatch overhead and back toward the hardware limits captured by the roofline model.

9.3.2 Fusion categories

Fusion is profitable when the HBM traffic it removes is worth the kernel complexity it introduces. The three common categories differ by that trade-off: how much data movement they eliminate, how much synchronization they require, and how often a compiler can apply them automatically.

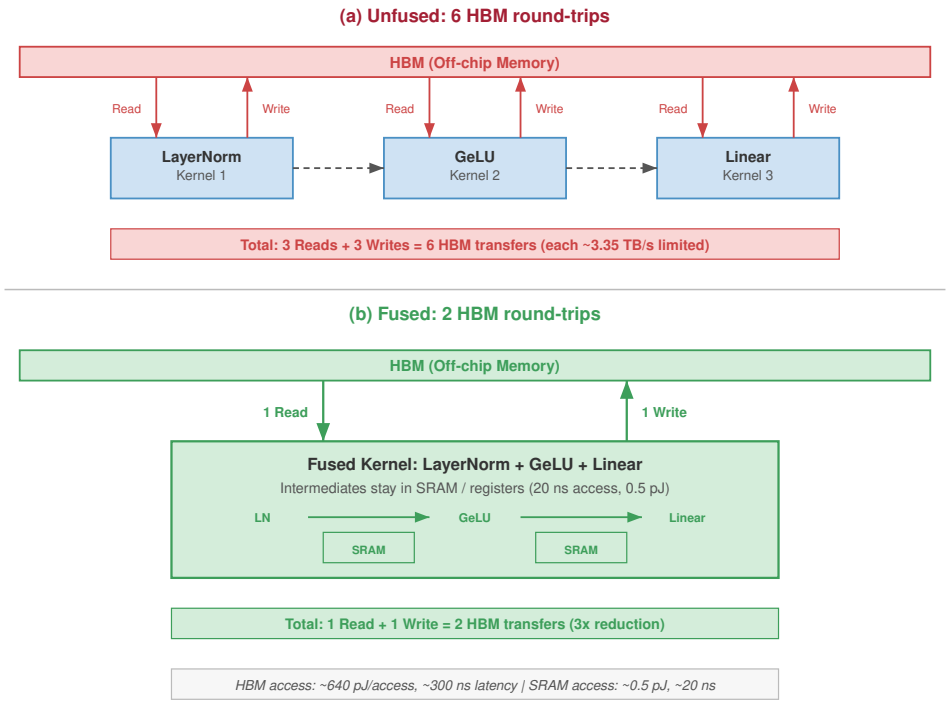


Figure 9.4: Operator Fusion: Before and After: Top: three separate kernels each read from and write to HBM, materializing intermediates Z_1 and Z_2, with six HBM round-trips along the unfused path. Bottom: a single fused kernel reads inputs once, holds intermediates in SRAM, and writes only the final output, leaving just two HBM round-trips. The fused path delivers roughly a $3\times$ reduction in HBM traffic per layer.

Element-wise fusion sits at the low-complexity end: consecutive element-wise operations (add, multiply, activation functions) combine into a single kernel. Because each output element depends on exactly one input element, this fusion is always legal and straightforward to implement. Deep learning frameworks commonly perform element-wise fusion automatically for supported patterns.

When the sequence includes a reduction, the fusion decision becomes more constrained. Reduction fusion combines an element-wise operation with a subsequent reduction (such as summing elements for a loss function, or computing mean and variance for layer normalization). Reductions require inter-thread communication within the kernel, using warp-level shuffle instructions or shared memory to aggregate partial results across threads. Despite this complexity, the memory savings are substantial: the intermediate tensor before the reduction never materializes in HBM. For layer normalization specifically, reduction fusion avoids writing the large prenormalization tensor to HBM and reading it back for the mean/variance computation.

The highest-payoff case is operator-specific fusion. These are custom kernels designed for a specific sequence of operations, such as fused attention or fused GEMM-bias-activation. The kernel architect must reason about data flow, shared memory allocation, and thread scheduling simultaneously. The payoff is substantial: FlashAttention, which we examine next, removes the quadratic attention workspace and keeps the online softmax state linear in sequence length.

To appreciate the quantitative impact, consider each category applied to a single transformer layer with hidden dimension 4096 and batch size 2048 in FP16. Element-wise fusion of a bias-GELU-dropout chain in this shape eliminates two intermediate tensors of 16.8 MB each, saving 67.1 MB of HBM traffic per layer. Across 80 layers, this reclaims about 5.4 GB of HBM traffic per forward pass. Reduction fusion of LayerNorm avoids materializing large prenormalization tensors and intermediate statistics. Operator-specific attention fusion (FlashAttention) provides the largest single gain by removing the quadratic score and probability matrices that dominate long-context

attention. The cumulative effect of all three fusion categories can remove a large fraction of HBM traffic for a memory-bound transformer forward pass, but the end-to-end speedup must still be verified with profiling because the bottleneck may shift.

9.3.3 CUDA graphs: Eliminating launch overhead

An orthogonal technique for reducing the overhead term in the iron law is **CUDA Graphs**[5]. While operator fusion combines multiple operations into fewer kernels, CUDA Graphs eliminate the CPU overhead of launching those kernels.

In standard PyTorch execution, each kernel launch requires the CPU to push a command to the GPU's command queue. For a transformer decoder layer with 30+ kernels, this CPU-to-GPU roundtrip (typically 5–10 μs per launch) accumulates to 150–300 μs per layer. For a 70-layer model, kernel launch overhead alone contributes 10–20 ms per forward pass, a significant fraction of the total time for memory-bound inference.

CUDA Graphs address this by recording a sequence of GPU operations (kernel launches, memory copies) into a replayable graph. The recording happens once during a warmup phase. On subsequent iterations, replaying the graph requires only a single CPU-to-GPU command that dispatches the entire recorded sequence, reducing launch overhead to approximately 5–10 μs total regardless of the number of kernels.

The benefit is substantial: for a model with 30+ kernels per layer and 70+ layers, the baseline kernel launch overhead can exceed 15 ms per forward pass. CUDA Graphs reduce this to under 0.1 ms, reclaiming 15 ms that translates directly to higher token generation rates.

The constraint is that CUDA Graphs require deterministic execution: the sequence of operations, tensor shapes, and memory addresses must be identical across replays. This conflicts with dynamic inference patterns like variable-length sequences, changing batch sizes, and conditional computation (early exit, MoE routing). In practice, CUDA Graphs are most effective for the decode phase of LLM serving, where the computation pattern is repetitive (same operations per token), and less useful for the prefill phase, where input lengths vary.

The combination of operator fusion (reducing the number of kernels) and CUDA Graphs (reducing the per-kernel overhead) can together eliminate nearly all noncompute overhead from the forward pass. When profiling reveals that kernel launch gaps constitute more than 10 percent of execution time, CUDA Graphs should be the first intervention considered.

5 **CUDA Graphs**: Introduced in CUDA 10 (2018), originally for graphics rendering pipelines that replay identical command sequences every frame. The strict determinism requirement – identical operations, shapes, and memory addresses per replay – directly conflicts with the dynamic shapes and variable batch sizes of LLM serving, restricting their use primarily to the decode phase where the computation pattern repeats per token.

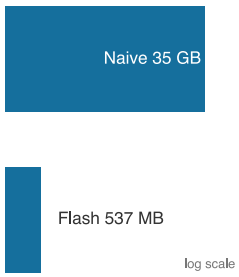
9.3.4 FlashAttention: Tiled attention as a system primitive

Standard self-attention computes Softmax$(QK^T/\sqrt{d_k})V$, where Q, K, and V are matrices of shape [sequence length$\times$head dimension]. The na"ive implementation materializes the full $S\times S$ attention matrix $A = QK^T$ in HBM, where S is the sequence length. For $S = 8192$ and FP16 precision, this matrix alone consumes $8192 \times 8192 \times 2 \approx 134$ MB per attention head, so the quadratic score tensors run to several gigabytes per layer. The canonical accounting below makes this precise for a 64-head Llama-style layer, counting the score and probability traffic alongside the persistent Q, K, V, and output tensors.

FlashAttention (Dao et al. 2022) reformulates attention using tiling. Instead of materializing the full $S\times S$ attention matrix, it processes Q, K, and V in small blocks that fit in on-chip SRAM. The algorithm loads tiles of Q, K, and V, computes partial attention scores, and maintains running statistics (online softmax) to produce the exact result without ever storing the full attention matrix in HBM.

The reduction in materialized attention state is dramatic. For a sequence length of 8192, 64 heads, and head dimension 128 in FP16, the naive implementation materializes and revisits approximately 34.9 GB of HBM-resident tensors for the full layer, or 545.3 MB per head. FlashAttention avoids the quadratic score and probability tensors; its persistent HBM-visible tensors are dominated by Q, K, V, and output Y, totaling approximately 536.9 MB for the full layer, or 8.4 MB per head. This simplified accounting excludes schedule-dependent tile reloads inside a particular kernel, but it captures the important scaling result: a 65× reduction in HBM-visible attention state for this configuration.

The key insight behind FlashAttention is the **online softmax**[6] trick, which makes tiling possible for an operation that appears to require global information. Standard softmax computes softmax$(s_i) =$



FlashAttention shrinks attention memory about 65 times.

6 **Online Softmax**: *Online* here is an algorithmic term meaning the computation processes data incrementally in a single pass without storing the full input – the same sense as in *online learning* or *online algorithms*. This property is what makes tiling possible: the algorithm never needs the complete $S\times S$ score matrix in memory simultaneously, reducing attention memory from $\mathcal{O}(S^2)$ to $\mathcal{O}(S)$ and making long-context inference feasible on fixed-size SRAM.

$e^{s_i}/\sum_j e^{s_j}$, but for numerical stability it first subtracts the global maximum: $\text{softmax}(s_i) = e^{s_i - m}/\sum_j e^{s_j - m}$ where $m = \max_j s_j$. Finding this global maximum seems to require seeing all scores first, which would force materializing the full $S \times S$ matrix.

The online algorithm avoids this by maintaining running statistics that are updated incrementally as each tile is processed. When processing tile t, the algorithm executes four steps:

1. Computes a local block of scores $A_t = Q_{\text{block}} K_t^T$.
2. Updates the running maximum: $m_{\text{new}} = \max(m_{\text{old}}, \max(A_t))$.
3. Rescales the previous running sum and output: multiply by $e^{m_{\text{old}} - m_{\text{new}}}$ to correct for the updated maximum.
4. Computes the local softmax contribution using m_{new} and accumulates into the running output.

After processing all tiles, the running output contains the mathematically equivalent result to the standard algorithm up to the floating-point roundoff of the chosen precision. The rescaling step (step 3) is the critical innovation: it allows the algorithm to "fix up" previous partial results when a new tile reveals a larger maximum value. The algorithm is exact in the mathematical sense, but floating-point execution order can still change the last few bits relative to an unfused implementation.

The cost of this tiling is additional arithmetic: the rescaling operations in step 3 add FLOPs that the standard algorithm does not perform. Because the operation is profoundly memory-bound (standard attention's arithmetic intensity falls to roughly 1–10 FLOP/byte once its materialized score and probability tensors are counted against the HBM traffic they generate, below the higher regime-dependent figures in table 9.4), the additional compute is "free" in the sense that the GPU's arithmetic units would otherwise be idle, waiting for HBM data transfers. *Trading extra compute for fewer memory accesses is profitable whenever the operation is memory-bound*, the central principle of this entire chapter.

The mechanism that delivers the materialized-state reduction quantified above is tiling. FlashAttention processes the computation in tiles (typically 128×128 on H100). For one tile, the algorithm loads a block of Q ($128 \times 128 \times 2 \approx 32.8$ KB), a block of K ($128 \times 128 \times 2 \approx 32.8$ KB), and a block of V (32.8 KB), totaling approximately 98.3 KB. This fits comfortably in the H100's 228 KB of shared memory per SM. The tile score $A_{\text{tile}} = Q_{\text{tile}} K_{\text{tile}}^T$ is computed and consumed entirely within SRAM; it is never written to HBM. The algorithm iterates over $8192/128 = 64$ column tiles for each of 64 row tiles, reloading tiles as required by the kernel schedule. The important point is that the quadratic $S \times S$ score and probability matrices are never materialized: the persistent per-head tensors are Q, K, V, Y, and $\mathcal{O}(S)$ softmax statistics, instead of the hundreds of megabytes of quadratic intermediates the canonical accounting above counted.

FlashAttention-2 (Dao 2023) further optimizes the algorithm for GPU architectures with many streaming multiprocessors by restructuring the parallelism pattern. The original FlashAttention parallelizes over batch and head dimensions, meaning each thread block handles one (batch, head) pair and iterates over the full sequence. FlashAttention-2 additionally parallelizes over the sequence dimension of the query matrix, distributing work across thread blocks more efficiently and achieving better occupancy. It also reduces the number of non-GEMM FLOPs by restructuring the rescaling operations and exploiting the asymmetry between the Q loop (outer) and K/V loop (inner).

Newer FlashAttention-family kernels target Hopper-era hardware features such as FP8 Tensor Cores and the Tensor Memory Accelerator (TMA), a hardware path for asynchronous bulk tensor movement between HBM and shared memory. The systems principle is the same as the original algorithm: as the hardware exposes faster movement and lower-precision execution paths, the attention schedule must be rewritten to use those paths without materializing the quadratic workspace.

The original breakthrough was a change in how engineers understood the bottleneck: the constraint on attention was the memory hierarchy, not the matrix multiplication.

Example 9.1: The FlashAttention breakthrough

Context: In 2022, Tri Dao and coauthors introduced FlashAttention (Dao et al. 2022), showing that the attention mechanism's performance bottleneck could be treated as an IO-aware data-movement problem rather than only as a matrix multiplication tuning problem.

Mechanism: The bottleneck was memory hierarchy, not compute. Standard attention materialized the massive $S \times S$ attention score matrix in high-latency HBM. FlashAttention restructured the algorithm using tiling to keep running statistics in on-chip SRAM, computing the softmax without ever writing the full matrix to global memory. This reduced memory complexity to linear $\mathcal{O}(S)$ and wall-clock time by 2–4$\times$ on relevant attention workloads.

Systems lesson: FlashAttention matters because it changes where the attention state lives. The durable optimization is not "a faster kernel" but a data-movement reduction that keeps quadratic intermediates out of HBM.

Before turning to the scaling plot, pause on the mechanism: FlashAttention trades a small amount of extra arithmetic for the elimination of quadratic HBM-resident state.

Checkpoint 9.2: FlashAttention mechanics

Verify your understanding of memory-aware attention:

- ☐ **Sequence-length scaling**: Explain why FlashAttention provides a larger speedup for *longer sequences* (for example, 32K) than for short sequences (for example, 512).
- ☐ **Online softmax storage**: Identify whether the "online softmax" statistics in the FlashAttention algorithm reside in HBM, L2 Cache, or SM SRAM.
- ☐ **FLOPs unchanged**: Evaluate the claim that FlashAttention reduces the total number of floating-point operations (FLOPs) required for attention.
- ☐ **Memory to batch size**: Explain how reducing the attention memory from $\mathcal{O}(S^2)$ to $\mathcal{O}(S)$ enables larger batch sizes for serving.

The 65× figure above is the canonical, full-layer materialized-state reduction; it is the number to carry forward. Figure 9.5 adds only the *scaling shape*: because standard-attention workspace grows quadratically while FlashAttention's running state grows linearly, the advantage widens with sequence length. The plot's larger ratios reflect a narrower accounting boundary, comparing only the quadratic score workspace against the linear running statistics rather than the full set of HBM-visible tensors, so they are not directly comparable to the 65× figure; the takeaway is that the gap grows, not its exact value at a given length.

FlashAttention reduces the memory wall within a single GPU by tiling across the SRAM-HBM boundary. For sequence lengths that exceed the memory capacity of a single GPU, the same tiling principle extends across multiple GPUs via Ring Attention (Liu et al. 2023). By distributing the sequence blocks across a ring of accelerators and overlapping communication with computation, Ring Attention enables context windows that would be impractical on single-GPU configurations. Section 5.5.2.3 examines the distributed mechanics of Ring Attention within the broader tensor-parallelism discussion.

9.4 Precision Engineering

Fusion reduces the number of trips through HBM; precision engineering reduces the payload of each trip. Moving FP16 weights through HBM consumes twice as many bytes as FP8. For bandwidth-bound kernels, shrinking weights from 2 bytes to 1 byte can roughly halve weight-read traffic and increase effective memory bandwidth. The engineering decision is where numerical noise can be tolerated to reduce bandwidth pressure and where quality demands higher precision. While section 5.6 examines 8-bit floating point (FP8) as a training-time primitive, here we focus on the quantization techniques that enable efficient inference at scale.

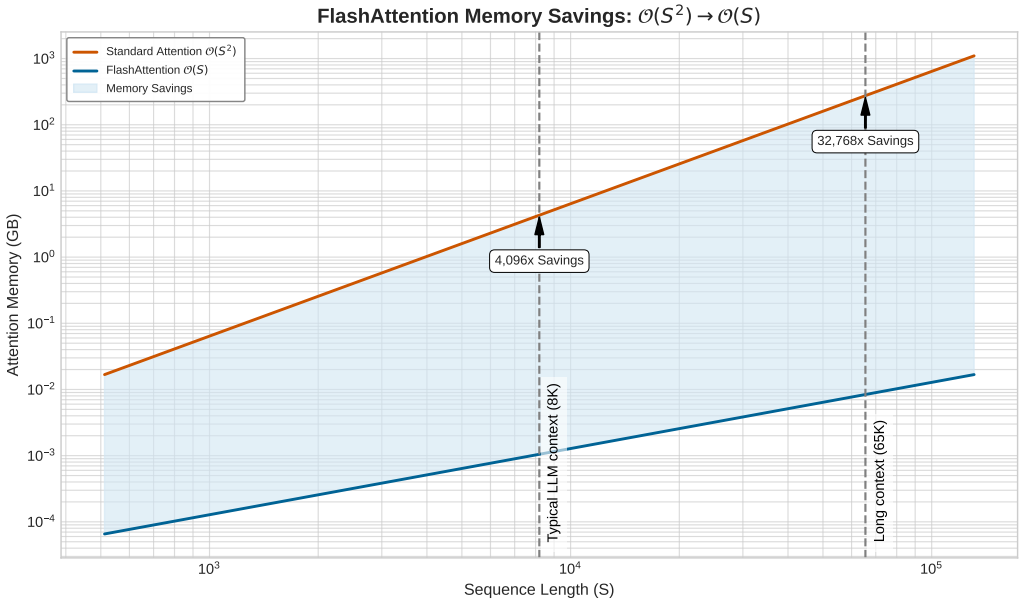


Figure 9.5: FlashAttention Memory Savings: $\mathcal{O}(S^2) \rightarrow \mathcal{O}(S)$. Standard attention allocates the full $S \times S$ score matrix in HBM, while FlashAttention maintains only $\mathcal{O}(S)$ running statistics. The shaded region represents memory freed for KV cache, activations, or larger batch sizes. This workspace-only comparison uses a narrower boundary than the full-layer materialized-state accounting in the text; its point is the widening shape of the gap as context grows, not a headline ratio.

9.4.1 Block-wise quantization

The first inference-side precision decision is how to shrink weights while protecting the channels that carry rare but essential signal. Post-training quantization to INT8 or INT4 delivers even greater bandwidth savings for inference, but LLMs present a unique challenge: **outlier features**[7]. Dettmers et al. (2022) discovered that large language models develop a small number of hidden dimensions (about 0.1 percent of features) with activation magnitudes roughly 3–20× larger than the rest. Applying uniform per-tensor INT8 quantization clips these outliers, destroying the information they carry, or expands the quantization range to accommodate them, wasting precision on the majority of near-zero values.

7 | **Outlier Features**: Large-scale transformers develop emergent "outlier" dimensions with activation magnitudes up to about 20× larger than typical values (Dettmers et al. 2022). While these outliers constitute about 0.1 percent of all features, clipping them during INT8 quantization destroys the model's reasoning capabilities. This physical property of large models is the reason post-training quantization (PTQ) requires "outlier-aware" strategies like LLM.int8() or Activation-Aware Weight Quantization (AWQ).

Definition 9.2: Block-wise quantization

Block-wise Quantization is an ML quantization scheme that partitions a weight tensor into nonoverlapping groups of G_{block} elements and computes a per-group scale $s_i = (x_{\max,i} - x_{\min,i})/(2^b - 1)$, bounding worst-case quantization error within each group independently.

1. **Significance**: With block size $G_{\text{block}} = 64$ and $b = 4$ bits, weights compress from 16 bits to 4 bits (a 4× memory reduction) while each FP16 scale adds $16/64 = 0.25$ bits per weight. This yields an effective bit-width of 4.25 bits per weight: 6.25 percent overhead relative to the INT4 payload, or about 1.6 percent of the original FP16 weight size.
2. **Distinction**: Unlike Per-Tensor Quantization, which applies a single scale across the entire weight matrix and forces that scale to accommodate outlier values at the cost of wasting precision on the majority of near-zero weights, Block-wise Quantization contains outlier damage within individual blocks, preventing a single extreme value from degrading quantization fidelity for the whole tensor.
3. **Common pitfall**: A frequent misconception is that block size is a free hyperparameter. Smaller blocks ($G_{\text{block}} = 32$) reduce quantization error but double the metadata overhead

vs. $G_{\text{block}} = 64$. At $G_{\text{block}} = 16$, the scale adds 1 bit per weight: 25 percent overhead relative to the INT4 payload, or 6.25 percent of the original FP16 weight size.

The deployment choice is where to pay for outlier protection. Each widely used method protects the same sensitive information, but it moves the cost to a different place in the serving pipeline.

LLM.int8() keeps the outlier cost at runtime by decomposing each matrix multiplication into two parts: a small set of outlier dimensions processed in FP16, and the remaining dimensions processed in INT8. The system identifies outlier dimensions at runtime (those exceeding a magnitude threshold, typically 6.0), routes them to an FP16 GEMM, and routes the remaining dimensions to an INT8 GEMM. The results are combined to produce the final output. This achieves nearly lossless INT8 inference for models that would otherwise degrade substantially under uniform quantization.

GPTQ (Frantar et al. 2023) moves the cost into calibration through weight-only quantization using second-order information. Instead of quantizing each weight independently, GPTQ performs a layer-wise reconstruction pass that uses an approximate Hessian inverse from calibration activations to estimate which quantization errors matter most, then compensates for those errors in the remaining unquantized weights. This produces INT4 weight representations with low accuracy loss for many transformer models. The key insight is that quantization error in one weight can sometimes be offset through correlated weights in the same layer.

AWQ [Activation-Aware Weight Quantization; Lin et al. (2023)] reduces the calibration burden by observing that not all weights are equally important: weights connected to high-activation channels contribute disproportionately to model output. AWQ identifies these salient weights by analyzing activation magnitudes across a calibration dataset, then applies per-channel scaling to protect them before uniform group quantization. This achieves INT4 weight quantization with quality competitive with reconstruction-based PTQ methods while avoiding GPTQ-style Hessian reconstruction.

SmoothQuant (Xiao et al. 2023) shifts the outlier burden from activations to weights before inference. Rather than handling outliers at runtime (LLM.int8()) or through weight optimization (GPTQ, AWQ), SmoothQuant smooths the activation distribution before quantization by migrating the quantization difficulty from activations to weights. The key observation is that activation outliers are channel-specific: certain hidden dimensions consistently produce large values across all tokens. SmoothQuant applies a per-channel scaling transformation that divides the activation by a smoothing factor and multiplies the corresponding weight by the same factor. This mathematically equivalent transformation reduces activation outlier magnitudes at the cost of slightly increasing weight magnitudes, making both tensors more amenable to uniform INT8 quantization. The result is efficient W8A8 (weight-8-bit, activation-8-bit) quantization that exploits INT8 Tensor Cores for both bandwidth and compute benefits.

The practical choice depends on the binding resource and the quality budget. LLM.int8() handles outliers at runtime with mixed-precision decomposition but limits compression to INT8. GPTQ uses second-order information for aggressive INT4 weight compression but requires hours of calibration per model. AWQ reaches similar INT4 quality with minutes of calibration by focusing on activation-aware scaling. SmoothQuant enables W8A8 quantization by preprocessing the weight-activation pairs. For weight-only LLM serving, AWQ is often attractive when calibration time matters; for workloads that need activation quantization and INT8 Tensor Cores, SmoothQuant is the more relevant option.

The choice among these techniques also depends on the deployment target. For GPU inference with Tensor Core support, GPTQ and AWQ produce INT4 weight representations that are dequantized to FP16 during the GEMM computation, using the GPU's FP16 Tensor Cores. For CPU inference or edge deployment, INT8 representations (LLM.int8() or static per-channel INT8 quantization) can directly exploit integer arithmetic units without dequantization overhead.

The storage cost for block-wise quantization is minimal. Storing one FP32 scale (32 bits) for every block of 128 INT8 weights (1024 bits) increases total model size by only 3 percent. This small overhead allows block-wise quantization to isolate the destructive impact of outliers, preserving the effective dynamic range for the 99 percent of normal weights, without the bandwidth penalty of higher-precision formats. At the extreme end of the deployment spectrum, quantization moves from an optimization to a physical necessity. A mobile or federated deployment provides that limiting

case: when the device has only kilobytes or megabytes of memory, precision is no longer a tuning knob after the model is chosen; it is part of the feasibility test.

Lighthouse 9.1: Archetype C (Federated MobileNet): TinyML survival

For Archetype C (Federated MobileNet), quantization is *a prerequisite for survival*, not *an optimization*. On a microcontroller with only 512 KB of SRAM, an FP16 model is physically impossible to load. Binary neural networks (BNNs) and 1-bit quantization push this to the extreme, representing weights as single bits (+1/-1). While this trades significant accuracy, it reduces the weight memory footprint by 16× relative to FP16 (32× relative to FP32) and the energy per operation by up to 100×, enabling intelligence on devices that operate on harvested energy (microwatts).

9.4.2 Post-training vs. quantization-aware training

When a post-training recipe misses the quality budget, the precision decision shifts from calibration to training cost. The trade-off between **Post-Training Quantization (PTQ)** and **Quantization-Aware Training (QAT)** centers on the balance between engineering agility and model fidelity. For a model like Llama-2-70B, PTQ is a common first choice for immediate deployment. Techniques like GPTQ or AWQ process the model layer-by-layer using a small calibration dataset (typically 128–1024 samples) to minimize reconstruction error. This process is computationally cheap, often requiring hours rather than a distributed training run for a 70B model. While PTQ is usually robust at INT8, aggressive quantization to INT4 or INT3 can incur a visible penalty: perplexity may degrade, and reasoning benchmarks such as Massive Multitask Language Understanding (MMLU) can drop several percentage points if the quantization recipe does not protect sensitive layers and outlier channels.

When PTQ fails to meet quality thresholds, QAT provides the remedy by integrating quantization noise directly into the training loop. By simulating low-precision rounding during the forward pass and approximating gradients during the backward pass via the straight-through estimator[8] (STE), the network learns to adjust its weights to be robust to quantization.

The cost is substantial: QAT is effectively a full fine-tuning run, often requiring hundreds of GPU-hours and a distributed training cluster. For a 70B model, this can mean a multi-day multi-GPU job instead of an hours-scale PTQ calibration run. Adapter-based quantized fine-tuning narrows the gap by freezing the low-precision base model and updating only small low-rank adapter matrices rather than all model weights. This hybrid approach offers some of the quality recovery of QAT with a memory footprint small enough to run on much smaller hardware, but it is a task-adaptation remedy rather than a universal replacement for full quantization-aware retraining.

In many deployment environments, a practical workflow follows a two-stage approach: deploy with PTQ first when it meets quality requirements, then apply QAT or adapter-based quantized fine-tuning if the PTQ model fails at the target precision. This sequence minimizes engineering effort while preserving the option of higher quality when needed.

9.4.3 Weight-only vs. weight-activation quantization

The final precision choice is whether the optimization should stop at the weights or include activations as well. Weight-only quantization (GPTQ, AWQ) reduces weight precision to INT4 or INT3 while keeping activations in FP16. During a GEMM, the INT4 weights are dequantized to FP16 on-the-fly, and the computation proceeds using FP16 Tensor Cores. The benefit is reduced memory for weight storage and reduced HBM bandwidth for weight reads, but the GEMM itself still operates at FP16 precision. This approach is ideal for memory-bound inference (batch size 1 decode), where the bottleneck is reading weights from HBM.

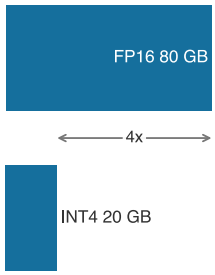
Weight-activation quantization (SmoothQuant, FP8 training) reduces both weights and activations to lower precision, enabling the GEMM to execute using lower-precision arithmetic (INT8 Tensor Cores, FP8 Tensor Cores). This provides both bandwidth and compute benefits but is more challenging to implement without quality degradation, because activation distributions are more dynamic and harder to quantize than weight distributions.

8 **Straight-Through Estimator (STE)**: Discussed by Bengio et al. (2013) for hard stochastic neurons, the STE handles a fundamental calculus problem that also appears in quantization: the gradient of a rounding function is zero almost everywhere, making backpropagation through quantized layers impossible by standard rules. The STE passes the upstream gradient through the rounding step as a surrogate gradient, pretending the rounding did not happen. This approximation is useful in practice for training networks with discrete or quantized operations, but it is a heuristic rather than a general convergence guarantee for every QAT setup.

The choice depends on the operational regime. For memory-bound inference (small batch sizes), weight-only INT4 quantization often provides the largest speedup per unit of quality degradation. For compute-bound inference (large batch sizes) or training, weight-activation FP8 quantization provides throughput gains that weight-only quantization cannot match. High-performance serving systems often use different quantization strategies for different operating points: INT4 weight-only at low batch sizes (for latency) and FP8 weight-activation at high batch sizes (for throughput).

The same precision problem extends to the key-value (KV) cache, which determines whether weight savings become larger batches. In decode, weights may be compressed aggressively while per-request cache state still grows with sequence length, so a precision change that leaves the cache untouched may fail to move the serving bottleneck. Numerical compression reduces the bytes stored per cached key or value, while grouped query attention (GQA) reduces how many key-value heads must be cached for each layer. Both mechanisms matter because the scheduler admits requests against the combined memory footprint of weights plus active cache state. The calculation below isolates the local capacity effect before Chapter 10 returns to full serving policy.

A capacity calculation makes the serving impact of precision choices concrete.



Lower precision frees memory, enlarging the feasible batch.

Napkin Math 9.2: The precision dividend

Problem: A 70B parameter model is served on 8× H100 GPUs. The model weights in FP16 consume 140 GB (17.5 GB per GPU). KV cache at FP16 consumes 1.34 GB per request. How does quantizing weights to INT4 and KV cache to INT8 change the maximum batch size?

Before optimization (all FP16):

- Weights: 17.5 GB/GPU
- Available for KV cache: 80 GB - 17.5 GB = 62.5 GB/GPU
- KV cache per request: 1.34 GB total, or approximately 0.17 GB/GPU
- Maximum batch size: approximately 372 requests

After optimization (INT4 weights, INT8 KV cache):

- Weights: 17.5 GB × (4/16) = 4.4 GB/GPU (INT4)
- Available for KV cache: 80 GB - 4.4 GB = 75.6 GB/GPU
- KV cache per request (INT8): approximately 0.67 GB total, or 0.08 GB/GPU
- Maximum batch size: approximately 901 requests

Systems insight: Precision engineering changes serving economics by enabling larger batch sizes. Larger batches amortize the fixed cost of weight loading, shifting operations from memory-bound toward compute-bound. This single optimization can increase throughput by 2.4× or more.

Precision engineering reduces the bytes per memory transaction. Operator fusion reduces the number of transactions. Together, they attack the same fundamental bottleneck from complementary directions: when data must traverse a slow bus, *move less of it* (precision) and *move it fewer times* (fusion). The multiplicative interaction between these two techniques explains why high-performance serving stacks often deploy both simultaneously: FlashAttention removes the quadratic attention intermediates, and INT8 KV cache compression further halves the remaining KV-cache bytes. The combined effect exceeds what either technique achieves alone. Because these fusion and precision patterns recur across stable layer structures, the next question is when a compiler can own the transformations across the whole model graph.

9.5 Graph Compilation

After precision and fusion expose repeatable optimization patterns, graph compilation asks whether those patterns are stable enough for the compiler to own. Manually writing fused CUDA kernels for every possible combination of layers in a massive neural network is a Sisyphean task for human engineers. A graph compiler analyzes the model's computational graph and generates optimized,

hardware-aware machine instructions, transforming high-level PyTorch code into specialized kernels when shape stability and replay volume justify the compilation cost.

> Systems Perspective 9.3: Hardware-software co-design
>
> The deeper payoff of compilation is spatial, not just arithmetic. A compiler like XLA or TensorRT reshapes the computation to fit the *physical layout* of the target accelerator: it tiles a matrix multiply to the dimensions of a systolic array, places intermediates to match the memory hierarchy, and routes element-wise work to Vector ALUs while reserving the systolic array for GEMMs. This is what makes heterogeneous execution at scale tractable. The same model graph maps onto different silicon by changing the mapping, not the math.

9.5.1 The compilation pipeline

A graph compiler transforms a high-level model definition (Python code) into optimized hardware instructions through a multi-stage pipeline. To visualize this process, consider a standard transformer FFN block consisting of a projection, an activation, a second projection, and a layer normalization: `LayerNorm(Linear(GELU(Linear(x))))`. In standard PyTorch eager execution, this sequence triggers four separate kernel launches, each reading from and writing to HBM.

In the **graph capture** stage, the compiler traces the model's execution to construct a computational graph, a directed acyclic graph where nodes represent operations and edges represent tensor dependencies. For the FFN block, this results in a graph with four primary nodes plus their associated parameter tensors. Dynamic Python control flow (loops, conditionals) must be handled by either tracing through a representative execution path or by using compiler-specific annotations to mark dynamic dimensions.

During **graph-level optimization**, the compiler applies algebraic simplifications and operation rewriting. It identifies that the bias addition in the first `Linear` layer can be folded into the matrix multiplication kernel. It also recognizes that the `GELU` activation is an element-wise operation that depends only on the output of the first `Linear`. These standard compiler optimizations can reduce graph size by 10–30 percent before any hardware-specific work begins.

The operator fusion pass is the most critical for performance. It identifies sequences of operations that can be combined into single kernels to reduce memory traffic. For the FFN block, the compiler fuses the `GELU` activation into the tail of the first `Linear` kernel (if supported as an epilogue) or fuses the `GELU` and the subsequent `LayerNorm` into a single kernel. Instead of writing the intermediate result of the first `Linear` to HBM and reading it back for `GELU`, the fused kernel keeps the data in the GPU's SRAM or registers. This typically reduces the number of HBM accesses by 30–50 percent, directly alleviating the memory bandwidth bottleneck.

The memory planning pass determines when to allocate and free tensors. Without optimization, a transformer might allocate separate buffers for every intermediate activation. The compiler analyzes tensor lifetimes, recognizing that the input to the first `Linear` is no longer needed after the second `Linear` computes its output, and reuses the same physical memory addresses. For inference and other forward-only workloads, this buffer reuse can turn peak temporary memory from the sum of many layer-local buffers into the maximum live working set. Training activations that must be saved for the backward pass require checkpointing or rematerialization; ordinary buffer reuse cannot make those saved tensors disappear. Memory planning also interacts with operator fusion: fusing two operations eliminates the intermediate tensor between them, which both removes the HBM traffic and removes the memory allocation. The compiler must reason about both effects jointly to make profitable decisions.

The kernel selection pass maps each fused operation to a specific machine code implementation. For the computationally heavy linear projections, the compiler selects a vendor-optimized cuBLAS or CUTLASS GEMM kernel. For the fused `GELU-LayerNorm` sequence, it generates a custom Triton kernel that keeps data in SRAM. The result for the FFN block is a reduction from 4 separate kernels to 2 highly optimized kernels, with a corresponding reduction in global memory traffic.

9.5.2 torch.compile

For workloads already written in PyTorch, `torch.compile` is the least disruptive compiler intervention: it tries to capture enough static graph to fuse memory-bound regions while preserving PyTorch's dynamic Python execution model. It operates through three components: **TorchDynamo** for graph capture, **TorchInductor** for code generation, and **AOTAutograd** for ahead-of-time backward graph construction when training workloads need compiled backward passes.

TorchDynamo operates at the Python bytecode level[9], a design choice that distinguishes it from earlier tracing approaches. Previous tracing methods (torch.jit.trace, torch.fx) operated at the Python source or abstract syntax tree (AST) level, requiring users to avoid unsupported Python constructs. TorchDynamo intercepts the bytecode interpreter itself, capturing a computational graph without requiring the user to modify their model code. When TorchDynamo encounters Python constructs it cannot trace (data-dependent control flow, unsupported operations), it inserts a **graph break** that splits the trace into multiple subgraphs, each compiled independently. The goal is to capture as large a subgraph as possible while gracefully handling dynamic Python behavior.

TorchInductor generates optimized Triton kernels (for GPU) or C++/OpenMP code (for CPU) from the captured graph. Triton is a domain-specific language for writing GPU kernels in Python-like syntax, abstracting away thread block management and memory coalescing while still exposing tiling and fusion decisions. TorchInductor automatically fuses element-wise operations, reduces memory traffic by combining operations that share inputs, and selects tile sizes through autotuning.

A minimal example illustrates the usage:

```
import torch


def transformer_block(x, w1, w2, ln_weight, ln_bias):
    """Unfused transformer FFN block."""
    h = x @ w1  # Linear projection
    h = torch.nn.functional.gelu(h)  # Activation
    h = h @ w2  # Output projection
    # Layer normalization
    mean = h.mean(dim=-1, keepdim=True)
    var = h.var(dim=-1, keepdim=True, unbiased=False)
    h = (h - mean) / torch.sqrt(var + 1e-5)
    h = h * ln_weight + ln_bias
    return h


# Compile the function-TorchDynamo traces, TorchInductor optimizes
compiled_block = torch.compile(transformer_block)

# First call triggers compilation; subsequent calls use compiled code
output = compiled_block(x, w1, w2, ln_weight, ln_bias)
```

In this example, `torch.compile` will fuse the GELU activation with surrounding operations, combine the layer normalization mean/variance/normalize steps into a single kernel, and potentially fuse the bias addition with the preceding GEMM. Model code remains standard PyTorch; the compiler handles the optimization.

9.5.3 XLA and TPU optimization

XLA pays for performance with static structure. Used as the backend for JAX and TensorFlow, it generates the High Level Optimizer (HLO) intermediate representation that targets multiple backends, including Google Tensor Processing Units (TPUs), NVIDIA GPUs, and CPUs. Unlike TorchInductor, which generates Triton code targeting NVIDIA GPUs while preserving more Python flexibility, XLA enforces **whole-program compilation**, tracing the entire computation as a single static graph and enabling global optimizations that span across layers and even across the forward and backward passes.

The global view enables XLA's most distinctive capability: **General and Scalable Parallelization for ML Computation Graphs (GSPMD)**, where SPMD denotes the single program, multiple data execution model in which every device runs the same program over a different shard of the data. In

9 **TorchDynamo Bytecode Interception**: By hooking CPython's frame evaluation function (PEP 523, added in Python 3.6), TorchDynamo captures the computation graph at the lowest level of the Python interpreter, below any source-level abstractions. This is why it can trace through decorators, closures, and third-party libraries that defeated earlier tracing approaches. The trade-off is tight coupling to CPython internals: TorchDynamo must be updated for each new Python version, and it cannot run on alternative interpreters like PyPy.

distributed training, GSPMD automatically partitions the computation graph across thousands of TPU cores based on a few high-level user annotations. While a PyTorch user must manually wrap models with `DistributedDataParallel` or `FullyShardedDataParallel`, an XLA user defines the computation for a single device and allows the compiler to infer the necessary communication primitives (AllReduce, AllGather) and insert them into the graph. This allows for complex hybrid sharding strategies that are difficult to implement manually.

For TPU hardware specifically, XLA performs layout optimizations unavailable on other platforms. It maps matrix multiplications onto the TPU's systolic array architecture, padding dimensions to align with the 128×128 hardware units and scheduling instructions to hide the latency of HBM fetches. The impact of these optimizations is visible in MFU metrics. On large-scale LLM training workloads, highly tuned JAX/XLA TPU runs have reported MFU in the 55–65 percent range, while less-tuned PyTorch/GPU setups can sit lower.

The trade-off for XLA's performance is compilation latency and rigidity. Because XLA must analyze the full static graph, initial compilation can take minutes for large models, compared to seconds for `torch.compile`. Any change in input shape triggers a full recompilation. This makes XLA excellent for steady-state production workloads where the graph is static and the model runs for days or weeks, but challenging for research environments involving dynamic shapes or rapid experimental iteration. The choice between `torch.compile` and XLA often follows the hardware and software stack: NVIDIA GPU workflows commonly start from PyTorch, while TPU workflows commonly use JAX or TensorFlow with XLA.

9.5.4 TensorRT: Inference optimization

NVIDIA TensorRT is a specialized inference compiler that treats the model not as a *flexible program* but as a *rigid global optimization problem*. Because inference requires no backward pass and no gradient storage, TensorRT applies aggressive transformations that would be mathematically invalid or impractically slow for training. It performs a calibration pass where it runs the model on representative data to determine the numerical range of every activation tensor.

This calibration enables **mixed-precision quantization** at a granular level. Blindly quantizing all layers of a 70B parameter LLM to INT8 often degrades perplexity. TensorRT can calibrate INT8 ranges from representative data and can build mixed-precision inference engines when precisions are enabled or constrained; for LLMs, layer-wise fallback policies usually require explicit quantization analysis or user-specified precision constraints. TensorRT also eliminates training-only operations (dropout, batch normalization running statistics updates), optimizes for static shapes by generating kernels tuned for exact dimensions, and plans memory precisely since no gradient tensors are needed.

TensorRT performs **kernel autotuning** far beyond simple heuristics. For every operation in the graph, it benchmarks dozens of candidate kernels, varying tile sizes, thread block configurations, and unrolling factors, on the actual target hardware. It selects the fastest implementation for that specific GPU and input shape. The performance gap between TensorRT and general-purpose compilers can be substantial, especially for stable inference workloads where the engine can specialize aggressively to a fixed shape range and GPU architecture.

The trade-off for TensorRT's aggressive optimization is reduced flexibility and high compilation cost. Compilation times are measured in minutes to hours (30–60 minutes for a 70B model), and the resulting engine is strictly tied to the specific GPU architecture and input shape range. Changing any of these requires recompilation. This makes TensorRT a common fit for stable, high-volume production deployments, while `torch.compile` is often a better fit for development and lower-volume services where rapid iteration matters more than extracting the last percentage of throughput.

9.5.5 Compilation overhead and trade-offs

Graph compilation is not free. The compilation process itself takes time, ranging from seconds for small models with `torch.compile` to minutes or hours for large models with TensorRT's full optimization pipeline. This overhead must be amortized over the number of times the compiled model executes.

For training workloads that run for hours or days, compilation overhead is negligible. For inference workloads that serve millions of requests, the one-time compilation cost is similarly amortized. The

problematic case is dynamic or infrequent workloads: a model that is compiled once but serves only a few hundred requests before being replaced by a new version may not recoup the compilation cost.

Graph breaks are a related challenge specific to `torch.compile`. When TorchDynamo encounters Python code it cannot trace (data-dependent control flow, calls to uncompiled libraries, dynamic tensor shapes that change between iterations), it inserts a graph break. Each break produces a separate compiled subgraph with its own compilation overhead and potential optimization boundaries. A model with 50 graph breaks produces 50+ small compiled regions, each potentially too small for meaningful fusion. Reducing graph breaks requires refactoring the model code to be more "compiler-friendly," replacing Python control flow with tensor operations and ensuring static shapes where possible.

Dynamic shapes present a fundamental tension between compilation and flexibility. A model compiled for input shape [batch=32, seq=512] will recompile when it encounters [batch=16, seq=1024]. TorchInductor supports "dynamic shapes" by generating kernels with symbolic dimensions, but this generality comes at the cost of reduced optimization compared to kernels specialized for exact shapes. TensorRT sidesteps this by requiring the user to specify a range of input shapes at compilation time, generating kernels that handle the specified range but nothing outside it.

Despite these limitations, graph compilation remains one of the most accessible optimization techniques: it requires no model modifications, no custom kernels, and minimal code changes. For many stable-shape PyTorch workloads with launch overhead or fusible element-wise regions, a single `torch.compile` call can provide 10–40 percent speedup, making it a natural early optimization to test before considering more specialized techniques.

Graph compilation has reshaped the performance engineering workflow. For workloads that match the compiler's assumptions, a single line of code can capture a significant fraction of the improvement that once required manual kernel optimization, freeing the engineer to focus on algorithmic and architectural optimizations such as speculative decoding, MoE, and precision engineering that compilers cannot automate. The compiler handles routine graph transformations; the engineer handles the workload-specific design choices. As compiler coverage improves, this division of labor makes the higher-level system design skills in this chapter more valuable relative to routine low-level kernel tuning.

9.5.6 Compilation modes and backends

`torch.compile` mode selection is an amortization decision: spend more compile time only when the workload will replay stable shapes enough times to recover that cost. The `default` mode is the development baseline, applying standard optimizations such as element-wise fusion, memory planning, and pretuned kernel selection with compilation times suitable for iteration. The `reduce-overhead` mode is the launch-bound choice, adding CUDA Graphs (discussed in section 9.3.3) to eliminate kernel launch overhead when small models spend a meaningful fraction of time in dispatch. The `max-autotune` mode is the production specialization choice, benchmarking multiple tile sizes, thread-block configurations, and memory-access patterns per operation to select the fastest kernel; the 10–30 minute compile cost is justified only when a stable deployment will reuse the optimized graph many times.

Backend choice follows the same constraint: the more stable and deployment-specific the workload, the more specialized the runtime can be. TorchInductor (the default) generates Triton kernels for GPU and C++ for CPU. For deployment-specific optimization, the model can be exported through `torch.export` to an intermediate representation that can be consumed by TensorRT, Open Neural Network Exchange (ONNX) Runtime, or other inference-specialized runtimes. Each backend applies its own optimization passes on top of the common graph-level transformations.

9.5.7 The triton language

Between hand-written CUDA and fully automated graph compilers sits **Triton**[10], a Python-based language for writing GPU kernels. Triton occupies a middle ground: the programmer specifies the algorithm (tiling strategy, fusion pattern) while Triton handles low-level concerns (thread block scheduling, memory coalescing, shared memory management).

A Triton kernel for fused GELU activation illustrates the programming model:

[10] **Triton**: Introduced by Tillet, Kung, and Cox as an intermediate language and compiler for tiled neural-network computations (Tillet et al. 2019), and later developed at OpenAI. The key design decision was making the *tile* – not the *thread* – the fundamental programming abstraction. This choice directly mirrors the tiling strategy of FlashAttention: the programmer reasons about blocks of data that fit in SRAM, and the compiler maps those blocks to GPU threads and shared memory. This abstraction reduces exposure to low-level CUDA concerns such as warp divergence, bank conflicts, and coalescing while preserving control over the memory hierarchy decisions that determine ML kernel performance.

```python
import triton
import triton.language as tl


@triton.jit
def fused_gelu_kernel(
    input_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    # Each program instance handles BLOCK_SIZE elements
    pid = tl.program_id(0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    # Load input tile from HBM into registers
    x = tl.load(input_ptr + offsets, mask=mask)

    # Fused GELU computation (tanh approximation)
    # GELU(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 *
    # x^3)))
    x_cubed = x * x * x
    inner = 0.7978845608 * (x + 0.044715 * x_cubed)
    gelu = 0.5 * x * (1.0 + tl.math.tanh(inner))

    # Store result back to HBM
    tl.store(output_ptr + offsets, gelu, mask=mask)
```

The programmer thinks in terms of tiles (`BLOCK_SIZE` elements), not individual threads. Triton compiles this to Parallel Thread Execution (PTX)/SASS instructions, handling thread-to-data mapping, memory coalescing, and register allocation automatically. This makes it feasible for ML engineers (rather than GPU specialists) to write custom fused kernels when the automatic compiler misses a fusion opportunity.

The real power of Triton emerges when fusing multiple operations. A Triton kernel that implements `y = LayerNorm(GELU(x))` loads `x` once from HBM, computes both GELU and LayerNorm in registers/shared memory, and writes `y` once to HBM. Without fusion, this sequence requires three HBM round-trips: read `x`, write `GELU(x)`, read `GELU(x)`, write `norm_input`, read `norm_input`, write `y`. The fused kernel reduces HBM traffic by $3\times$, and for memory-bound operations, this translates directly to a $3\times$ speedup.

The performance-engineering consequence is that Triton turns many custom fusion opportunities into compiler output. When `torch.compile` identifies a fusion opportunity that requires a custom kernel, TorchInductor automatically generates Triton code for the fused operation. Many of the fusion benefits described in this section therefore come through a single `torch.compile` call, without hand-written Triton code. For advanced cases where the compiler's heuristics miss the bottleneck, hand-written Triton kernels provide a middle ground between the accessibility of PyTorch and the performance of hand-tuned CUDA.

A simple profile breakdown shows how compiler optimizations convert overhead into useful throughput.



Napkin Math 9.3: The compilation dividend

Problem: A 13B parameter model is deployed for inference. Without compilation, the PyTorch eager mode processes 120 tokens/s on a single H100. The Nsight Systems trace reveals that 35 percent of step time is spent in element-wise kernels (LayerNorm, GELU, residual additions) and 15 percent is kernel launch overhead. Applying `torch.compile` with the `max-autotune` backend, estimate the new throughput.

Math:

Step 1: Identify the addressable overhead. Element-wise kernels (35 percent) and launch overhead (15 percent) total 50 percent of execution time. torch.compile fuses element-wise operations (reducing their time by approximately 70 percent due to eliminated HBM round-trips) and reduces kernel launches (eliminating most launch overhead).

Step 2: Estimate postcompilation time. Original time per token: $1/120 = 8.33\,\text{ms}$.

- GEMM time (unchanged): $8.33 \times 0.50 = 4.17\,\text{ms}$
- Element-wise time (70 percent reduction): $8.33 \times 0.35 \times 0.30 = 0.87\,\text{ms}$
- Launch overhead (80 percent reduction): $8.33 \times 0.15 \times 0.20 = 0.25\,\text{ms}$

New time per token: $4.17 + 0.87 + 0.25 = 5.29\,\text{ms}$, yielding approximately 189 tokens/s.

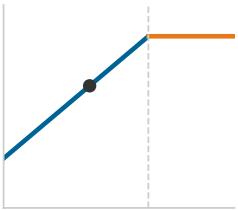
Systems insight: torch.compile delivers a 1.58× speedup by fusing element-wise operations and reducing launch overhead, without touching the GEMM kernels. The remaining bottleneck is now the GEMM itself (79 percent of step time), indicating that further improvement requires either precision reduction or batching.

Graph compilation automates what manual kernel engineering achieves for individual operations, applying it systematically across the entire model graph. It also exposes a boundary: compilers can reorganize known operations, but they cannot invent a different computation. The next section turns to optimizations that cross that boundary.

9.6 Algorithmic Performance Transformations

The preceding sections optimized the same dense computation by moving fewer bytes, using narrower formats, or letting compilers reorganize kernels. Algorithmic transformations change the work itself. Speculative decoding attacks the memory-bound decode loop by trying to accept multiple tokens per target-model pass, while MoE attacks dense-model cost by activating only the experts a token needs. Both techniques can improve the iron-law budget, but both can also create new bottlenecks if their extra control logic, communication, or imbalance exceeds the work they remove.

9.6.1 Speculative decoding



Speculative decoding shifts decode toward the compute-bound ridge.

Speculative decoding[11] is a latency optimization that breaks the sequential bottleneck of autoregressive generation by using a smaller draft model to predict multiple tokens, which are then verified in parallel by the target model (Leviathan et al. 2023; Chen et al. 2023). The performance-engineering relevance is arithmetic intensity: a single target-model forward pass can process K candidate tokens for roughly the same weight-streaming cost as one token, shifting the decode operating point from the memory-bound slope toward the roofline ridge.

Figure 9.6 contrasts the two decode paths: standard decoding advances one token per sequential target-model pass, while speculative decoding lets the draft model propose a block of K tokens that the target model verifies in a single parallel pass, emitting the accepted prefix together and resampling only at the first rejection.

The transformation is useful only when the draft model is accurate enough and cheap enough. If too many draft tokens are rejected, the target model still loads its weights but accepts little useful work. If the draft model is too expensive, verification becomes compute-bound and erases the bandwidth benefit. Batching changes the calculation as well: at large batch sizes the decode phase is already closer to compute-bound, so the marginal benefit of speculation shrinks. This chapter uses speculative decoding to show how an algorithm can alter the performance equation; section 10.4.6 treats it as a serving policy with admission control and SLA consequences.

11 **Speculative Decoding**: The draft model may produce tokens that the target model rejects, so the technique trades extra draft-model compute for reduced target-model memory bandwidth pressure. The target model's weights are loaded from HBM once to process K candidate tokens, effectively increasing the arithmetic intensity of the decode phase by $K\times$ when enough tokens are accepted.

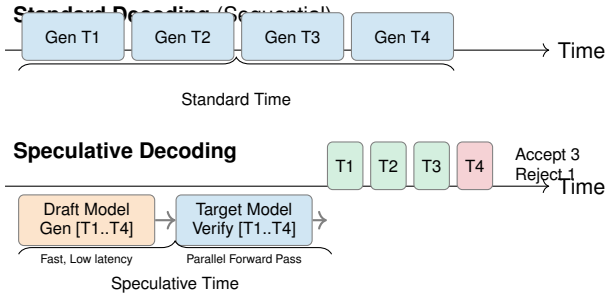


Figure 9.6: Speculative Decoding Process: Instead of generating tokens sequentially with the large target model, a small draft model proposes a sequence of K tokens. The target model verifies all K tokens in a single parallel forward pass. Drafted tokens that pass the acceptance test are emitted together; the first rejected token triggers a corrected sample and discards the remaining draft.

Algorithm 9.1 makes the acceptance test precise: after the single target-model pass, each drafted token is accepted with probability $\min(1, p_\theta/q_\phi)$, and a rejection resamples from the normalized positive residual $(p_\theta - q_\phi)_+$, so the emitted sequence is distributed exactly as the target model would have produced unaided. That correctness guarantee is what lets speculative decoding trade draft-model compute for latency without shifting the output distribution.

Algorithm 9.1 Speculative decoding

Require: target distribution $p_\theta(\cdot \mid \cdot)$; draft distribution $q_\phi(\cdot \mid \cdot)$; prefix $y_{1:t}$; draft length K
Ensure: one or more new tokens distributed as p_θ

for $i = 1$ to K **do**
  draft $\hat{y}_{t+i}$ from $q_\phi(\cdot \mid y_{1:t}, \hat{y}_{t+1:t+i-1})$
end for
run p_θ once on the prefix plus all K drafts to obtain target distributions for the drafted positions and final next-token position
for $i = 1$ to K **do**
  $a_i \leftarrow \min\left(1, \frac{p_\theta(\hat{y}_{t+i} \mid y_{1:t}, \hat{y}_{t+1:t+i-1})}{q_\phi(\hat{y}_{t+i} \mid y_{1:t}, \hat{y}_{t+1:t+i-1})}\right)$
  accept $\hat{y}_{t+i}$ with probability a_i
  if $\hat{y}_{t+i}$ is rejected **then**
    sample one corrected token from the normalized positive residual $(p_\theta - q_\phi)_+$
    emit accepted draft tokens and the corrected token; discard the remaining draft; start a new round
  end if
end for
if all K drafts were accepted, emit them and sample one extra token from the target distribution

Speculative decoding pays off when the draft model is cheap and its proposed tokens agree with the target model often enough that one target pass validates several tokens. The limiting cost is the rejection rate: rejected drafts waste draft-model work and shorten the accepted run, so the serving system must tune draft length and draft-model quality together rather than treating speculation as free parallelism.

9.6.2 Mixture of experts

A 1-trillion parameter dense model delivers superior reasoning capabilities, but its latency and compute costs are prohibitive for most serving budgets. The Mixture-of-Experts (MoE) architecture resolves this tension by training a massive model but activating only a small, relevant fraction of it for any given token. By routing inputs to specialized sub-networks, MoE breaks the iron link between model size and inference cost.

For performance engineering, the local question is whether active-parameter savings exceed the costs of routing and communication. Sparse activation improves the iron law budget only when inactive experts stay off the critical path and the router avoids creating a new AllToAll or load-imbalance bottleneck.

A standard MoE transformer layer replaces the feed-forward network (FFN) with multiple parallel "expert" FFNs and a lightweight **router** (also called a gating network) that selects which experts process each token. That architecture shifts the performance problem from dense matrix throughput to routing, memory residency, and load balance. When experts are distributed across GPUs, expert parallelism places different experts on different devices, and an AllToAll exchange moves each token's hidden state to its assigned expert before returning the result. The active-parameter savings are valuable only if that routing traffic and any expert imbalance remain smaller than the dense compute they replace.

9.7 Communication-Computation Overlap

After changing the local work through fusion, precision, compilation, speculation, or sparse expert routing, the next exposed iron-law term is often communication. Section 6.8 established overlap as the technique that removes communication from the critical path when enough useful compute remains to hide it; this section supplies the single-node mechanics those chapters assumed, namely the CUDA streams, SM partitioning, and bucket hooks that actually run computation and communication concurrently on one accelerator.

Consider a concrete example: a 70B model with tensor parallelism across 8 H100 GPUs. Each transformer layer requires two AllReduce operations (one after attention, one after FFN). Each AllReduce transfers approximately $2 \times d_{\text{model}} \times B \times 2$ bytes at FP16 (where $d_{\text{model}} = 8192$ for a 70B

model and B is batch size). At batch size 1, the data volume is $2 \times 8192 \times 1 \times 2 = 32.8$ KB per AllReduce. At 900 GB/s NVLink bandwidth, this transfer takes approximately 36.4 ns. However, the AllReduce launch overhead (approximately 5 μs) dominates the actual data transfer time by 100×. At batch size 1, the AllReduce overhead is dominated by software launch latency, not bandwidth, and overlap provides limited benefit because there is insufficient compute to hide behind.

At batch size 64, the data volume per AllReduce grows to $2 \times 8192 \times 64 \times 2 = 2.1$ MB, taking approximately 2.3 µs at NVLink bandwidth. The compute path over a shard is now a bundle of kernels, not a single GEMM: the attention and FFN work together take on the order of 20 μs for this configuration, while the communication remains around 2.3 µs. Here, the compute time exceeds the communication time by about 9×, and overlap becomes highly effective. This illustrates why batch size is the universal control knob: it simultaneously improves arithmetic intensity, GPU utilization, and communication overlap effectiveness.

This per-layer tensor-parallel example is the small-payload side of the overlap problem. Training gradient synchronization uses the same exposed-time test with much larger payloads, so the gradient-overlap calculation switches from activation exchanges to ring AllReduce over the gradient tensor.

9.7.1 Quantifying the overlap opportunity

The potential benefit of communication-computation overlap depends on the relative magnitudes of communication and computation time, which vary dramatically across system configurations.

Consider a 7B parameter model trained on 8 H100 GPUs within a single node connected by NVLink at 450 GB/s per direction. The gradient tensor contains 16.1 GB of FP16 values. By the ring AllReduce analysis in section 6.4.3, synchronization transfers $2 \times (N-1)/N = 1.75\times$ times the gradient volume, taking approximately 62.5 ms at NVLink bandwidth. The backward pass computation, at 40 percent Model FLOPs Utilization (MFU, the fraction of peak throughput spent on useful model FLOPs, defined in section 5.4), takes approximately 166.3 ms.

Without overlap, the training step requires 311.9 ms (forward + backward + AllReduce). With overlap, the AllReduce is fully hidden behind the backward pass, reducing the step time to 249.4 ms, a 1.25× improvement. This example illustrates a critical property: overlap is most effective when backward compute time exceeds AllReduce time. As figure 9.7 shows, as cluster scale increases from 8 to 1024 GPUs, overlap efficiency degrades from ~90 percent to ~18 percent because communication overhead eventually exceeds the computation budget. In that figure, each bar is normalized to the fixed compute budget of 100 units, and the bar height grows roughly fivefold across the x-axis because the exposed-communication segment stacked on top of that budget is precisely the portion of communication that overlap can no longer hide. For smaller models or slower interconnects (for example, PCIe at 64 GB/s instead of NVLink at 900 GB/s), the AllReduce would exceed the backward pass, and no amount of overlap can fully hide the communication.

9.7.2 CUDA streams and asynchronous execution

The mechanism enabling overlap on NVIDIA GPUs is **CUDA streams**. A CUDA stream is an ordered sequence of GPU operations (kernel launches, memory copies, NCCL collectives) that execute sequentially within the stream but can execute concurrently with operations in other streams. The application maintains two distinct streams: a compute stream for matrix multiplications and element-wise kernels, and a communication stream for NCCL operations. The workflow proceeds by launching a compute kernel on the first stream and immediately triggering an asynchronous communication call on the second:

```
compute_stream = torch.cuda.Stream()
comm_stream = torch.cuda.Stream()

with torch.cuda.stream(compute_stream):
    output = torch.matmul(A, B)  # Non-blocking compute

with torch.cuda.stream(comm_stream):
    dist.all_reduce(gradients, async_op=True)  # Non-blocking comm

torch.cuda.synchronize()  # Wait for both to complete
```

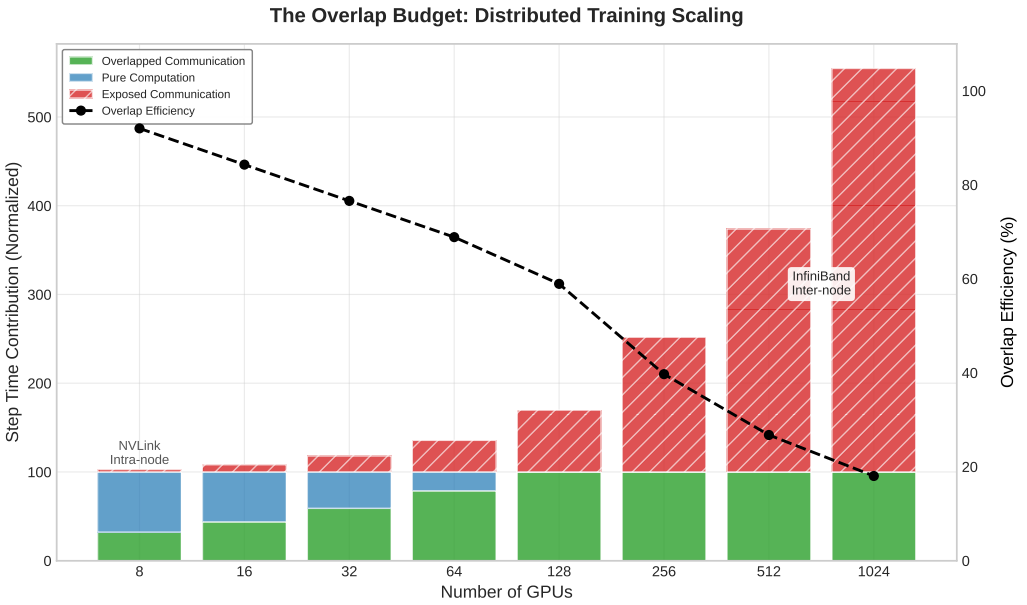


Figure 9.7: The Overlap Budget: As cluster scale increases (8 to 1024 GPUs), the communication overhead grows, eventually exceeding the computation budget. The overlap efficiency (black line) represents the percentage of communication hidden behind computation, which degrades from ~90 percent to ~18 percent due to inter-node latency and bandwidth constraints on a 70B parameter model.

The GPU hardware scheduler interleaves execution units from both streams, running the GEMM on the SMs while the NVLink engine handles the AllReduce data transfer. However, while streams provide *logical* concurrency, they contend for *physical* resources. The SMs must manage the data movement instructions for the communication kernel. On an H100 with 132 SMs, a heavy NCCL operation might occupy 4–8 SMs solely for protocol processing and memory copying, leading to **SM partitioning**: the available compute throughput is reduced by 3–6 percent during communication. If the compute kernel is dense enough to saturate 100 percent of the SMs, enabling overlap can paradoxically slow down execution due to this resource contention, a phenomenon known as interference. In practice, the 3–6 percent compute throughput reduction is far smaller than the communication time that would otherwise be exposed, making the trade-off overwhelmingly favorable.

In practice, achieving effective overlap requires attention to several details. The communication operation must be launched early enough to overlap with subsequent compute; in backward passes, DDP starts each bucket's AllReduce as soon as that bucket is ready so NCCL launch latency is hidden behind later gradient kernels. Synchronization points (where one stream waits for another) must be minimized, as each synchronization serializes execution.

PyTorch's Distributed Data Parallel (DDP) module implements gradient overlap by registering backward hooks on each parameter. When a parameter's gradient is computed during the backward pass, the hook triggers an asynchronous AllReduce on a separate NCCL stream. DDP synchronizes this communication stream with the main compute stream at the end of the backward pass, so every gradient reduction is guaranteed complete before the optimizer step reads the gradients. This design overlaps gradient communication with gradient computation automatically, without requiring user intervention.

When engineers create their own side streams for custom overlap patterns, a set of PyTorch-specific footguns regularly causes silent correctness or performance failures. The default behavior in PyTorch is that all operations enqueue onto the default stream. A tensor produced on the default stream and consumed on a side stream is not automatically safe: without an explicit `event.record()` / `event.wait()` barrier between them, the GPU may begin consuming the tensor before it is fully written. Similarly, memory copies launched without `non_blocking=True` insert implicit synchro-

nization points that serialize the streams the engineer was trying to overlap, often eliminating the overlap benefit entirely. DDP avoids these hazards by managing `record_event()` calls inside its bucket hooks, ensuring each AllReduce starts only after the corresponding gradient compute event has completed. Manually written overlap code must replicate this discipline or risk data races that produce incorrect gradients with no error signal at the framework level.

The techniques covered so far – fusion, precision, compilation, speculative decoding, MoE, and communication overlap – address different aspects of the performance equation. Identifying which technique to apply in a given situation requires systematic measurement, and the profiling tools that make this diagnosis possible are the final piece of the optimization toolkit.

9.8 System Profiling

An engineer spends two weeks rewriting a PyTorch module into a custom CUDA kernel to make it 5× faster, only to discover the overall model latency did not budge because the system was entirely I/O bound. Performance engineering without measurement is expensive guesswork. System profiling provides the surgical diagnostics required to identify exactly where the GPU is waiting, allowing us to apply our optimizations with precision.

> Systems Perspective 9.4: The physics of profiling (the Heisenberg effect)
>
> Observing a distributed ML system fundamentally alters its behavior. Gathering high-resolution telemetry (such as tracking every CUDA kernel launch or memory allocation) consumes PCIe bandwidth, CPU cycles, and memory capacity. In heavily optimized serving pipelines, the overhead of profiling can artificially induce the very latency spikes the engineer is trying to diagnose. This "Heisenberg Effect" requires a deliberate strategy: deploying low-overhead, sampling-based telemetry for continuous production monitoring, and only enabling high-fidelity, synchronous tracing in isolated diagnostic environments.
>
> The effect is particularly pronounced in PyTorch workloads. Enabling `torch.autograd.profiler` or `torch.cuda.memory._record_memory_history()` instructs the framework to retain references to intermediate activation tensors beyond their normal lifetime so that allocation metadata can be recorded. This prevents the memory allocator from reusing tensor buffers as it normally would, inflating peak HBM consumption and, in borderline cases, triggering Out-Of-Memory errors that do not occur in unobserved execution. Additionally, graph-optimization passes in `torch.compile` detect tensor observation and conservatively disable certain buffer-reuse fusions, causing the profiled execution to follow a slower code path than the production path. The practical consequence is that a profiling run must be treated as a distinct experiment: it characterizes the structure of the computation accurately, but its absolute latency and memory numbers will overstate production measurements.

9.8.1 The profiling hierarchy

Profiling levels are useful only if they guide the drill-down from symptom to intervention. ML system profiling operates at four levels, each providing different granularity and targeting different bottleneck categories. Figure 9.8 makes this drill-down explicit, starting from application-level symptoms and descending toward kernel and hardware-counter evidence.

The drill-down starts at the application level, where end-to-end metrics such as tokens per second, time-to-first-token, P99 latency, and GPU utilization over time reveal that the user-facing system has missed its target. It then descends to distributed profiling across GPUs and nodes, where communication patterns show whether collectives overlap with compute, which operation dominates step time, and whether load imbalance exists across ranks. Trace-level profiling narrows the diagnosis to the timeline of GPU kernels, CPU operations, and data transfers within a training step or inference request, exposing launch gaps, idle bubbles, sequential bottlenecks, and overlap opportunities. Operation-level profiling completes the chain with tools like NVIDIA Nsight Compute, where achieved memory bandwidth, compute utilization, occupancy, and instruction mix determine whether a specific kernel is memory-bound, compute-bound, or limited by implementation details—including whether it is issuing Tensor Core instructions (HMMA for FP16/BF16, IMMA for INT8)

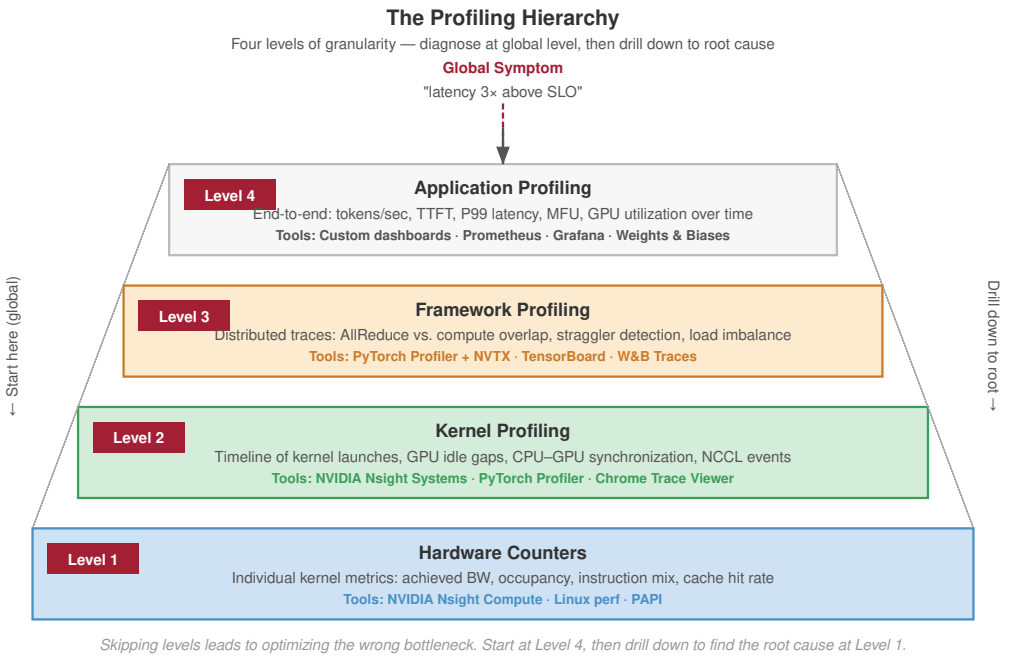


Figure 9.8: The Profiling Hierarchy: Four levels of profiling granularity. Start at Level 4 (Application Profiling) for global symptoms, and drill down through Framework and Kernel profiling to find the root cause at Level 1 (Hardware Counters).

rather than falling back to scalar CUDA Core instructions (FMA/ALU). A matrix multiplication whose hidden-dimension or sequence-length tile is not divisible by the Tensor Core alignment requirement (multiples of 8 for FP16, 16 for INT8) silently bypasses Tensor Cores entirely and executes on CUDA Cores at a fraction of the peak throughput. This shape mismatch is one of the most common silent performance bugs in custom ML kernels and is invisible without inspecting the instruction mix counter.

Diagnosing performance requires a drill-down approach, moving from global symptoms to local causes. When profiling a 70B model serving pipeline, the application level might reveal "end-to-end latency is 150 ms/token, 3× above the SLO." Descending to the distributed level, traces might show one GPU consistently lagging in AllReduce operations, pointing to a straggler or network congestion. Zooming into the trace level on that specific GPU reveals the timeline of kernel execution, exposing gaps where the SMs are idle due to scheduling overhead. Finally, kernel-level profiling (using Nsight Compute) inspects the specific instruction mix of a single matrix multiplication, revealing cache misses or register pressure. Skipping levels often leads to optimizing the wrong bottleneck: optimizing a kernel is futile if the GPU is spending 40 percent of its time waiting on the network.

9.8.2 Key performance metrics

Metrics are not interchangeable; each answers a different bottleneck question. MFU and MBU diagnose useful hardware work, TTFT and ITL diagnose serving latency, and throughput exposes capacity under batching. Understanding their relationships and trade-offs is essential for performance engineering.

Table 9.5 maps the common metrics to the bottleneck question each one answers.

Table 9.5: Performance Metrics and Their Diagnostic Use: Each metric measures a different layer of the system, so choosing the wrong metric can hide the real bottleneck.

Metric	Definition	Best diagnostic use	Caveat or target
Model FLOPs Utilization (MFU)	Fraction of hardware peak FLOP/s productively used by model computation	Training efficiency across memory limits, launch overhead, communication wait, and software overhead	40–60% is common in well-tuned LLM training; above 60% is excellent
Hardware FLOPs Utilization (HFU)	Total arithmetic operations executed, including recomputation, divided by peak FLOP/s	Separating useful model work from overhead such as activation recomputation	Usually exceeds MFU, and the HFU–MFU gap estimates wasted compute
Time-to-first-token (TTFT)	Latency from request arrival to the first generated token	Interactive serving responsiveness, including queueing, prefill, and initialization	Below 500 ms is generally acceptable; below 200 ms feels responsive
Inter-token latency (ITL)	Time between consecutive generated tokens during decode	Perceived generation speed after the first token	Below 50 ms supports comfortable reading; real-time speech may require below 25 ms
Throughput	Tokens/second for the system, or tokens/second/GPU for per-device efficiency	Aggregate serving or training capacity under batching	Large batches improve throughput but increase per-request latency; section F.1.1 covers the queueing trade-off
Model Bandwidth Utilization (MBU)	Achieved memory bandwidth divided by hardware peak bandwidth	Memory-bound inference, especially decode workloads	An H100 decode step can show 2% MFU but 85% MBU, which indicates a memory-saturated and well-optimized path

9.8.3 Using the roofline for diagnosis

The metric choice in table 9.5 reflects the fundamental bottleneck shift between training and inference: MFU is usually the right training-efficiency lens, while MBU is often the right inference-efficiency lens when decode is limited by HBM traffic. The roofline model from section 9.1.5 becomes a diagnostic tool when combined with profiling data. Diagnosis follows three steps:

1. **Measure** the achieved FLOP/s and memory bandwidth for a kernel using Nsight Compute.
2. **Compute** the operational arithmetic intensity from the algorithm (FLOP/byte).
3. **Plot** the measured performance against the roofline ceiling.

A kernel that falls far below both the compute ceiling and the bandwidth ceiling has an implementation problem: launch overhead, poor memory access patterns, or low occupancy. A kernel that reaches the bandwidth ceiling but falls below the compute ceiling is memory-bound, and further optimization requires reducing memory traffic (fusion, precision) rather than improving compute efficiency. A kernel at the compute ceiling is compute-bound and can only be improved by algorithmic changes (reducing FLOPs) or faster hardware.

Consider a concrete example: a LayerNorm kernel profiled on an H100 reports 15 TFLOP/s of achieved compute and 2.8 TB/s of achieved memory bandwidth. Its arithmetic intensity is 15 TFLOP/s / 2.8 TB/s $\approx$ 5.4 FLOP/byte. The H100's ridge point is approximately 295.2 FLOP/byte at FP16. Since 5.4 FLOP/byte is far below 295.2 FLOP/byte, the kernel is strictly memory-bound. Its achieved 2.8 TB/s (83.6 percent of the H100's 3.35 TB/s peak bandwidth) confirms it is operating near the physical limit of the memory subsystem. The diagnosis is clear: further FLOP-level optimizations will yield little gain; performance can only be improved materially by reducing data movement, either through fusion (eliminating the HBM round-trip) or precision reduction (halving the bytes per element).

9.8.4 PyTorch profiler workflow

The PyTorch Profiler is most useful when application metrics show a slowdown but the responsible layer is still unknown. Its role is to narrow a global symptom to a layer, kernel family, or synchro-

nization point before a heavier tool is used. It integrates with the training loop to capture detailed traces with minimal code modification:

```
import torch
from torch.profiler import (
    profile,
    schedule,
    tensorboard_trace_handler,
)

# Profile 2 warmup steps + 3 active steps
with profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA,
    ],
    schedule=schedule(wait=1, warmup=2, active=3, repeat=1),
    on_trace_ready=tensorboard_trace_handler("./profiler_logs"),
    record_shapes=True,
    profile_memory=True,
    with_stack=True,
) as prof:
    for step, batch in enumerate(dataloader):
        if step >= 6:  # 1 wait + 2 warmup + 3 active
            break
        output = model(batch)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        prof.step()
```

The `schedule` parameter defines a warmup period where the profiler runs but does not record, allowing CUDA caches and just-in-time (JIT) compilation to stabilize before measurement begins. Without warmup, the first few iterations include one-time costs (kernel compilation, memory allocation, CUDA context initialization) that inflate the measured times and misrepresent steady-state performance.

The resulting trace, viewable in TensorBoard or Chrome's trace viewer, matters only insofar as each view maps to an intervention. Table 9.6 turns those views into a diagnostic checklist.

Table 9.6: Profiler Views and Interventions: PyTorch profiler views are useful when each visual signal points to a concrete next experiment.

Trace view	Diagnostic signal	Likely intervention
Kernel timeline	CUDA kernels, their duration, GPU idle gaps, and unexpectedly long kernels	Fuse operators, use CUDA Graphs, or investigate long kernels with Nsight Compute
Memory timeline	Allocation and deallocation spikes or gradual memory growth	Reuse buffers, plan allocations, or investigate leaks
CPU-GPU synchronization	Points where the CPU waits for the GPU, or the GPU waits for CPU launch work	Remove blocking synchronization and reduce launch overhead
Communication events	NCCL collective duration relative to compute kernels	Tune overlap, bucket sizing, or parallelization strategy

9.8.5 Nsight systems: Reading the timeline

The profiler is therefore a triage tool, not the final performance proof: it identifies whether the next experiment should target fusion, memory planning, CUDA Graphs, or overlap. NVIDIA Nsight Systems answers the trace-level question raised by higher-level profiling: whether the GPU timeline is continuous, whether the CPU is feeding it fast enough, and whether communication overlaps compute. The tool captures every GPU kernel launch, CUDA memory operation, NCCL communication, and CPU thread activity onto a unified timeline.

Nsight Systems becomes useful when the question has shifted from which layer is slow to why the full timeline has gaps. A typical workflow begins with capturing a trace of a few training or inference iterations:

```
nsys profile --trace=cuda,nvtx,osrt,cudnn,cublas \
    --output=llm_profile \
    --force-overwrite=true \
    python3 inference_server.py --num-steps=5
```

The `--trace` flags control which activities are recorded. The `cuda` flag captures kernel launches and memory operations. The `nvtx` flag captures user-annotated regions (PyTorch automatically annotates module boundaries with NVTX markers). The `cublas` and `cudnn` flags capture library-level operations, which helps identify whether a GEMM is using cuBLAS or a custom kernel. The command is useful because these rows expose whether the bottleneck is launch overhead, communication serialization, or kernel implementation.

The resulting `.nsys-rep` file is opened in the Nsight Systems GUI, which presents a multi-row timeline. Four rows carry the most diagnostic weight.

The CUDA HW row shows the actual kernel execution on the GPU. Each colored bar represents a kernel, with width proportional to execution time. Gaps between bars indicate GPU idle time, which represents wasted potential. For a well-optimized inference pipeline, the CUDA HW row should show nearly continuous kernel execution with minimal gaps.

The CUDA API row shows CPU-side CUDA API calls (kernel launches, memory allocations, synchronization). If CUDA API bars are significantly wider than the corresponding CUDA HW bars, the CPU is the bottleneck: it cannot launch kernels fast enough to keep the GPU busy. CUDA Graphs and `torch.compile` address exactly this kernel launch overhead problem.

The NCCL row (for distributed workloads) shows collective communication operations. Comparing the NCCL row with the CUDA HW row reveals whether communication overlaps with computation. If NCCL bars appear during gaps in the CUDA HW row, communication is serialized. If NCCL bars overlap with CUDA HW bars, overlap is working correctly.

The NVTX row shows user-annotated regions, which PyTorch maps to module names (`Linear`, `LayerNorm`, `Attention`). This connects low-level kernel names (often cryptic strings like `volta_fp16_s1688gemm_fp16_256x128_ldg8_f2f_nn`) to the model-level operations that produced them.

Five patterns reveal the most diagnostic information from a Nsight Systems trace. Table 9.7 maps each visual pattern to the bottleneck question it answers.

Table 9.7: Nsight Timeline Diagnostic Patterns: Timeline patterns connect visual structure to utilization, fusion, overlap, allocation, and arithmetic-efficiency questions.

Timeline pattern	Diagnostic question	Optimization direction
Kernel execution vs. idle gaps	How much of the trace is useful GPU execution rather than waiting?	Investigate launch overhead, CPU stalls, or input starvation
Distribution of kernel durations	Are many short kernels fragmenting the workload?	Try fusion, CUDA Graphs, or compiler capture
NCCL alignment with CUDA HW rows	Does communication overlap with computation?	Tune bucket sizes, scheduling, or parallelization layout
Memory allocation spikes	Are tensors being materialized or allocated repeatedly?	Add memory planning, buffer reuse, or fused kernels
GEMM vs. non-GEMM time	How much execution is useful matrix arithmetic vs. overhead?	Move non-GEMM work into fused kernels or reduce memory traffic

Experienced performance engineers develop pattern recognition for these traces, quickly identifying the dominant bottleneck from the visual structure of the timeline. A trace dominated by thin, closely packed kernel bars with minimal gaps indicates a well-optimized pipeline. A trace with large gaps between kernels, or with NCCL bars that do not overlap with CUDA HW bars, immediately reveals the primary optimization target.

9.8.6 Common bottleneck patterns

Profiling is valuable because recurring trace patterns map directly to optimization choices. Table 9.8 turns that pattern recognition into a diagnostic map: first identify the visual signature, then connect it to the likely cause and the intervention that changes the system.

Table 9.8: Common Bottleneck Trace Patterns: Diagnostic map from trace signature to likely cause, intervention, and trade-off. The table is most useful after the profiler has already localized the bottleneck to launch overhead, memory movement, communication, load balance, allocation behavior, or input preparation.

Trace Pattern	Likely Cause	Primary Intervention	Trade-Off to Check
Small gaps between many kernels	Kernel launch overhead dominates	Graph compilation with `torch.compile`, CUDA Graphs, or fusion	Compilation warmup and debugging complexity
Large memory spikes	Unfused operators materialize intermediate tensors	FlashAttention or custom Triton fusion	Numerical validation and kernel maintenance
Low FLOP/s with high bandwidth	Memory-bound operations dominate	Precision reduction, batching, or algorithmic changes	Accuracy, cache pressure, and quality guardrails
NCCL bars during GPU idle gaps	Communication is serialized rather than overlapped	DDP gradient overlap or pipeline restructuring	Bucket size, schedule complexity, and synchronization
Some GPUs wait for stragglers	Uneven work distribution or MoE load imbalance	MoE capacity tuning, auxiliary loss adjustment, or even tensor-parallel splits	Routing quality and utilization balance
Repeated large allocations	Buffers are recreated rather than reused	Memory planning, preallocated buffer pools, or checkpointing	Recompute cost and allocator behavior
Periodic GPU utilization drops	CPU preprocessing or tokenization is not pipelined	More data workers, preprocessing services, or pretokenized datasets	Storage cost and data freshness

A full decode trace shows how several of these smaller bottlenecks can combine into the dominant loss.

Example 9.2: The profiler detective

Problem: A 7B parameter LLM runs on a single H100 and shows 45 tokens/s during autoregressive generation at batch size 1. The pure FP16 weight-read roofline is higher, but after budgeting roughly 35 GB of total per-token traffic for weights, KV-cache reads, sampling, synchronization, and launch overhead, the bandwidth-limited full-decode ceiling is approximately 96 tokens/s. Where is the remaining 53 percent of realizable decode-step performance hiding?

Investigation:

Step 1: Kernel-level analysis. Run Nsight Compute on the dominant GEMM kernel. Result: achieves 2.8 TB/s effective bandwidth out of the H100's 3.35 TB/s peak. Efficiency: 84 percent. This kernel is performing well.

Step 2: Trace-level analysis. Run Nsight Systems on a full decode step. Result: 42 percent of step time is spent in GEMM kernels. The remaining 58 percent is split between:

- Attention kernels (including KV cache reads): 28 percent
- Layer normalization and activation kernels: 12 percent
- Softmax and top-k sampling: 8 percent
- Kernel launch gaps: 10 percent

Step 3: Identify optimization targets.

- The KV cache attention kernel achieves only 1.9 TB/s bandwidth because of irregular memory access patterns. Fix: Implement KV cache quantization (INT8) with better memory layout.

- Kernel launch gaps (10 percent of time) come from 120+ individual kernel launches per layer. Fix: Apply `torch.compile` to fuse element-wise operations, reducing to ~30 kernels per layer.
- Layer normalization and activation kernels are unfused. Fix: Fused LayerNorm-GELU kernel via Triton.

Systems insight: The dominant GEMM is near-optimal, but secondary operations and launch overhead consume over half the execution time. Systematic profiling reveals that the bottleneck is not where intuition suggests (the largest kernel) but in the accumulated overhead of many small operations.

9.8.7 The profiling feedback loop

Effective performance engineering follows an iterative cycle: *profile, diagnose, optimize, verify*. The verification step is critical and often skipped. After applying an optimization, reprofile to confirm that the targeted bottleneck was addressed and that no new bottleneck emerged. Performance optimization is a waterbed problem: fixing one bottleneck often exposes the next.

A common trap is optimizing based on microbenchmarks rather than end-to-end traces. A kernel that appears 2× faster in isolation may deliver only 5 percent improvement in end-to-end throughput if it was not the bottleneck, or if the surrounding code cannot take advantage of its speedup due to data dependencies. Always measure impact at the application level (tokens/second, step time, P99 latency) in addition to kernel-level metrics. The same measurement discipline governs rollout risk: a change that looks local can consume a global resource once it reaches production traffic.

War Story 9.1: The regex that saturated the edge

Context: Cloudflare's Web Application Firewall runs on edge machines that process HTTP and HTTPS traffic across the global network (Cloudflare 2019).

Failure mode: On July 2, 2019, a newly deployed managed WAF rule caused excessive CPU consumption on every HTTP/HTTPS-handling core across Cloudflare's network.

Consequence: Cloudflare reported losing most of its traffic until the team identified the WAF as the cause and used a global disable mechanism to restore service.

Systems lesson: Performance engineering includes algorithmic complexity and rollout design. A small rule change can become a global outage when it is deployed everywhere before bounded-runtime checks, canaries, and kill switches catch the cost. ML deployments follow the same pattern: a new model, a recalibrated threshold, or a retrained feature can ship a worst-case latency cliff into production that no unit test catches, which is why canarying, request shadowing, and per-request kill switches are first-class ML serving infrastructure.

Profiling itself can also perturb the system being measured. The PyTorch profiler adds approximately 10–20 percent overhead when recording full traces with memory profiling enabled. Nsight Systems adds less overhead but still affects scheduling. Profile warmup steps before active measurement, and discount the first few profiled iterations where JIT compilation or CUDA context initialization may dominate.

9.8.8 Profiling at scale

Profiling a single GPU is straightforward; profiling a distributed system with hundreds or thousands of GPUs introduces unique challenges. The volume of trace data grows linearly with the number of GPUs: a 5-second Nsight Systems trace for one GPU is approximately 500 MB; the same trace for 1,000 GPUs would be 500 GB, impractical to store or analyze.

Production systems address this through **hierarchical profiling**. At the top level, application-level metrics (MFU, throughput, step time) are collected continuously from every GPU with negligible overhead. These aggregate metrics detect when performance degrades. When a degradation is detected, **targeted profiling** is triggered on a representative subset of GPUs (typically one GPU per

pipeline stage, per data-parallel group) for a short window (a few training steps). The resulting traces are analyzed to identify the specific bottleneck.

Another approach is **statistical profiling**, where each GPU randomly samples a small fraction of its kernels for detailed timing. Over many training steps, the aggregated samples provide a statistically accurate picture of the kernel time distribution without the overhead of full tracing. The approach is analogous to the sampling profilers (like Linux `perf`) used in traditional systems engineering, adapted for the GPU context.

The most challenging profiling scenario is intermittent stragglers: GPUs that are occasionally slow due to thermal throttling, memory errors, or network congestion, but fast most of the time. These stragglers may not appear in a short profiling window but can reduce training throughput by 10–20 percent over hours. Detecting them requires continuous per-GPU step-time monitoring with statistical anomaly detection, a form of profiling infrastructure that operates at the monitoring layer rather than the kernel layer.

The profiling tools and techniques described in this section provide the measurement foundation for all optimization work. Without measurement, performance engineering degenerates into guesswork. With measurement, it becomes a systematic discipline guided by quantitative evidence.

9.9 Measurement at Scale

Optimizing a single node is a prerequisite, but the ultimate test of performance engineering is efficiency at fleet scale. When we move from 8 GPUs to 1,024 GPUs, new sources of overhead emerge that are invisible in local traces. A profiler can explain why one kernel stalls; it cannot by itself say whether thousands of accelerators are converting power, memory bandwidth, and network time into useful model progress. Measurement at scale requires shifting from kernel-level micro-benchmarks to global efficiency metrics that capture the interaction of computation, communication, and hardware variability.

9.9.1 The fleet efficiency metric

While Hardware Utilization reports how often GPUs are busy, it fails to distinguish between useful work and wasted cycles (such as activation recomputation or communication bubbles). Fleet measurement therefore needs a useful-work utilization metric. By focusing on the "useful" FLOPs required by the model architecture, this metric provides an invariant measure of system efficiency that remains comparable across different software stacks and parallelization strategies.



Definition 9.3: Model FLOPs utilization (MFU)

Model FLOPs Utilization (MFU), introduced in section 5.4, is the fraction of the hardware's theoretical peak throughput (R_{peak}) consumed by FLOPs that directly advance model training or inference, excluding overhead from recomputation, padding, and synchronization. The new content here is its fleet-scale behavior: how the per-node figure aggregates across thousands of accelerators and how the scaling tax pulls it down.

1. **Significance**: At fleet scale, MFU aggregates across all nodes: communication overhead, load imbalance, and pipeline bubbles each compound the utilization loss, so fleet MFU is consistently below single-node MFU. It is the primary diagnostic for whether hardware investment is translating into model progress, and a 1 percent improvement in MFU across a 10,000-GPU cluster reduces cost by the equivalent of 100 GPUs.
2. **Distinction**: Unlike Hardware Utilization (which reports how often the accelerator is "busy"), MFU reports how much of that activity contributes to Model Convergence or inference, excluding waste FLOPs from recomputation, padding, and gradient checkpointing overhead.
3. **Common pitfall**: A frequent misconception is that high GPU utilization implies high efficiency. A system can show 90 percent hardware utilization while achieving 30 percent MFU if it is wasting cycles on communication bubbles or inefficient kernel implementations, making MFU the correct metric for optimization decisions, not raw utilization.

With MFU defined as useful work rather than raw busyness, the next calculation shows how distributed overhead turns a strong single-node number into a weaker fleet-wide result.

Napkin Math 9.4: The scaling tax

Problem: A 70B parameter model achieves 65 percent MFU on a single 8-GPU node. When training scales to 128 GPUs, how much MFU does inter-node communication overhead consume?

- **Local node baseline**: A single 8-GPU node achieves 65 percent MFU.
- **Fleet performance**: At 128 GPUs, the step time increases to 245 ms, dropping MFU to 48 percent.

Math: Both MFU figures come from the same ratio, the useful FLOPs a step performs divided by the FLOPs the cluster could execute at peak in that step's wall time:

$$\text{MFU} = \frac{\text{FLOPs}_{\text{useful}}}{N \times R_{\text{peak}} \times t_{\text{step}}}$$

For the fleet case, a step performs 14.9 PFLOP of useful work, while 128 GPUs $\times$ 989 TFLOP/s per GPU $\times$ 245 ms of wall time could execute 31 PFLOP at peak; the quotient 14.9 PFLOP $\div$ 31 PFLOP $\approx$ 48 percent. The same substitution on the single node yields 65 percent, and the scaling tax is $1 -$ Fleet MFU/Local MFU, giving 26.2 percent.

Systems insight: The 26.2 percent scaling tax represents the cost of inter-node communication (InfiniBand latency) and synchronization barriers. In a healthy fleet, this tax should remain stable; a sudden increase in the scaling tax signals a scaling regression, typically caused by a misaligned parallelization strategy or a "gray failure" in the network fabric.

9.9.2 Detecting scaling regressions

At scale, the system is nonlinear. A code change that introduces a minor memory overhead on a single GPU can trigger a catastrophic performance collapse at 1,000 GPUs due to increased garbage collection pauses or exhausted InfiniBand credit buffers. Table 9.9 shows how tiered tests catch that collapse before it becomes a fleet incident.

Table 9.9: Scaling Regression Test Tiers: Fleet performance testing moves from small canaries to production baselines and gray-failure detection so that local changes are checked against distributed behavior.

Testing tier	What it measures	Regression caught
Small-scale canaries	Model behavior on 8 and 64 GPUs to establish a scaling efficiency curve	Code changes that look harmless on one GPU but bend the curve before full-fleet launch
Fleet baseline comparison	Every production run's MFU against the reference baseline for that model architecture	Architecture, compiler, or configuration changes that reduce useful fleet work
Gray failure detection	Distribution of step times across the fleet	Straggler nodes, such as one 10% slower node that can reduce synchronous data-parallel MFU by 10%

Those same measurements also explain why benchmark numbers cannot be copied directly into production capacity plans.

Systems Perspective 9.5: Benchmark vs. reality: The hero run tax

Industry benchmarks like MLPerf are often "hero runs"—highly tuned configurations where logging is disabled, safety checks are bypassed, and the hardware is freshly rebooted.

In production, achieved MFU typically sits 10–20 percent lower than these hero numbers. Essential operational overhead consumes the difference:

- Observability: Metrics collection and logging.
- Reliability: Checkpointing and health heartbeats.
- Entropy: Thermal throttling, memory fragmentation, and multi-tenant network noise.

When planning capacity, engineers must budget for the reality tax. If a benchmark projects 30 days of training, the production plan should assume 35–40 days.

Measurement without action is overhead. The optimization playbook translates node-level traces and fleet-wide MFU into a surgical sequence of interventions, each targeting the specific bottleneck that measurement identified.

9.10 The Optimization Playbook: A 70B LLM Case Study

Consider a raw, unoptimized 70-billion parameter PyTorch model that must serve 1,000 tokens per second in production by next week. The optimization sequence begins with baseline measurement, followed by roofline classification to identify the dominant bottleneck. Applying optimization techniques indiscriminately is ineffective. The optimization playbook requires a systematic, prioritized attack: first unblocking the memory wall, then fusing operators, and finally applying algorithmic techniques like speculative decoding in a specific, compounding sequence.

9.10.1 The diagnostic sequence

Optimization begins by measuring the whole workload before touching individual kernels. End-to-end throughput, an Nsight Systems trace, and Model FLOPs Utilization (MFU) from section 9.9 establish whether the system has substantial headroom. The next diagnostic move is roofline classification: compute arithmetic intensity for the dominant kernels and place them against the machine balance point. That classification determines the primary bottleneck, and the primary bottleneck determines which remedy should be tried first. For memory-bound, compute-bound, and communication-bound workloads, table 9.10 turns that decision into a compact map: identify the binding resource, use the typical setting as a sanity check, and try the interventions in order until a reprofiled trace shows the bottleneck has moved.

Table 9.10: Optimization Paths by Primary Bottleneck: Diagnostic map from bottleneck class to the ordered interventions that should be tried before the next profiling pass.

Primary bottleneck	Typical setting	Optimization path
Memory-bound	Inference, especially token decode	Reduce precision to raise effective bandwidth; fuse operators to eliminate intermediate HBM traffic; compile the graph to catch remaining fusion opportunities; then consider algorithmic changes such as speculative decoding or MoE if the bottleneck remains.
Compute-bound	Large-batch training	Ensure Tensor Cores are in use; apply graph compilation for kernel selection and memory planning; consider FP8 for 2× compute throughput; overlap communication so compute does not idle.
Communication-bound	Distributed training at scale	Overlap gradient communication with the backward pass; compress gradients or use reduced-precision communication; restructure pipeline schedules so stage-to-stage communication is hidden under useful compute; use topology-aware placement to minimize cross-node traffic.

The fourth step is to apply and verify. Implement the highest-impact optimization, reprofile, and verify improvement. Then iterate from the roofline classification step with the new profile, as the bottleneck may have shifted.

9.10.2 Combining techniques

Production systems combine techniques because no single intervention covers every term in the iron law. A highly optimized LLM serving system may use FlashAttention-2 or later FlashAttention-family

kernels to reduce attention memory traffic, INT4 weight quantization with GPTQ or AWQ to reduce HBM reads, and INT8 KV cache compression with per-channel scaling to increase feasible batch size. Compiler and runtime tools such as `torch.compile` or TensorRT then handle element-wise fusion and kernel selection.

The serving layer adds another set of controls. Speculative decoding reduces latency at low batch sizes, continuous batching refills the batch as requests finish (introduced in section 9.2.1), dynamic sequence grouping improves throughput, and tensor parallelism spreads the model across GPUs while overlapping AllReduce. These techniques belong together only when the profile shows that each one moves a currently binding term.

The speedups from these techniques are not additive; they interact in ways that demand careful sequencing. For instance, INT4 weight quantization reduces per-token HBM traffic by $4\times$, which might shift the bottleneck from memory-bound to compute-bound. Once compute-bound, further bandwidth optimizations (KV cache compression) yield diminishing returns, and compute optimizations (FP8 Tensor Cores) become the priority. This is why the iterative profile-optimize-verify loop is essential: the optimal combination depends on the specific model, hardware, and workload characteristics.

The interaction between optimizations creates a dependency graph that the performance engineer must navigate. Some combinations are synergistic: FlashAttention removes attention intermediates, and INT8 KV cache compression reduces KV cache memory, together freeing enough memory for larger batch sizes that transform the economics of serving. Other combinations are redundant: applying both CUDA Graphs and the `reduce-overhead` mode of `torch.compile` achieves the same result, since `reduce-overhead` internally uses CUDA Graphs. Still other combinations conflict: speculative decoding benefits most at small batch sizes (where decode is memory-bound), while many throughput optimizations work by increasing batch size. At large batch sizes, speculation adds overhead without proportional benefit.

A practical heuristic for sequencing optimizations is to apply the cheapest bottleneck-moving intervention before adding new algorithmic machinery. The first step depends on the measured bottleneck. Compiler-first is a common starting point for stable-shape workloads when memory is not obviously the binding constraint. For batch-1 70B decode, precision comes first because weight bandwidth and KV capacity dominate before graph overhead does.

1. Primary bottleneck fix: apply `torch.compile` when the trace shows launch overhead and stable shapes, or apply weight/KV precision first when the profile is memory-capacity or bandwidth bound.
2. FlashAttention-family kernels (library swap): Apply when attention materialization or attention HBM traffic remains visible in the profile.
3. Weight quantization (INT4/FP8, calibration required): Apply early for memory-bound serving, but validate quality before treating the speedup as usable.
4. KV cache compression (INT8, library support, $2\times$ cache reduction): Apply fourth. Enables larger batches.
5. Speculative decoding (requires draft model, engineering effort): Apply last, only if latency target not met. Most complex to deploy.

This ordering reflects the principle that passive optimizations (compiler, library swaps) should precede active ones (algorithmic changes, new model components). Each step is validated by reprofiling before proceeding to the next.

Checkpoint 9.3: Optimization strategy

Test your ability to design an optimization plan:

- ☐ **Top optimizations**: Given an LLM serving workload at batch size 1 that achieves 25 percent of peak bandwidth, identify the three most impactful optimizations and their expected interaction.
- ☐ **FP8 on comm-bound**: Explain why applying FP8 quantization to a workload that is already communication-bound provides no speedup.

- ☐ **Profiling workflow**: Describe the iterative profiling workflow and explain why verifying each optimization is as important as applying it.
- ☐ **Compile pitfalls**: Identify a scenario where applying `torch.compile` would degrade performance.

9.10.3 Case study: Optimizing a 70B LLM serving pipeline

To illustrate how the diagnostic sequence and combining principles work in practice, consider the task of optimizing a 70B parameter LLM for production serving. The target is a real-time chatbot application requiring time-to-first-token (TTFT) under 500 ms, inter-token latency (ITL) under 50 ms, and throughput of at least 1,000 tokens/second across the cluster. The model is deployed on a node of 8 H100 GPUs connected by NVLink.

9.10.3.1 Baseline measurement

The initial deployment uses FP16 weights, standard PyTorch eager execution, and tensor parallelism across 8 GPUs. The 70B model in FP16 requires 140 GB of weight storage, distributed as approximately 17.5 GB per GPU. Four baseline measurements reveal the optimization gap:

- **TTFT**: 1,200 ms (well above the 500 ms target)
- **ITL**: 85 ms (above the 50 ms target)
- **Throughput**: 280 tokens/second (below the 1,000 token/second target)
- **Maximum batch size**: 4 (limited by KV cache memory)

A Nsight Systems trace breaks down a single decode step at batch size 1 into five time categories:

- GEMM kernels: 38 percent of step time
- Attention (including KV cache reads): 24 percent of step time
- Element-wise operations (LayerNorm, GELU, residual): 14 percent of step time
- AllReduce communication (tensor parallelism): 12 percent of step time
- Kernel launch gaps and overhead: 12 percent of step time

The roofline analysis confirms that decode is deeply memory-bound, with arithmetic intensity approximately 1 FLOP/byte at batch size 1. The GPU achieves 2.7 TB/s effective bandwidth (80 percent of peak), indicating reasonable kernel-level efficiency but a fundamental algorithmic limitation.

9.10.3.2 Optimization round 1: Precision engineering

The first optimization targets the largest opportunity: reducing the bytes per weight read from HBM. Applying AWQ INT4 weight quantization reduces the per-GPU weight footprint from 17.5 GB to about 4.4 GB on an 8-GPU node. The raw weight-read traffic drops by 4×, and the effective bandwidth for weight reads roughly doubles once on-the-fly dequantization to FP16 is included.

Simultaneously, applying INT8 quantization to the KV cache reduces per-request cache size by 2×. The combined effect on memory budget is dramatic: each GPU now has approximately 75.6 GB available for KV cache, up from 62.5 GB. Under the 4096-token GQA cache calculation from section 9.4.3, this would permit much larger batches; in this deployment, the serving policy reserves memory for longer contexts, fragmentation headroom, and tail-latency protection. With that policy cap, the maximum admitted batch size increases from 4 to approximately 32.

Postoptimization metrics show that batch-1 ITL reaches 48 ms, meeting the 50 ms target, while throughput at batch size 32 reaches 720 tokens/second and remains below target.

The Nsight Systems trace shows that GEMM time decreased by approximately 45 percent due to reduced weight reads, but attention and element-wise operations remain unchanged. The bottleneck has partially shifted.

9.10.3.3 Optimization round 2: Operator fusion

The second round targets the 14 percent of step time consumed by element-wise operations and the 24 percent consumed by attention. Applying `torch.compile` with the `max-autotune` backend

fuses element-wise operations (GELU, LayerNorm, residual additions), reducing their contribution from 14 percent to approximately 4 percent of step time. Simultaneously, enabling FlashAttention-2 replaces the standard attention implementation, reducing attention HBM traffic by approximately 16× for the prefill phase.

For the decode phase, FlashAttention's impact is more modest because decode attention is dominated by KV cache reads rather than the $S \times S$ score matrix. However, the combination of INT8 KV cache compression and FlashAttention's efficient PagedAttention kernel reduces attention decode time by approximately 30 percent. The `reduce-overhead` mode in `torch.compile` wraps the decode step in a CUDA Graph, eliminating the 12 percent kernel launch overhead almost entirely.

Postoptimization metrics show TTFT at 380 ms, meeting the 500 ms target; ITL at 32 ms for batch size 1, well below the 50 ms target; and throughput at batch size 32 at 1,050 tokens/second, meeting the throughput target.

9.10.3.4 Optimization round 3: Speculative decoding

With the throughput target met, the team focuses on further reducing ITL for the best user experience. Speculative decoding with a 1.5B draft model (AWQ INT4 quantized to 0.75 GB) is deployed on the same GPUs. The draft model generates 5 candidate tokens in 4 ms (benefiting from the INT4 quantization applied in Round 1). The target model verifies the candidate block in approximately 32 ms, comparable to one optimized autoregressive decode step.

Under the standard geometric-acceptance model, where each draft token is independently accepted with probability p_{acc} and one bonus token always follows the last accepted one, the expected accepted tokens per round for k draft tokens is $(1 - p_{\text{acc}}^{k+1})/(1 - p_{\text{acc}})$. At $p_{\text{acc}} = 0.78$ and $k = 5$, this gives $(1 - 0.78^6)/(1 - 0.78) \approx 3.5$ tokens per round. The effective ITL becomes:

$$\text{ITL}_{\text{effective}} = \frac{4 + 32}{3.5} \approx 10.3 \text{ ms per token}$$

The result is a 3.1× improvement over the Round 2 ITL of 32 ms. However, speculative decoding interacts with batching. At batch size 32, the verification step is no longer "free" because the GPU is closer to compute saturation. The system therefore applies speculation only when the current batch size is below 16, falling back to standard autoregressive decoding at higher loads. This adaptive policy maintains both the latency benefit at low load and the throughput benefit at high load.

9.10.3.5 Lessons from the case study

This optimization journey illustrates the principle that optimization order matters because each step moves the bottleneck. Precision engineering (Round 1) was applied first because it yields the largest single improvement and enables subsequent optimizations by freeing memory for larger batch sizes. Fusion (Round 2) addressed the new bottleneck exposed by precision engineering. Speculative decoding (Round 3) provided latency improvement once the throughput target was met.

Each optimization changed the bottleneck. Before Round 1, the system was purely memory-bandwidth-bound. After INT4 quantization and batching, the system was partially compute-bound at large batch sizes. After fusion, kernel launch overhead was negligible, making the remaining bottleneck the fundamental memory-bandwidth limit for decode. Each optimization was validated by reprofiling to confirm the bottleneck shift.

The final system combines five distinct techniques: INT4 weight quantization, INT8 KV cache compression, FlashAttention-2, torch.compile with CUDA Graphs, and adaptive speculative decoding. These techniques are not independent; they interact. INT4 quantization enables larger batch sizes, which changes whether speculative decoding is profitable. FlashAttention's benefit depends on sequence length, which grows during generation. The performance engineer must reason about these interactions holistically, guided by profiling data at each stage.

The case study demonstrates how disparate optimizations compound sequentially, transforming an unusable prototype into a production-grade deployment. The path to these speedups, however, is lined with conventional wisdom that often proves disastrous at scale.

9.11 Fallacies and Pitfalls

A team upgrades their inference cluster from A100s to H100s, expecting a 3× latency reduction based on the spec sheet's teraFLOP/s rating, only to find their generative model barely runs 15

percent faster. The trap is pervasive: assuming that raw compute capacity dictates inference speed when the workload is entirely bound by memory bandwidth.

Fallacy: *More FLOP/s means faster inference.*

The roofline model demonstrates that most inference operations are memory-bound, not compute-bound. A GPU with 2× the peak FLOP/s but the same memory bandwidth will not generate tokens any faster for batch-1 LLM decode. The correct metric for memory-bound workloads is bandwidth, not FLOP/s. This fallacy leads organizations to purchase the most expensive compute hardware when a mid-range GPU with equivalent HBM bandwidth would deliver identical inference throughput.

Pitfall: *Planning FP8 adoption as an automatic training-time halving.*

FP8 doubles the peak TFLOP/s and doubles the effective memory bandwidth, but these gains are realized only for operations that are bottlenecked by compute or bandwidth at FP16. Element-wise operations like activation functions are already limited by kernel launch overhead, not by precision. Communication-bound distributed training steps gain nothing from reduced arithmetic precision if the communication volume (gradient sizes) is not also reduced. The actual speedup depends on the fraction of execution time spent in precision-sensitive operations.

Fallacy: *The largest kernel is the whole performance problem.*

The profiling case study in section 9.10.3 illustrates this pitfall. Engineers naturally focus on the single largest kernel, which is often the GEMM in a transformer layer. When the GEMM is already near-optimal, however, the remaining performance budget is distributed across dozens of smaller operations: normalization, activation, attention scoring, KV cache management, and kernel launch overhead. Collectively, these "small" operations can consume more than half of total execution time. Graph compilation and systematic fusion address this long tail more effectively than further GEMM optimization.

Pitfall: *Applying speculative decoding without considering batch dynamics.*

Speculative decoding excels at batch size 1, where decode is deeply memory-bound and the verification step is essentially "free" (the GPU has ample spare compute). At large batch sizes, decode approaches the compute-bound regime, and the verification step adds meaningful compute cost. Furthermore, the variable number of accepted tokens per request complicates continuous batching schedulers. In high-throughput serving scenarios with large batches, the overhead of speculation may outweigh its latency benefits.

Fallacy: *MoE expert count is a free scaling knob.*

Increasing the number of experts in an MoE model increases total parameters (capacity) without proportionally increasing per-token compute, which seems like a free lunch. Each additional expert, however, increases: (1) total memory requirements, requiring more GPUs; (2) AllToAll communication volume for expert routing; (3) load balancing difficulty, since the router must distribute tokens across more experts; and (4) training instability, as more experts compete for activation. Beyond approximately 64–256 experts, the system-level costs often outweigh the capacity benefits.

Pitfall: *Using graph compilers as a substitute for kernel analysis.*

Graph compilers have improved dramatically, but they remain limited by their cost models and fusion heuristics. FlashAttention required human insight to recognize that attention could be reformulated as a tiled algorithm with online softmax, an algorithmic insight beyond the scope of ordinary compiler rewrite rules. Similarly, speculative decoding and MoE routing require algorithmic innovation that compilers cannot discover. Compilers automate known optimizations; human engineers discover new ones.

Fallacy: *The lowest supported precision is always the best precision.*

Aggressive quantization (INT4 weights, INT4 KV cache, FP8 activations) can degrade model quality in ways that are difficult to detect with standard benchmarks but visible to users. Perplexity on a held-out dataset may change by less than 1 percent, but the model may produce subtly worse responses for edge cases, rare languages, or complex reasoning tasks. The correct approach is targeted quantization: apply the most aggressive precision to the least sensitive components (KV cache, intermediate activations) and preserve higher precision for the most sensitive (first and last layers, attention logits). Calibration on a representative dataset, followed by evaluation on diverse quality benchmarks, is essential before deploying any quantized model to production.

Pitfall: *Measuring throughput without measuring quality.*

A model serving system that generates 200 tokens/second is not twice as good as one generating 100 tokens/second if the first system achieves that throughput by using INT4 quantization that degrades answer quality by 15 percent. Performance metrics must always be reported alongside quality metrics. The correct optimization target is the Pareto frontier of throughput vs. quality, not throughput alone.

Fallacy: *A single profiling run is sufficient to characterize performance.*

ML system performance is nonstationary. GPU thermal throttling reduces clock speeds (and therefore FLOP/s) after sustained workloads, sometimes by 10–15 percent. Memory fragmentation accumulates over hours of serving, gradually reducing effective batch size. Network congestion varies with cluster-wide traffic patterns. A profiling run during a cold start may show different bottleneck patterns than one after hours of production serving. Reliable performance characterization requires profiling under realistic, sustained conditions, ideally sampling multiple times across a production run.

Pitfall: *Optimizing for average case while ignoring tail latency.*

A serving system may achieve excellent average inter-token latency (30 ms) while exhibiting P99 latency of 500 ms due to garbage collection pauses in the Python runtime, CUDA memory allocation stalls, or occasional AllReduce delays from network congestion. For interactive applications, the user experience is dominated by the worst case, not the average. Performance engineering for production systems must profile and optimize tail latency specifically, often through techniques orthogonal to the throughput optimizations in this chapter: preallocated memory pools, CUDA graph replay (which eliminates allocation variance), and priority scheduling for latency-sensitive requests.

9.12 Summary

Recognizing these pitfalls saves teams from wasting months optimizing the wrong layer of the stack. Performance engineering transforms a model that should be efficient into one that is, by attacking a fundamental bottleneck of accelerator-based ML systems: the memory wall. The Roofline Model provides the diagnostic framework, classifying operations as compute-bound or memory-bound based on their arithmetic intensity relative to the hardware's ridge point. For the NVIDIA H100, this ridge point is approximately 295.2 FLOP/byte at FP16, meaning most transformer operations fall in the memory-bound regime.

The chapter's techniques form an optimization stack rather than a menu. Operator fusion eliminates redundant HBM round-trips by combining sequences of operations into single kernels. FlashAttention is the canonical example, avoiding the quadratic attention intermediates that dominate long sequences through tiling and online softmax; in the 8K attention example, this is a 65× reduction in materialized attention state. Precision engineering then reduces the bytes in the trips that remain: FP8 formats improve effective bandwidth on H100 hardware, block-wise quantization protects the outlier features that defeat uniform quantization, and KV cache compression directly increases the batch sizes a serving node can admit.

Graph compilation moves those local transformations from hand-written kernels into the model graph. torch.compile/TorchInductor generates optimized Triton kernels from standard PyTorch code, XLA provides whole-program optimization for JAX/TPU workloads, and TensorRT specializes stable inference graphs. Communication-computation overlap applies the same bottleneck logic at the distributed boundary, hiding network latency when communication can run under useful compute. Speculative decoding and MoE go one level higher: instead of optimizing the same dense computation, they restructure the computation itself by obtaining more accepted tokens per target-model pass or by activating only the experts a token needs. System profiling closes the loop by showing which term dominates after each change, so the next intervention follows the new bottleneck rather than the old intuition.

The case study in section 9.10.3 demonstrated how these techniques compound in practice: INT4 quantization freed memory for larger batches, which changed the arithmetic intensity, which determined whether further bandwidth or compute optimizations were profitable. Each optimization shifted the bottleneck in a continuous displacement of overhead, requiring reprofiling and a new optimization decision. That profile-optimize-reprofile loop is the discipline of performance engineering: not making hardware faster, but making software match the physics of the hardware it

runs on. The memory wall is a physical constraint that grows wider with each hardware generation, so the engineer's response is to keep data close to compute, reduce precision to the minimum that preserves quality, and restructure algorithms to avoid unnecessary work. *Profiling is not merely the first step; it is every step.*

This iterative mindset also determines which skills endure as hardware evolves. Individual techniques may change as accelerator generations shift the ridge point of the Roofline Model and as architectures alter the dominant computational patterns. The fundamental discipline endures: measure the system, identify the binding constraint, apply the optimization that addresses that specific constraint, and then measure again. Engineers who internalize this cycle treat performance engineering as a continuous practice rather than a checklist, and that practice is what separates systems that merely run from systems that run efficiently at scale.

Key Takeaways: Match the software to the silicon

- **Bytes usually bind**: On H100-class accelerators, the chapter's roofline recap places most transformer work below the FP16 ridge point of 295.2 FLOP/byte, so useful speedups come first from reducing HBM traffic, keeping intermediates in SRAM, and spending compute only when it moves the active bottleneck.
- **Fusion makes locality real**: Operator fusion, CUDA graphs, and FlashAttention are not just kernel tricks; they remove launch overhead and materialized attention state. In the 8K example, tiled attention cuts stored intermediate state by 65×, turning memory pressure into usable throughput.
- **Precision buys bandwidth with risk**: FP8, INT8, and INT4 increase effective bandwidth and batch capacity only when outliers, scale factors, and quality checks are managed. Quantization is a systems contract between numerical format, kernel implementation, serving memory, and acceptable model behavior.
- **Algorithms can move the roofline**: Speculative decoding and mixture-of-experts change how much useful work each target-model pass performs, but they introduce acceptance-rate, routing, AllToAll, and load-balancing constraints. The win is real only after communication and scheduler costs are measured.
- **Profiling is every step**: The 70B case study shows optimization as bottleneck displacement: INT4 changes batch size, batch size changes arithmetic intensity, and the next limit moves. Fleet performance engineering means measure, optimize the binding term, then measure again before believing the speedup.

Hardware is sold by its *peak*. It is paid for by what it *sustains*, and the gap between the two is almost entirely data movement. Every technique here, FlashAttention most visibly, narrows that gap by keeping bytes off the slow path between memory and compute. This is the memory wall again, the same physical limit met one layer up in the stack: in the hardware it set the ceiling, and here it is the thing the software spends all its effort trying to reach. The accelerator's advertised number was always real; this chapter is how much of it the fleet actually gets to keep.

What's Next: From optimization to serving

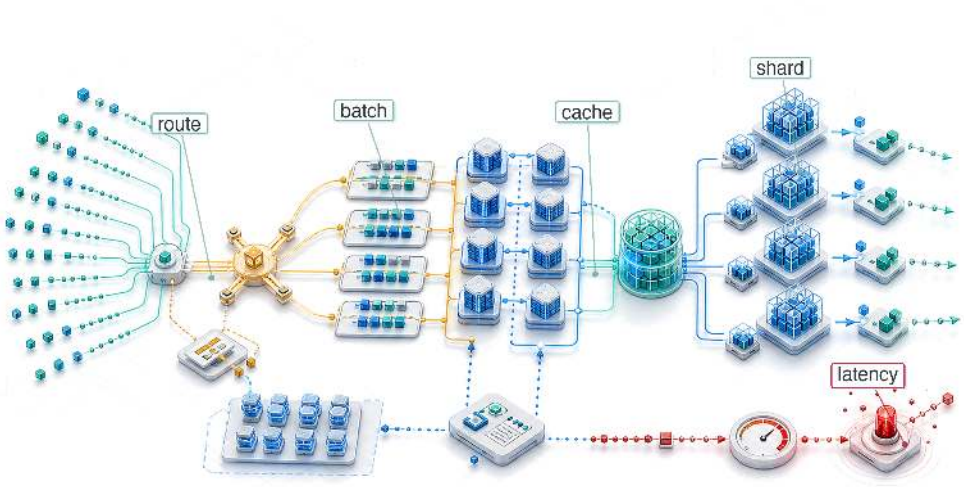
Operator fusion, precision reduction, graph compilation, and algorithmic innovations like speculative decoding and MoE extract maximum computational efficiency from a single model executing on bare metal. Optimizing an isolated forward pass, however, is merely the precursor to deployment. In production, the same model must handle bursts of concurrent requests, share accelerators with other work, and preserve latency budgets while request lengths and arrival rates change. Chapter 10 transitions from local execution mechanics to the architecture of robust serving systems. The low-level optimizations developed here become the primitives that make high-availability serving possible: they reduce per-request memory, increase useful batch size, and give the scheduler more room to protect tail latency.

Research Questions: For further inquiry

- How should roofline and profiler evidence decide whether to optimize memory traffic, compute, launch overhead, or communication?
- How do distributed-training and collective-communication costs limit the value of local kernel optimizations?
- When does lower precision improve effective bandwidth without violating quality or latency requirements?
- How should fusion, tiling, FlashAttention, and CUDA Graphs be evaluated when each shifts the next bottleneck?

10

Inference at Scale

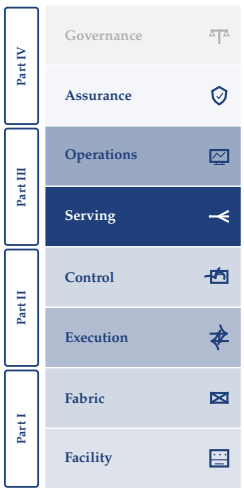


10.1 The Economics and Architecture of Inference
10.2 Serving Architecture Dimensions
10.3 Batching Strategies at Scale
10.4 Memory and Decode-Time Management
10.5 Model Sharding for Inference
10.6 Load Balancing and Request Routing
10.7 Multi-Tenancy and Isolation
10.8 Autoscaling and Global Infrastructure
10.9 Weight Quantization for Serving
10.10 Case Studies
10.11 Fallacies and Pitfalls
10.12 Summary

Purpose

Why does inference cost eventually dwarf training cost, and what does this mean for system design?

Training a model is a one-time expense; serving it is perpetual. A model trained once for millions of dollars may serve billions of requests over its lifetime, and every request consumes compute, memory, network, and energy budget. The serving cost dominance law captures this economic reality: for a successful production model, ongoing inference operating expense can exceed the one-time training capital expenditure by orders of magnitude. That cost structure changes the architecture: optimizations that shave milliseconds from inference latency or percentage points from accelerator utilization compound across billions of requests into substantial savings or costs. Performance engineering supplied the local toolkit—fusion, precision, compilation, and algorithmic changes such as speculative decoding—and this chapter applies those tools at serving scale, where batching, sharding, routing, autoscaling, and isolation must preserve strict latency budgets under live traffic. Reliability requirements intensify as well, because downtime during serving means lost revenue and broken user experiences, not merely delayed experiments. Inference at scale is where ML systems succeed or fail economically. Serving the fleet is a continuous exercise in C^3 management: batching compute for throughput while holding coordination overhead within strict tail-latency budgets.



Learning Objectives

- Quantify lifetime serving cost from request volume, token mix, accelerator price, utilization, and latency targets
- Select batching and scheduling policies for vision, LLM, recommender, and streaming workloads
- Analyze attention-cache capacity, fragmentation, and decode-time policies for memory-bound language-model serving
- Compare sharding, disaggregation, and quantized serving designs using communication, memory, and quality constraints
- Evaluate routing, load balancing, and isolation controls against tail latency and noisy-neighbor risks
- Design autoscaling and multi-region failover policies that balance cold starts, cost, and SLO compliance
- Synthesize serving architectures that manage C^3 trade-offs across request, replica, service, and platform layers

10.1 The Economics and Architecture of Inference

Imagine a language model that costs $5 million to train over two months. Once deployed to a global user base serving thousands of queries per second, that same model burns through $5 million in inference compute every single week. The economics of inference force a radical architectural shift: we are no longer optimizing a temporary batch job; we are operating a perpetual, latency-sensitive factory.

Operator fusion, precision engineering, and graph compilation can maximize throughput on individual forward passes. The next challenge emerges when a single optimized model must serve thousands of concurrent users across a globally distributed fleet. Single-machine inference optimization, including batching, caching, model optimization, and hardware acceleration, provides the building blocks. Distributed approaches become necessary when those techniques reach their limits.

Distributed inference systems must solve problems that do not exist at single-machine scale. Load balancing[1] becomes critical when requests must be distributed across hundreds of GPU instances while maintaining latency guarantees. Request routing must account for model-specific characteristics: recommendation systems with trillion-parameter embedding tables require different placement strategies than large language models that generate responses token by token. Autoscaling must anticipate demand fluctuations that can change request volume by orders of magnitude within minutes while maintaining latency bounds users expect.

The economics of inference at scale differ fundamentally from training economics. Training costs are dominated by compute time and can be amortized over the lifetime of the resulting model. Inference costs are directly tied to user traffic and revenue. An e-commerce recommendation system might serve millions of requests per second during peak shopping periods, with each request contributing directly to potential revenue. The cost of overprovisioning during quiet periods or underprovisioning during peaks translates immediately to business impact. Inference efficiency becomes a first-order concern in ways that training efficiency rarely achieves. Building such a service requires three linked decisions: when distribution becomes necessary, how the architecture preserves latency bounds under varying load, and how resource utilization stays high enough that serving cost does not erode the value the model creates.

10.1.1 When single-machine serving is insufficient

Three distinct signals indicate when distributed inference becomes necessary rather than merely optional. Table 10.1 categorizes these triggers by constraint type and corresponding strategy.

The first signal is memory exhaustion, which occurs when model parameters, key-value caches, or embedding tables exceed single-device capacity. A single NVIDIA H100 GPU provides 80 GB of

1 **Load Balancing (Inference)**: GPU inference requests range from 10 ms to 30+ seconds with high variance, unlike web requests that complete in uniform milliseconds. This variance invalidates round-robin and random assignment, forcing queue-depth-aware routing strategies that add monitoring overhead but prevent tail-latency blowups across heterogeneous GPU fleets.

2 **HBM3 (High Bandwidth Memory 3)**: 3D-stacked DRAM delivering 3.35 TB/s on the H100 vs. 2.04 TB/s for HBM2e on the A100. Because large language model (LLM) decode is memory-bandwidth-bound, this improvement translates directly into higher tokens-per-second and larger feasible batch sizes, making HBM generation one major variable in inference cost-per-token.

HBM3[2] memory; section G.1 records the provenance of this capacity figure and the other canonical hardware constants used throughout the analysis that follows. GPT-4-class models dwarf that capacity: the public ~1.8T-parameter mixture-of-experts estimate implies ~3.5 TB just for weights in FP16 precision, forcing distribution across multiple GPUs regardless of throughput requirements. Recommendation systems with trillion-parameter embedding tables face similar constraints: Meta's DLRM architecture[3] stores embedding tables that require multiple terabytes of memory.

Beyond memory constraints, throughput limitations emerge when request volume exceeds single-machine capacity even with optimal batching. Consider a recommendation system serving 100,000 queries per second with a 10 ms latency budget. If single-machine throughput peaks at 10,000 QPS, no amount of optimization on that machine can satisfy demand. Horizontal scaling across multiple replicas becomes mandatory.

Finally, strict latency requirements drive distribution when model execution time exceeds latency budgets even at batch size one. Large language models generating responses token by token face this constraint acutely. A 70-billion parameter model requires approximately 140 GB of memory in FP16, exceeding a single 80 GB H100-class GPU before KV cache is considered. Even when quantization or larger-memory hardware makes a single-replica configuration possible, decode throughput is often limited by memory bandwidth. Sharding the model across multiple GPUs enables parallel computation that reduces time-to-first-token below acceptable thresholds.

3 **DLRM**: Meta's 2019 reference architecture separates dense features – GPU-bound multilayer perceptrons (MLPs) – from sparse features (CPU-bound embedding lookups), creating a hybrid serving topology (Naumov et al. 2019). Production recommendation characterizations show why this structure matters for serving: embedding-heavy recommendation models create memory-access and capacity constraints that differ from dense CNN, RNN, or LLM inference (Gupta et al. 2020).

Table 10.1: Triggers for Distributed Inference: Each constraint type indicates different distribution strategies. Memory constraints require model sharding; throughput constraints require replication; latency constraints may require either depending on whether the bottleneck is compute or memory bandwidth.

Constraint	Single-Machine Limit	Example Workload	Distribution Strategy
Memory	80 GB (H100)	GPT-4 (~3.5 TB FP16)	Tensor/pipeline parallelism
Throughput	~10K QPS (vision)	100K QPS RecSys	Horizontal replication
Latency	Model execution time	500 ms LLM TTFT	Model sharding

Memory, throughput, and latency triggers all become user-visible once an interactive endpoint starts streaming. **Time to First Token (TTFT)**[4] determines when the user sees a response begin, while **Time Per Output Token (TPOT)** determines whether the rest of the response arrives at a readable pace; this is why inference reverses many of the optimization priorities from training.

4 **TTFT vs. TPOT**: Time to First Token (prefill) determines the perceived responsiveness; Time Per Output Token (decode) determines the perceived reading speed. Humans read at roughly 5–10 tokens/sec; TPOT above roughly 100–200 ms can begin to feel slower than natural reading, while many products target lower TPOT for smooth streaming. This dual service level agreement (SLA) requirement forces schedulers to prioritize decode tokens over new prefill prompts.

Checkpoint 10.1: Distribution strategy selection

Verify your understanding of when to move from single-machine to distributed inference:

- ☐ **Shard vs. replicate**: For a 13B model (26 GB weights) that fits on a single A100, select model sharding or horizontal replication when throughput requirements double.
- ☐ **Latency floor**: For a real-time speech-to-text service where inter-token latency is the primary bottleneck, determine whether data-parallel replication reduces this latency.
- ☐ **Mandatory sharding**: Explain why model sharding is mandatory for a 175B model on 80 GB GPUs, regardless of the request volume.
- ☐ **Latency trigger**: Identify when a latency target forces distribution and when added network, serialization, coordination, or queuing overhead can erase the benefit.

10.1.2 The fundamental inversion: Training vs. inference

The contrast between training and inference optimization extends beyond the basic throughput-vs.-latency distinction. Training optimizes for samples processed per hour and tolerates latency variations. Inference optimizes for response time and must meet strict latency bounds. At scale, this inversion manifests in system architecture, resource allocation, and operational priorities. Table 10.2 details six key system aspects where these differences emerge.

Table 10.2: Training vs. Inference System Requirements: The fundamental inversion from throughput to latency optimization ripples through every aspect of system design.

Aspect	Distributed Training	Distributed Inference
Primary metric	Throughput (samples/hour)	Latency (P99 ms)
Acceptable variance	Hours	Milliseconds
State management	Checkpoints (periodic)	Session state (continuous)
Batch formation	Large, controlled	Request-driven, variable
Failure tolerance	Restart from checkpoint	Redirect without user impact
Cost structure	Fixed duration, variable rate	Variable duration, fixed SLO

Training tolerates substantial latency variance because the optimization target is aggregate progress over hours or days. A training iteration that takes 2 seconds instead of the usual 1 second represents acceptable variation. An inference request that takes 2 seconds instead of 100 milliseconds represents catastrophic failure, potentially causing user abandonment or cascading timeouts in dependent services.

State management differs fundamentally. Training maintains model state (parameters, optimizer states) that evolves gradually and can be captured in periodic checkpoints. Inference often maintains session state (conversation history, key-value caches, user context) that must be preserved across requests and cannot tolerate the staleness that checkpoint-based recovery would introduce.

Failure handling diverges correspondingly. Training failures trigger checkpoint restoration and continuation, with minutes of lost progress being acceptable. Inference failures must be invisible to users. Requests redirect to healthy replicas, degraded results substitute for unavailable models, and SLOs must be maintained despite infrastructure instability.

10.1.3 The serving tax: Overhead of distribution

Distributing inference across multiple machines introduces overhead absent from single-machine serving. This "serving tax" must be understood and budgeted within latency constraints.

Network communication adds latency for every cross-machine interaction. Within a data center, network round-trip times range from 50–500 microseconds depending on topology and congestion. For model sharding that requires synchronization between GPUs on different machines, each synchronization point adds this overhead. A model sharded across 8 machines with 4 synchronization points per inference adds 200 microseconds to 2 milliseconds of network latency. Section D.1 develops the α-β model that decomposes each such transfer into a fixed startup latency plus a bandwidth-dependent term, giving the quantitative framework for budgeting these synchronization costs against a latency SLO.

Serialization overhead compounds the problem by converting in-memory tensors to network-transmittable formats. Traditional serializers like Protocol Buffers, JSON, and Python's `pickle` perform a parse-allocate-copy cycle on every transfer, and a 1 GB activation tensor takes approximately 100 milliseconds to serialize and deserialize through such formats. Zero-copy alternatives like FlatBuffers and Cap'n Proto[5] sidestep most of this cost by accessing wire-format data in place, but they impose stricter schema layout requirements that not every production stack adopts.

Load balancer latency adds another layer. Requests must be routed to appropriate replicas, which requires examining request metadata, consulting routing tables, and forwarding to selected backends. Well-optimized load balancers add 100–500 microseconds; poorly configured ones can add milliseconds.

Coordination overhead emerges when requests require fan-out to multiple services. A recommendation system that queries a user model, item model, and ranking model in parallel must coordinate these queries and aggregate results. The coordination logic itself consumes CPU cycles and introduces latency variation.

The total serving tax often consumes 10–30 percent of the latency budget in distributed systems, as equation 10.1 shows:

$$T_{\text{total}} = T_{\text{compute}} + T_{\text{network}} + T_{\text{serialization}} + T_{\text{coordination}} + T_{\text{queuing}} \tag{10.1}$$

[5] **Zero-Copy Serialization**: FlatBuffers and Cap'n Proto access wire-format data in place, eliminating the parse-allocate-copy cycle of Protocol Buffers or JavaScript Object Notation (JSON). For distributed inference with large activation tensors, this drops per-hop serialization overhead from milliseconds to microseconds, reclaiming latency budget that would otherwise consume 5–10 percent of a tight SLO.

Minimizing this tax requires co-locating communicating components, using high-bandwidth interconnects, and designing communication patterns that minimize round trips.

10.1.4 Serving cost can dominate training cost

The serving tax quantified above consumes a fraction of the latency budget per request. The true economic impact of inference emerges when we consider cost over a model's entire operational lifetime. The **serving cost dominance law** (principle 15) states that serving cost can dominate training cost by orders of magnitude because training is a *one-time capital expenditure* (CapEx) while serving is a *continuous operational expenditure* (OpEx) that scales with user growth. A quick cost calculation makes the multiplier concrete.



Lifetime serving cost dwarfs one-time training.

> **Napkin Math 10.1: The serving cost multiplier**
>
> **Problem**: A team has spent $2,000,000 training a 70B parameter model and now serves it to 1,000,000 daily active users (DAU), each making 50 requests/day. Is training or serving the dominant cost over 1 year?
>
> **Math**:
>
> 1. **Training Cost**: $2,000,000 (One-time).
> 2. **Metric**: Annual inference volume is 10^6 users $\times$ 50 reqs $\times$ 365 days $\approx$ 18.25B requests/year.
> 3. **Cost per Request**: Assume a 70B model on H100 costs $\approx$ $0.001/request (input + output).
> 4. **Annual Serving Cost**: 18.25B requests $\times$ $0.001/request = $18,250,000.
>
> **Systems insight**: Serving costs 9.1× more than training in just the first year. A 10 percent serving-cost or latency-linked efficiency improvement saves about $1.8M in the first year, nearly paying for the original training run.

The total cost of operating a model comprises training cost (a one-time expense) and serving cost (an ongoing expense), as equation 10.2 shows:

$$C_{\text{total}} = C_{\text{training}} + C_{\text{serving}} \times T_{\text{deployment}} \times Q_{\text{rate}} \tag{10.2}$$

where C_{training} is the one-time cost to train the model, C_{serving} is the cost per query served, $T_{\text{deployment}}$ is the deployment duration in appropriate time units, and Q_{rate} is the query rate.

The same leverage holds across application categories with different cost structures. A recommendation model (DLRM) trained for $12,000 ($3000 hardware for 1,000 GPU-hours at $3/GPU-hour, plus 3× additional engineering cost for data and experimentation) and served at 10,000 QPS for 730 days accumulates 6.31×10^{11} lifetime queries, which at $10/million queries costs $6,307,200 to serve—a 525.6× leverage on serving optimization over training optimization. The cost dominance ratio varies by application. Table 10.3 quantifies this disparity:

Table 10.3: Training vs. Serving Cost Ratios: High-QPS applications like recommendation systems show the most extreme cost dominance of serving over training.

Application	Training Cost	Annual Serving Cost	Ratio
Recommendation (high QPS)	$10K–$100K	$1M–$10M	100–1000×
Search ranking	$100K–$1M	$10M–$100M	100–1000×
LLM API	$1M–$100M	$10M–$1B	10–100×
Internal analytics	$1K–$10K	$10K–$100K	10–100×

The cost structure motivates serving optimization: every percentage point of efficiency improvement yields ongoing cost reduction over the model's operational lifetime. Figure 10.1 illustrates why optimization matters by showing how the gap between naive and optimized serving determines whether an inference service is profitable or hemorrhaging money.

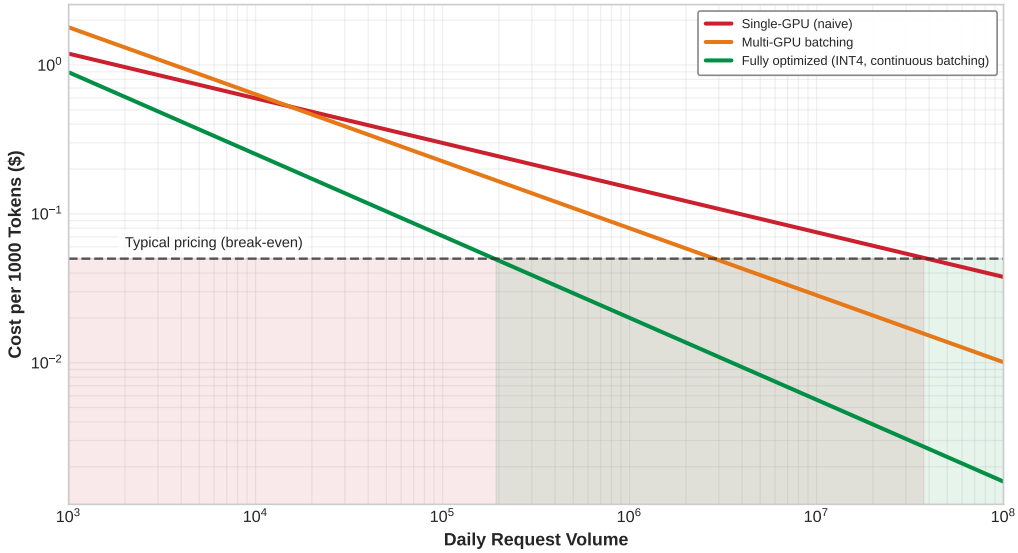


Figure 10.1: The Serving Cost Curve: Cost per 1000 tokens decreases with scale and optimization. Single-GPU serving reaches break-even only at very high volume. Multi-GPU batching breaks even at moderate volume. Fully optimized serving (INT4, continuous batching, speculative decoding) achieves profitability at much lower volumes, fundamentally changing the economics of inference.

The cost ratios in table 10.3 tell a *static* story, but the *dynamics* of cost accumulation reveal an even more striking pattern. Figure 10.2 plots cumulative total deployment cost over a 36-month window for three representative scenarios: a small model with low traffic, a 70B LLM serving one million daily active users, and a recommendation system serving 100 million users. The vertical crossover markers show when cumulative serving spend equals the original training cost; because the plotted curves include both training and serving, the total-cost curve is at roughly twice the training baseline at that marker. For recommendation systems, serving dominates within days; for high-traffic LLMs, within roughly six weeks. Only small models with expensive training and low traffic see training cost dominate for extended periods. This temporal view reinforces why inference optimization deserves early engineering attention at production scale. For generative LLM services, serving-specific designs can translate directly into throughput, cost, and power gains (Patel et al. 2024).

10.1.5 The inference landscape: Beyond LLMs

A common misconception frames inference at scale as synonymous with LLM serving. Large language models present distinctive challenges and attract attention, but they are only one part of production inference. Appropriate technique selection requires understanding the full inference landscape. In large consumer platforms, recommendation and ranking workloads can dominate AI inference cycles and capacity, even though they are not the only user-visible model class (Gupta et al. 2020; Hazelwood et al. 2018). Vision and image processing, NLP and LLM workloads, fraud detection, ads, and other classification tasks fill out the remainder. Table 10.4 breaks down these model types qualitatively by serving pressure and optimization challenge, and the comparison shows that each class binds on a different constraint: recommendation on embedding lookup, vision on batch efficiency, LLMs on memory bandwidth, and speech on sequential decode. No single optimization serves all four.

Recommendation systems often dominate high-volume consumer inference because they serve predictions for many user interactions. Every page load, scroll, or click can trigger ranking or retrieval inference. A user browsing an e-commerce site might generate many recommendation requests in a single session. In contrast, LLM queries typically require explicit user action and occur less frequently.

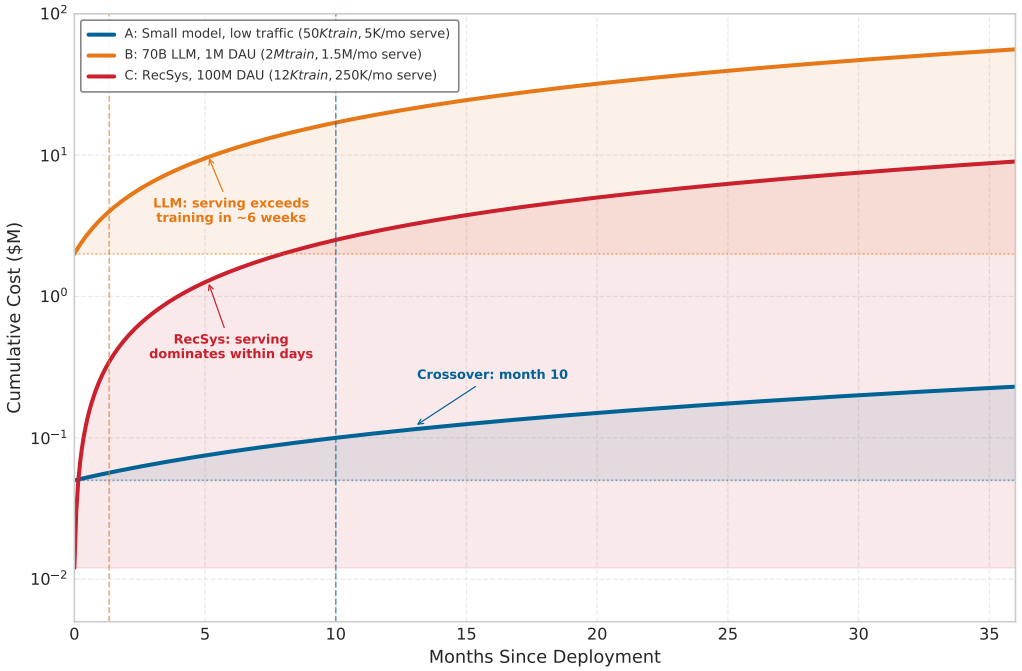


Figure 10.2: The Serving Cost Crossover: Cumulative total deployment cost (initial training plus ongoing serving) for three scenarios. Vertical markers indicate when the serving portion equals the one-time training cost. For high-traffic LLMs, serving cost matches training cost within weeks. For recommendation systems at 100M daily active users, serving dominates within days. Only low-traffic, expensive-to-train models see training cost dominate for extended periods.

The distribution has direct implications for technique selection. Recommendation systems have driven important production inference innovations: dynamic batching, embedding sharding, feature store architectures, and low-latency serving were all developed primarily for recommendation workloads (Naumov et al. 2019; Gupta et al. 2020). LLM-specific techniques like continuous batching and KV cache management address a different slice of production inference. Text-to-image systems such as DALL-E provide one multimodal model example (Ramesh et al. 2021), but multimodal serving volume and latency targets remain product-specific.

Table 10.4: Production Inference Landscape: Different model types create different serving pressure, latency requirements, and optimization challenges. The table is a qualitative workload map rather than a universal traffic-share distribution. Technique selection must match the specific workload.

Model Type	Serving Pressure	Latency Target	Key Challenge
Recommendation/ranking	Very high in consumer platforms	<10 ms P99	Embedding lookup
Vision (CNN)	Workload-dependent	20–100 ms	Batch efficiency
LLM	High cost per request	100 ms–10s	Memory bandwidth
Speech/Audio	Stream-driven	Real-time	Sequential decode
Multimodal/text-to-image	Emerging and product-specific	Varies	Cross-modal coordination

10.1.6 The serving hierarchy

The optimization techniques organize into a **serving hierarchy**, analogous to the memory hierarchy in computer architecture. Each level owns a different bottleneck, so an optimization that helps one level can leave another unchanged. The request level follows one request through preprocessing, batching, caching, and model execution, where the target is the latency a user sees. The replica level looks inside one model instance, where GPU utilization, memory management, kernel efficiency,

and model optimization determine how much useful throughput that replica can deliver before it saturates. The service level then works across many replicas of the same model, using load balancing, request routing, and autoscaling to turn individual replicas into aggregate capacity. The platform level works across services and tenants, where resource allocation, multi-tenancy, scheduling, and placement decide whether a shared serving fleet remains efficient without allowing one workload to degrade another.

The hierarchy matters because each level changes a different metric and fails at a different boundary. A related deployment stack appears in figure 10.3, showing how requests pass through edge, routing, and model-serving infrastructure in production.

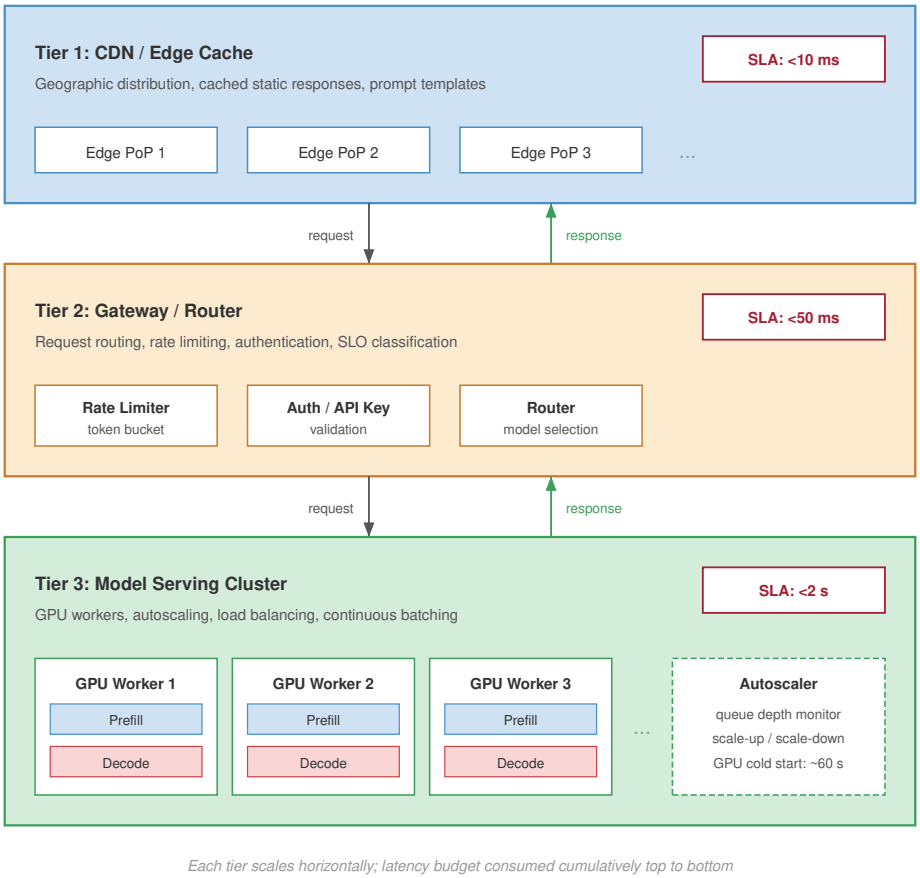


Figure 10.3: Serving Deployment Stack: A three-tier serving stack with cumulative latency budget from top to bottom. Tier 1 (CDN/Edge Cache) handles geographic distribution and cached static responses (SLA <10 ms). Tier 2 (Gateway/Router) handles request routing, rate limiting, and authentication (SLA <50 ms). Tier 3 (Model Serving Cluster) runs GPU workers with autoscaling and continuous batching (SLA <2 s).

Each level has distinct optimization levers, and table 10.5 shows why the lever is level-specific: request-level changes reduce per-request latency, replica-level changes raise utilization, service-level changes add aggregate capacity, and platform-level changes improve fleet efficiency across tenants. Optimizing the wrong level moves the wrong metric.

Table 10.5: Serving Hierarchy Optimization Targets: Each level of the hierarchy addresses different metrics with different techniques.

Level	Optimization Target	Key Techniques
Request	Per-request latency	Dynamic batching, caching, prefetching
Replica	Throughput, utilization	Memory optimization, kernel fusion
Service	Aggregate capacity	Load balancing, routing, autoscaling
Platform	Resource efficiency	Multi-tenancy, scheduling, placement

Checkpoint 10.2: The serving hierarchy

Verify your understanding of where specific optimizations sit within the serving hierarchy:

- ☐ **Memory level**: Identify the hierarchy level responsible for managing GPU memory waste to increase concurrent request capacity.
- ☐ **Caching level**: Identify the hierarchy level responsible for reducing KV-cache waste when request lengths vary.
- ☐ **Load balancing**: Identify the primary optimization target of Service-level load-balancing algorithms like Power-of-Two-Choices.
- ☐ **Isolation level**: Evaluate the claim that multi-tenancy and resource isolation are primarily Request-level concerns.

The hierarchy guides the rest of the design while allowing a few techniques to cross levels. Batching, KV-cache layout, and decode-time optimizations start at the request level. Quantization, adapter state, and model sharding change the replica's memory and compute budget. Disaggregated serving, load balancing, and autoscaling coordinate replicas into a service, while multi-tenancy and resource isolation govern the platform. Quantization appears across the hierarchy because it is a representation-level lever that changes memory, bandwidth, and cost budgets wherever model state or KV state resides.

10.2 Serving Architecture Dimensions

A recommendation system that processes 100,000 embedding lookups per second across a sharded feature store requires a fundamentally different serving architecture than a 70-billion parameter language model generating tokens one at a time. The difference is not merely one of framework choice; it reflects distinct constraints in batching strategy, memory management, scheduling policy, and deployment topology. Rather than catalog specific tools, this section identifies the architectural dimensions that distinguish serving systems and the constraints that drive each design decision. Specific frameworks serve as examples of these principles, not as the subject of study.

10.2.1 Batching strategy: The throughput-latency trade-off

The most consequential architectural decision in a serving system is how it forms batches from incoming requests. This choice determines the fundamental throughput-latency operating point.

Static batching collects a fixed number of requests before dispatching them to the accelerator. For vision models processing fixed-size inputs, this approach maximizes GPU utilization because all requests in the batch execute identical computation graphs with predictable memory access patterns. A ResNet-50 inference server can batch 32 or 64 images with near-linear throughput scaling because the batch amortizes fixed launch overhead and reuses resident weights across many inputs.

Autoregressive language models break this assumption. Each request generates a different number of output tokens, so static batching forces all requests to wait for the longest generation in the batch and can leave accelerator capacity idle after shorter requests finish. Continuous batching[6] solves this by allowing requests to enter and exit the batch at each decoding step, an iteration-level scheduling approach introduced by Orca (Yu et al. 2022). When one request finishes generation, a new request

6 **vLLM (Virtual LLM)**: The name signals its core design: applying OS-style virtual memory to KV cache management, decoupling logical sequence addresses from physical GPU memory. Developed at UC Berkeley (2023), vLLM achieved 2–4× throughput over FasterTransformer and up to 24× over HuggingFace Transformers by eliminating the 60–80 percent memory fragmentation that capped batch sizes in prior serving systems (Kwon et al. 2023).

immediately fills its slot. Systems like vLLM combine this scheduling style with PagedAttention-based KV-cache management, achieving 2–4× throughput gains over prior LLM serving systems by reducing memory waste (Kwon et al. 2023).

Recommendation systems introduce a third pattern: **feature-parallel batching**. Because the bottleneck is distributed embedding lookup rather than dense matrix multiplication, these systems batch requests by feature type and shard the batch across embedding servers. The dense MLP computation that follows can then operate on prefetched, prebatched feature vectors. The batching strategies section (section 10.3) develops these approaches in full quantitative detail.

10.2.2 Memory management: From preallocation to paging

Serving systems differ fundamentally in how they manage accelerator memory, and this difference determines the maximum concurrent request capacity.

Definition 10.1: KV cache

KV Cache is a per-request LLM inference memory buffer that stores the Key and Value attention tensors of all previously generated tokens so the next token can be produced without recomputing the full attention over the prefix.

1. **Significance**: It reduces per-token attention compute from $O(n^2)$ to $O(n)$ in the sequence length, but its footprint grows linearly with both context length and batch size, often consuming more HBM than the model weights themselves and becoming the binding constraint on serving capacity. Section F.1.2 derives the fundamental sizing math.
2. **Distinction**: Unlike model weights (which are shared across all requests and statically allocated), the KV cache is request-private and dynamically sized; each concurrent user pays a separate, sequence-length-proportional memory tax that the scheduler must budget for.
3. **Common pitfall**: A frequent misconception is that reducing model size proportionally reduces serving memory. In practice, the KV cache dominates HBM at long contexts, so a quantized 70B model can still be capacity-bound at batch sizes the weights alone would easily admit.

Preallocated memory management reserves a fixed memory budget per request at admission time based on the maximum possible output length. For a model supporting 4,096-token outputs, every request reserves memory for 4,096 tokens regardless of whether it generates 50 or 4,000. This approach is simple and predictable but wastes memory proportional to the gap between maximum and actual output lengths. In practice, 60–80 percent of reserved KV cache memory goes unused.

Paged memory management, inspired by operating system virtual memory, allocates memory in fixed-size blocks (pages) and maps logical sequence positions to physical memory locations through a block table. As a request generates tokens, new physical blocks are allocated on demand. When generation completes, blocks return to the free pool immediately. This approach, exemplified by PagedAttention (covered in section 10.4.3), achieves near-100 percent memory utilization by eliminating both internal fragmentation (partially filled preallocations) and external fragmentation (unusable gaps between allocations).

The memory management strategy directly determines batch capacity: paged systems can serve 2–4× more concurrent requests than preallocated systems on the same hardware, because they reclaim memory that preallocation wastes. For a service with a fixed GPU budget, this translates directly to 2–4× lower cost per request.

10.2.3 Scheduling policy: FCFS, preemptive, and priority-aware

The scheduling policy determines which requests receive GPU time and in what order. This decision becomes critical when request mix is heterogeneous, with some requests requiring 50 tokens of output and others requiring 4,000. **First-come-first-served (FCFS) scheduling** processes requests in arrival order. FCFS is fair and simple but suffers from head-of-line blocking: a single long-generation

request delays all subsequent requests. For workloads with high output-length variance, FCFS produces poor tail latency.

Preemptive scheduling allows the system to pause a long-running request and swap its KV cache to CPU memory (or discard it for later recomputation) to make room for shorter, higher-priority requests. The cost of preemption is the swap overhead (transferring KV cache between GPU and CPU memory) or the recomputation cost (re-running the prefill phase when the preempted request resumes). Production systems typically preempt when a request has consumed more than 2× the median generation length.

Priority-aware scheduling assigns different service classes to requests and guarantees that high-priority requests receive GPU slots before lower-priority ones. A production API might classify revenue-generating customer requests as critical, internal batch processing as standard, and free-tier traffic as best-effort. The scheduling policy then ensures that critical requests never wait behind best-effort traffic, even during load spikes.

10.2.4 Deployment topology: Single-GPU to disaggregated

The deployment topology determines how model computation maps to physical hardware, and this mapping is driven by the ratio of model size to single-device memory capacity. Single-GPU deployment is the simplest topology: the entire model fits in one device's memory, and all inference computation occurs locally. For models under 15–20 billion parameters (30–40 GB in FP16), this topology provides the lowest latency because it eliminates all inter-device communication.

Multi-GPU tensor parallelism shards each layer's weight matrices across multiple GPUs connected by high-bandwidth interconnect (NVLink at 900 GB/s on H100). Each GPU computes a partial result for every layer, and an all-reduce operation synchronizes the partial results before the next layer. This topology is necessary when model weights exceed single-device memory and provides latency reduction proportional to the parallelism degree, at the cost of communication overhead per layer. The model sharding section (section 10.5) analyzes the communication overhead quantitatively.

Disaggregated serving separates the prefill phase (processing the input prompt, which is compute-bound) from the decode phase (generating tokens one at a time, which is memory-bandwidth-bound) onto different hardware pools optimized for each workload profile. Prefill nodes use high-FLOP/s accelerators; decode nodes use high-bandwidth-memory configurations. This separation allows each phase to operate at its hardware's optimal operating point rather than compromising between the two regimes on shared hardware. The disaggregated serving section (section 10.4.9) develops this architecture in detail.

10.2.5 Stateful vs. stateless: The scaling divide

A serving system's statefulness determines whether horizontal scaling is trivial or requires careful engineering. Vision models and embedding lookups are typically stateless: any replica can serve any request because no per-session state persists between requests. Horizontal scaling is straightforward – add replicas behind a load balancer – and failure recovery is instant because the load balancer simply routes to a healthy replica.

LLM serving with KV cache is inherently stateful: the cache accumulated during a conversation creates replica-specific state that cannot be reconstructed without re-running the entire conversation history through prefill. This statefulness has cascading implications for system design. Load balancing requires sticky routing to direct subsequent requests in a conversation to the same replica. Autoscaling down requires draining active sessions, which can take minutes for long conversations. Failure recovery is expensive: when a stateful replica crashes, users either experience the latency of regenerating context (seconds) or lose conversation state entirely.

The choice between stateless and stateful serving is *not a framework feature* but *a consequence of the model architecture*. Systems that serve autoregressive models must engineer for statefulness; systems that serve fixed-computation models can treat scaling as a simpler capacity-planning exercise.

10.2.6 Architectural comparison

Table 10.6 summarizes how the batching, memory, scheduling, topology, and state dimensions interact across the major workload types. The table reveals that no single serving architecture is

optimal for all workloads; the constraint profile of each workload type determines the appropriate design point along each dimension.

Table 10.6: Serving Architecture Dimensions by Workload Type: Each workload's constraint profile determines the optimal design point along five architectural dimensions. Selecting a serving system amounts to choosing the combination that matches the workload's constraints.

Dimension	Vision/Embedding	LLM (Autoregressive)	Recommendation
Batching	Static/dynamic (uniform inputs)	Continuous (variable outputs)	Feature-parallel (sharded embeddings)
Memory	Preallocated (predictable)	Paged (variable KV cache)	Distributed (embedding tables)
Scheduling	FCFS (uniform cost)	Preemptive (high variance)	Priority-aware (SLO tiers)
Topology	Single-GPU replicas	Tensor/pipeline parallel	Hybrid CPU-GPU sharding
State	Stateless	Stateful (KV cache)	Stateless (feature store)

Checkpoint 10.3: Serving dimensions

Verify your understanding of how workload constraints drive architectural choices:

- ☐ **LLM vs. vision**: Explain why preemptive scheduling is more critical for LLMs than for vision models.
- ☐ **Stateful workloads**: Identify the workload type for which stateful serving (requiring sticky routing) is a fundamental requirement.
- ☐ **RecSys standard**: Explain why feature-parallel batching is the standard for recommendation systems but not for LLMs.
- ☐ **Memory pressure**: Identify which dimension (batching, memory, scheduling, topology, state) is primarily affected when quantization reduces model or KV-cache size.

The architectural dimensions determine which serving system to select for a given workload. Rather than comparing frameworks feature by feature, an engineer identifies the workload's position along each dimension (batching pattern, memory profile, scheduling requirements, deployment topology, and statefulness) and selects the system that matches. The optimization techniques in this chapter operate within this architectural framework, each addressing a specific dimension.

10.3 Batching Strategies at Scale

A stream of hundreds of disparate chat requests arrives at an inference server every second. Processing them one by one leaves much of the GPU idle, starved for work between small decode steps. Waiting to gather a large static batch forces the first request in line to endure unacceptable latency. Batching strategies at scale demand dynamic algorithms that fuse requests on the fly without violating strict tail-latency deadlines.

Processing multiple requests together amortizes fixed costs (model loading, kernel launch overhead, and memory transfer latency) across more useful work, trading higher per-request latency for dramatically improved throughput. Single-machine serving applies this insight through **dynamic batching**, which collects requests within a time window before processing them together.

At scale, batching becomes more complex because different model architectures have distinct batching requirements. A strategy optimal for vision models may be catastrophic for LLMs, and techniques developed for recommendation systems may not apply to either.

In large consumer platforms, recommendation systems can constitute the majority of AI inference cycles or capacity pressure, with vision, language, ranking, fraud, ads, and other classification workloads sharing the remainder (Gupta et al. 2020; Hazelwood et al. 2018). Despite this distribution, we present batching strategies in order of conceptual complexity: vision (straightforward batching), LLMs (continuous batching with KV cache), and recommendation (feature-parallel batching with distributed embedding). This pedagogical ordering builds understanding progressively, even

though practitioners may frequently encounter recommendation workloads first. The taxonomy that follows matches batching strategies to model characteristics, providing quantitative analysis of when each approach applies and what performance to expect.

10.3.1 Why batching differs across model types

Batching efficiency depends on how computation scales with batch size relative to how memory and communication scale. Different model architectures exhibit different scaling relationships, requiring different batching strategies, as figure 10.4 summarizes.

For **vision models**, including convolutional neural networks (CNNs) and vision transformers (ViTs) processing fixed-size images, computation scales linearly with batch size while memory scales sub-linearly due to weight sharing. Larger batches improve GPU utilization with minimal overhead, making static or dynamic batching with large batch sizes optimal.

For LLMs in the decode phase, computation per token is small relative to memory bandwidth requirements for loading model weights. The bottleneck is *memory bandwidth*, not *compute*. Larger batches amortize weight loading across more tokens, dramatically improving throughput but with diminishing returns as batch size grows.

For recommendation systems, the bottleneck is often embedding lookup rather than dense computation. Batching strategies must optimize for parallel embedding access patterns rather than matrix multiplication throughput.

10.3.2 The physics of batching: The efficiency curve

Batching is not merely a heuristic; it is a trade-off governed by the physics of hardware utilization. We can model the relationship between batch size (B), request latency (T_{lat}), and throughput (X) to identify the optimal operating point for any inference system.

Here T_{lat} follows standard queuing notation for request time in system. Elsewhere in the book, L_{lat} denotes fixed latency overheads or latency components; this section keeps T_{lat} for the end-to-end request-time variable used in Little's Law and batching equations.

The latency equation decomposes per-request latency into fixed overheads (kernel launch, memory loading) and variable costs (compute per sample):

$$T_{\text{lat}}(B) = T_{\text{fixed}} + B \times T_{\text{variable}}$$

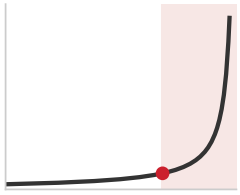
- T_{fixed}: Costs paid once per batch (for example, loading weights from HBM, kernel launch latency).
- T_{variable}: Marginal cost of adding one request (for example, compute time for that sample).

The throughput equation describes the system's capacity:

$$X(B) = \frac{B}{T_{\text{lat}}(B)} = \frac{B}{T_{\text{fixed}} + B \times T_{\text{variable}}}$$

The resulting **Batching Efficiency Curve** shows three distinct regimes. For small batch sizes (B), throughput is dominated by T_{fixed}, making the system latency-bound (or overhead-bound) where increasing B yields super-linear throughput gains. As B becomes large, the T_{fixed} term becomes negligible, and throughput asymptotically approaches the hardware limit $1/T_{\text{variable}}$, leaving the system compute-bound (or bandwidth-bound for LLMs). The optimal batch size sits at the knee of the curve, the point where throughput gains diminish while latency continues to grow linearly.

The engineering goal is to find the maximum B such that $T_{\text{lat}}(B) \leq \text{SLO}$. This formulation explains why vision models (high T_{variable}) saturate at smaller batches than LLMs (high T_{fixed} due to weight loading), requiring different tuning strategies.



Throughput saturates at the batch size where latency hits the SLO.

> Napkin Math 10.2: The batching efficiency curve
>
> **Problem**: An engineer optimizes two services: a Vision model (ResNet) and an LLM (70B). At what batch size does each hit the "Knee" of its efficiency curve?



Figure 10.4: Batching Strategies Compared: Three timelines show how batching policies trade wait time for GPU utilization. (a) No batching: requests processed sequentially, GPU idle between them (around 40 percent utilization). (b) Static batching: requests wait until a full batch is assembled before dispatch, with padding for short requests (around 70 percent utilization). (c) Dynamic batching: assembles a batch within a short time window with adaptive size, reducing wait and lifting utilization (around 85 percent).

Math: The knee occurs where the variable compute cost $(B \times T_{\text{var}})$ starts to exceed the fixed overhead (T_{fixed}).

1. **Vision Model** ($T_{\text{fixed}} = 2$ ms, $T_{\text{var}} = 1$ ms):
   - Knee: $B \approx 2\text{ ms}/1\text{ ms} = 2$.
   - **Result**: This simplified latency model reaches its first overhead-amortization knee at a very small batch. Production CNNs often continue gaining throughput until batches of 32–64+ before hardware saturation.
2. **LLM Decode** ($T_{\text{fixed}} = 40$ ms, $T_{\text{var}} = 0.5$ ms):
   - Knee: $B \approx 40\text{ ms}/0.5\text{ ms} = 80$.
   - **Result**: LLMs require massive batches to amortize the expensive weight-loading overhead.

Systems insight: Because the LLM's "Fixed Overhead" (loading 140 GB of weights from HBM) is so large, the system has not reached the efficiency knee until batch size 80. LLM serving therefore requires continuous batching and paged memory: the system must pack hundreds of concurrent users into a single batch just to overcome the memory bandwidth bottleneck.

Translating these batch-level equations into system-wide capacity planning requires a classical result from queuing theory: **Little's Law**[7], which relates concurrency, throughput, and latency in any stable system. Across model architectures, table 10.7 shows that batching choice follows the binding bottleneck: compute for vision models, memory capacity or bandwidth for LLM prefill and decode, embedding lookup for recommendation systems, and real-time latency for speech.

Table 10.7: Batching Strategy by Model Type: Each model type has characteristic batching behavior determined by its computational bottleneck.

Model Type	Batching Strategy	Typical Batch Size	Key Constraint	Throughput Scaling
Vision (CNN)	Static/Dynamic	32–256	GPU compute	Near-linear to 64+
LLM (prefill)	Dynamic	1–64	Memory capacity	Sub-linear
LLM (decode)	Continuous	100–1000s	Memory bandwidth	Log-linear
RecSys	Feature-parallel	1000–10000s	Embedding lookup	Depends on sharding
Speech	Streaming	1	Real-time	N/A (latency-bound)

7 **Little's Law**: From queuing theory, $Q_{\text{req}} = \lambda_{\text{arr}} T_{\text{lat}}$, so concurrency equals arrival rate times latency. For a serving fleet, this defines the capacity envelope: if a GPU can handle 32 concurrent requests and each takes 100 ms, the maximum arrival rate is 320 req/s. Beyond this, queues grow rapidly as utilization approaches 1, and tail latency (L_{lat}) explodes.

Theorem 10.1: Little's Law for inference

Concept: In any stable queuing system, the average number of requests in the system (Q_{req}) equals the arrival rate (λ_{arr}) multiplied by the average time a request spends in the system (T_{lat}).

$$Q_{\text{req}} = \lambda_{\text{arr}} \cdot T_{\text{lat}}$$

Application: Concurrency Planning

- **Target Throughput** (λ_{arr}): 1,000 requests/sec
- **Latency SLO** (T_{lat}): 100 ms (0.1 s)

Required Concurrency (Q_{req}): $Q_{\text{req}} = 1000 \times 0.1 = 100$ concurrent requests
Capacity Planning: If a single GPU replica handles batch size 8 with 80 ms latency:

1. Replica Throughput $= 8/0.08 = 100$ req/s
2. Throughput-only lower bound $= \lceil 1000/100 \rceil = 10$ replicas
3. SLO-sized replica count $= \lceil 100/8 \rceil = 13$ replicas

Verification: Total system concurrency $= 13$ replicas $\times\, 8$ batch $= 104$, which leaves 4 request slots above the 100 required by Little's Law. The throughput-only lower bound would run at the stability boundary; the SLO-sized fleet runs at about 76.9 percent service utilization, leaving finite headroom for queueing variance and tail latency.

The distinction between throughput capacity and latency capacity is exactly what queuing theory formalizes; section F.1.1 derives the M/D/1 wait-time distribution that determines how much headroom a serving SLO needs.

10.3.3 Static and dynamic batching for vision models

Vision models represent the simplest batching case because inputs have uniform size (after preprocessing) and computation follows a predictable pattern. Single-machine batching principles apply directly, with scale introducing considerations of batch formation across multiple replicas.

Static batching collects exactly B requests before processing. This maximizes GPU utilization when request arrival is predictable but causes unbounded latency during low-traffic periods.

Dynamic batching collects requests for a maximum time window T_{window} or until reaching maximum batch size B_{max}, whichever occurs first. The expected latency under Poisson arrivals with rate λ_{arr} follows equation 10.3:

$$E[T_{\text{total}}] = E[T_{\text{queue}}] + T_{\text{batch}} + T_{\text{inference}}(B) \tag{10.3}$$

where $E[T_{\text{queue}}]$ is the expected queuing delay, T_{batch} is the batch formation delay (up to T_{window}), and $T_{\text{inference}}(B)$ is the inference time for batch size B. The arrival rate enters through the queuing and formation terms: under Poisson arrivals the mean inter-arrival gap is $1/\lambda_{\text{arr}}$, so a higher λ_{arr} fills the batch faster and shrinks both $E[T_{\text{queue}}]$ and T_{batch}, while a lower rate pushes them toward the T_{window} ceiling. The worked example that follows makes these interactions concrete.

Example 10.1: Dynamic batching for ResNet-50

Consider a vision classification service with the following requirements:

- **Arrival rate**: 5,000 QPS
- **Latency SLO**: 50 ms P99
- **Batch service time**: 5 ms at batch=1, 25 ms total for batch=32
- **Number of replicas**: 10 (each handling 500 QPS)

For a single replica with Poisson arrivals at $\lambda_{\text{arr}} = 500$ QPS:

The three choices in table 10.8 differ only in how much latency budget they convert into batch size:

Table 10.8: Dynamic batching policy comparison: Option C achieves about 33 percent higher per-replica capacity, or 25 percent lower utilization, at the cost of higher average latency. Both dynamic batching policies meet the 50 ms P99 SLO.

Policy	Batching window	Expected batch	Per-request compute	Utilization	Latency outcome
No batching	0 ms	1 request	5 ms	$\rho_{\text{serv}} = \lambda_{\text{arr}} \times T_{\text{svc}} = 500 \times 0.005 = 2.5$	Overloaded; cannot meet demand
Dynamic B	10 ms	5 requests	1.6 ms	$\rho_{\text{serv}} = 500 \times 0.0016 = 0.8$	~15 ms mean, ~30 ms P99
Dynamic C	20 ms	10 requests	1.2 ms	$\rho_{\text{serv}} = 500 \times 0.0012 = 0.6$	~22 ms mean, ~42 ms P99

Systems insight: Dynamic batching is useful only when the latency budget has slack. The serving policy converts unused latency headroom into higher throughput, but the same window would violate an already tight SLO.

At scale with multiple replicas, batch formation can occur either at individual replicas or at a centralized batching layer. Replica-local batching has each replica independently form batches from its assigned traffic. This approach is simpler to implement but may result in uneven batch sizes across replicas when load is imbalanced. Centralized batching uses a batching service to collect requests and dispatch formed batches to replicas. This achieves more uniform batch sizes but adds a centralization bottleneck and additional network hop. Production systems typically use replica-local batching with load balancing that ensures roughly equal traffic distribution, achieving the benefits of centralized batching without the complexity.

10.3.4 Continuous batching for LLM inference

The autoregressive bottleneck (principle 14) governs this regime: in generative models, the decode phase is strictly memory-bandwidth bound because the entire model weight set must be loaded for every single token generated. Throughput scales with batch size, sharing weight loads across multiple requests, not compute power.

Definition 10.2: Continuous batching

Continuous Batching is a serving strategy that decouples batch membership from iteration boundaries, allowing new requests to enter and completed ones to exit at every decode step.

1. **Significance**: It maximizes system throughput (X) by eliminating the padding waste and head-of-line blocking inherent in static batching. It ensures the GPU remains saturated even when requests have widely varying sequence lengths.
2. **Distinction**: Unlike static or dynamic batching (which group requests at the request level), continuous batching operates at the Iteration Level, dynamically reshaping the compute tensor at each clock cycle.
3. **Common pitfall**: A frequent misconception is that continuous batching is "purely a scheduler change." In reality, it requires a Dynamic Memory Manager (like PagedAttention) because the KV caches for different requests grow and shrink at different rates, preventing static memory preallocation.

Decoupling batch membership from iteration boundaries is slot reuse: when one request reaches its stop token, the scheduler hands its KV-cache slot to a waiting request on the next decode step instead of leaving capacity idle until the longest sequence finishes.

Autoregressive language models present a unique batching challenge that static and dynamic approaches handle poorly. The key insight comes from the Orca system[8] (Yu et al. 2022): traditional batching forces all sequences in a batch to complete before any new sequences can join, wasting compute when sequences finish at different times.

Consider a batch of 8 sequences. If one sequence completes after 10 tokens while others require 100 tokens, the completed sequence's GPU resources sit idle for 90 iterations. With traditional batching:

$$\text{Wasted compute} = \frac{(100-10)\times 1}{100\times 8} = 11.25\%$$

For realistic output length distributions with high variance, wasted compute can exceed 50 percent. Continuous batching (also called iteration-level batching) decouples batch membership from iteration boundaries. Algorithm 10.1 states the scheduler as a decode loop: completed requests release KV-cache pages, waiting requests enter freed slots, and the next batched kernel runs over the reorganized active set. This technique is central to the throughput-latency trade-off that defines large-scale LLM serving—Archetype A workloads (GPT-4/Llama-3, section 1.6.1) would be economically unviable without it, because their memory-bandwidth-bound decode phase leaves compute cores idle waiting for weights to load, and only by interleaving unrelated requests can the bandwidth be saturated.

8 **Orca (Iteration-Level Scheduling)**: Introduced by Yu et al. (2022), Orca demonstrated that scheduling at iteration granularity rather than request granularity allows sequences to enter and exit batches at each decode step. This eliminated the head-of-line blocking that wasted 50 percent+ of GPU compute in static batching and established the scheduling paradigm adopted by LLM serving systems such as vLLM.

Algorithm 10.1 Continuous batching decode scheduler

Require: waiting queue Q; active set A; KV-cache page allocator; active-token capacity C_{tok}; batched decode kernel
Ensure: generated tokens for completed requests; a continually refreshed active batch

```
while the service is running do
    admit requests from Q while capacity and free KV-cache pages remain; allocate state, append to A
    run one batched decode kernel over all sequences in A
    append each sampled token; extend its KV-cache pages
    remove completed or length-limited sequences from A; return their pages to the allocator
    if HBM pressure exceeds the policy threshold then
        preempt low-priority sequences by paging their KV-cache to CPU DRAM
    end if
    fill freed capacity from Q, or resume paused sequences whose pages now fit
end while
```

Admitting, completing, and refilling slots on every iteration keeps active-token capacity from idling behind the longest sequence in the batch, while the preemption path adds HBM-to-CPU-DRAM transfer latency under memory pressure. The throughput gain therefore scales with output-length variance and must be balanced against KV-cache movement. As figure 10.5 illustrates, static batching leaves the GPU idle while waiting for the longest request, whereas continuous batching keeps it saturated.

10.3.5 Continuous batching throughput analysis

The contrast in figure 10.5 motivates the throughput analysis: static batching leaves GPUs idle for the duration of the longest sequence in each batch, while continuous batching eliminates these idle gaps by inserting new requests at every iteration boundary. Continuous batching's dynamic batch management maintains high GPU utilization regardless of sequence length variance. The throughput improvement depends on sequence length distribution. For a distribution with coefficient of variation $\text{CV} = \sigma/\mu$, the gain is approximately equation 10.4:

$$\text{Throughput gain} \approx 1 + \frac{\text{CV}^2}{2} \tag{10.4}$$

With typical LLM output lengths having $\text{CV} \approx 1.0$, continuous batching achieves approximately $1.5\times$ throughput improvement. For highly variable outputs (conversational vs. code generation),

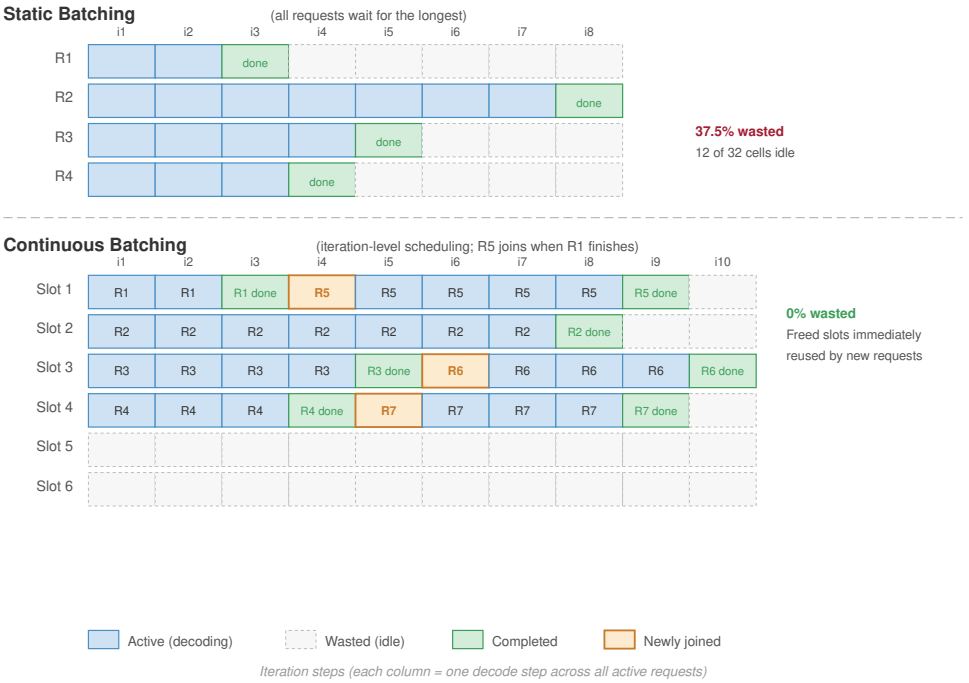


Figure 10.5: Static vs. Continuous Batching: In static batching (A), all requests in a batch must wait for the longest request to complete before the GPU can begin the next batch, leading to significant idle compute time (shaded gray). Continuous batching (B) allows new requests to enter the batch as soon as any request finishes, keeping the GPU saturated and dramatically improving throughput.

gains can reach 2–4×. This $1 + \text{CV}^2/2$ form estimates the average gain across the output-length distribution; section 10.3.6.3 develops a complementary $1 + k \cdot \text{CV}$ upper bound that compares the single longest sequence against the mean. The distributional estimate predicts typical throughput; the max-over-mean bound caps the best case a workload can reach.

The analytical gains translate directly into production systems. The following implementation study examines how vLLM realizes continuous batching through iteration-level scheduling, paged memory management, preemption, throughput, and GPU utilization.

Example 10.2: Continuous batching in vLLM

vLLM implements continuous batching with several key mechanisms. Iteration-level scheduling evaluates at each decode step which sequences have generated end-of-sequence tokens (remove from batch), which waiting sequences can fit in available KV cache slots (add to batch), and which sequences should be preempted if memory pressure exists (swap to CPU). Memory management uses PagedAttention (detailed in section 10.4), which enables dynamic allocation without fragmentation. When a sequence completes, its KV cache pages are immediately available for new sequences. The batched decode kernel processes all active sequences in a single batched operation despite dynamic batch composition. Sequences at different generation lengths are padded to a common shape within the kernel.

Preemption and swapping. A critical challenge in continuous batching is memory contention. As sequences grow during generation, they consume more KV cache pages. If the GPU memory fills up, the system cannot simply crash; it must preempt running requests.

vLLM implements a virtual memory mechanism similar to an operating system's swap. When memory is exhausted, the scheduler identifies low-priority requests (for example, those most recently started) and swaps their KV cache blocks from GPU HBM to CPU DRAM. These

requests are paused until memory becomes available, at which point they are swapped back in and resumed. This mechanism ensures system stability under heavy load at the cost of increased latency for preempted requests.

Typical performance: Table 10.9 reports the throughput and utilization improvement for Llama-2 70B on 8× A100:

Table 10.9: Continuous batching throughput on Llama-2 70B (8× A100): Tokens-per-second and GPU utilization for static, dynamic, and continuous batching of a Llama-2 70B serving workload on an 8× A100 node.

Batching Strategy	Throughput (tokens/s)	GPU Utilization
Static (batch=8)	400	45%
Dynamic (timeout=50 ms)	580	65%
Continuous	1,200	92%

Systems insight: As table 10.9 shows, the 3× throughput improvement from continuous batching comes from eliminating idle GPU cycles during sequence length variation.

The throughput numbers establish that continuous batching wins when output lengths vary, but they do not show where the gain comes from or how large it can grow. Quantifying the waste that traditional batching leaves on the table turns that intuition into a number a scheduler can act on.

10.3.6 Quantitative analysis: Traditional vs. continuous batching

The first baseline is traditional LLM batching, where unequal output lengths turn request heterogeneity into wasted decode cycles. The mathematics of batching waste reveals exactly how much throughput is lost and under what conditions continuous batching delivers the greatest improvement.

10.3.6.1 The waste function for traditional batching

Traditional batching (also called static batching) processes all requests in a batch through all decode iterations until the longest sequence completes. For batch size B with output lengths $\{S_1, S_2, ..., S_B\}$, the total compute performed is:

$$O_{\text{traditional}} = B \times S_{\max} \times c_{\text{decode}}$$

where $S_{\max} = \max_i(S_i)$ and c_{decode} is the compute cost per decode iteration per sequence. However, the useful compute is only:

$$O_{\text{useful}} = \sum_{i=1}^{B} S_i \times c_{\text{decode}}$$

Equation 10.5 defines the **waste ratio** that quantifies the inefficiency:

$$W = 1 - \frac{O_{\text{useful}}}{O_{\text{traditional}}} = 1 - \frac{\sum_{i=1}^{B} S_i}{B \times S_{\max}} = 1 - \frac{\bar{S}}{S_{\max}} \tag{10.5}$$

where $\bar{S}$ is the mean output sequence length. This reveals that waste depends entirely on the ratio of mean to maximum output length within the batch. For uniform output lengths ($\bar{S} = S_{\max}$), waste is zero. For highly variable lengths, waste can exceed 50 percent.

10.3.6.2 Worked example: LLM serving with variable-length outputs

Consider a GPT-class model serving four concurrent requests with the generation lengths shown in table 10.10 (in tokens):

Table 10.10: Concurrent Request Generation Lengths: Four concurrent requests on a GPT-class model with varying prompt and output lengths. The 4× spread in output length (50 to 200 tokens) drives the batching waste analyzed in this section: a static batch of four pads every request to the longest, wasting compute on the three shorter requests until the longest finishes generating.

Request	Prompt Length	Output Length	Total Tokens
R1	100	50	150
R2	80	200	280
R3	120	100	220
R4	90	150	240

Napkin Math 10.3: Continuous batching waste calculation

Problem: Given four LLM requests with unequal output lengths, quantify how much average latency traditional batching wastes and how continuous batching recovers that idle slot time.

Variables:

- Decode time per iteration (batch of 4): 20 ms
- Maximum output length in batch: 200 tokens (R2)
- Mean output length: $(50+200+100+150)/4 = 125$ tokens

Traditional batching: All four requests must wait for R2 to complete its 200 tokens.

- Total decode iterations: 200
- Total batch time: 200×20 ms $= 4{,}000$ ms
- Request completion times:
  - R1 completes useful work at iteration 50, but waits until iteration 200 → latency = 4,000 ms
  - R2 completes at iteration 200 → latency = 4,000 ms
  - R3 completes useful work at iteration 100, but waits until iteration 200 → latency = 4,000 ms
  - R4 completes useful work at iteration 150, but waits until iteration 200 → latency = 4,000 ms

Waste calculation using equation 10.5:
$W = 1 - \frac{125}{200} = 1 - 0.625 = 37.5\%$
The GPU performs $4 \times 200 = 800$ "sequence-iterations" but only 500 are useful.

Continuous batching: Sequences depart the batch upon completion, and new requests can join.

- Iteration 50: R1 completes → slot freed, new request R5 can join
- Iteration 100: R3 completes → slot freed, new request R6 can join
- Iteration 150: R4 completes → slot freed, new request R7 can join
- Iteration 200: R2 completes

Request latencies with continuous batching (assuming no queuing delay):

- R1: 50×20 ms $= 1{,}000$ ms (4× improvement over traditional)
- R3: 100×20 ms $= 2{,}000$ (2× improvement)
- R4: 150×20 ms $= 3{,}000$ (1.33× improvement)
- R2: 200×20 ms $= 4{,}000$ (no improvement for longest request)

Average latency comparison

- Traditional: 4,000 ms (all requests)

- Continuous: (1,000 ms + 2,000 + 3,000 + 4,000) / 4 = 2,500

Result: Continuous batching reduces average latency by 37.5 percent, exactly matching the waste ratio.

Systems insight: Continuous batching does not make the longest request faster; it prevents shorter requests from occupying finished slots while they wait for that longest request. The gain comes from reusing slots at iteration boundaries, which is the slot-reuse pattern shown in figure 10.5.

10.3.6.3 When continuous batching provides maximum benefit

The continuous-batching analysis reveals that benefit scales with output length variance. A useful upper-bound intuition comes from comparing the longest sequence in a traditional batch with the average sequence length. Let $\text{CV} = \sigma/\mu$ be the coefficient of variation of output lengths. If the longest request in a batch is roughly k standard deviations above the mean, then:

$$\text{Improvement} \approx \frac{S_{\max}}{\bar{S}} = \frac{\mu + k\sigma}{\mu} = 1 + k \cdot \text{CV}$$

where k is the number of standard deviations the maximum output exceeds the mean. Real systems realize less than this upper bound because scheduler overhead, KV-cache pressure, and refill gaps consume part of the theoretical gain. The table therefore reports effective waste and speedup, where speedup $\approx 1/(1 - W_{\text{effective}})$.

Table 10.11 quantifies this relationship across different workload types, including Retrieval-Augmented Generation (RAG) (Lewis et al. 2020):

Table 10.11: Continuous Batching Benefit by Workload Type: Higher output length variance (CV) yields greater improvement from continuous batching. Workloads with predictable output lengths (code completion) see modest gains, while highly variable workloads (RAG with documents of varying length) see dramatic improvement.

Workload Type	CV	k	Waste (Trad.)	Speedup (Continuous)
Code completion	0.3	2.5	16.7%	1.2×
Chat (short responses)	0.6	2	33.3%	1.5×
General text generation	1	2.5	50%	2×
Creative writing	1.5	3	64.3%	2.8×
RAG with variable docs	2	2.5	71.4%	3.5×

The systems pattern is straightforward: continuous batching is most valuable when output lengths are unpredictable, mixed workload types share the same cluster, request volume is high enough to refill vacated slots, and latency SLOs make early completion valuable. It provides minimal benefit when output lengths are uniform, as in classification or embedding generation, when batch sizes are too small for slot reuse to matter, or when request volume is too low to refill vacated slots.

10.3.6.4 Implementation complexity trade-offs

The same variability that makes continuous batching valuable also makes it harder to operate. Its performance benefits come with implementation complexity that systems engineers must weigh.

Memory management complexity increases substantially. Traditional batching allocates a fixed KV cache region per sequence at batch formation, deallocating only when the entire batch completes. Continuous batching requires dynamic allocation as sequences grow and immediate deallocation

upon completion, necessitating sophisticated memory management akin to operating system virtual memory.

Scheduler complexity rises correspondingly. Traditional batching uses simple FIFO scheduling: collect requests until the batch is full or the timeout expires, then execute. Continuous batching requires per-iteration decision-making about which sequences to admit, which to preempt if memory pressure exists, and how to handle priority classes. This increases scheduler overhead from $\mathcal{O}(1)$ per batch to $\mathcal{O}(B)$ per iteration.

Kernel design must also adapt. Batched GPU kernels traditionally assume fixed batch composition. Continuous batching requires kernels that handle variable-length sequences efficiently, often through techniques like packing multiple short sequences into shared attention masks or using specialized memory layouts that support dynamic batch membership.

Table 10.12 summarizes these trade-offs:

Table 10.12: Traditional vs. Continuous Batching Trade-Offs: Continuous batching provides significant throughput gains for variable-length workloads at the cost of implementation complexity. For uniform-length workloads, the complexity overhead may not justify adoption.

Dimension	Traditional Batching	Continuous Batching
Implementation effort	Low (standard frameworks)	High (custom scheduler, kernels)
Memory overhead	Fixed allocation	Dynamic + fragmentation mgmt
Scheduler latency	~0.1 ms per batch	~0.5-1 ms per iteration
Debugging complexity	Deterministic behavior	State-dependent, harder to trace
Throughput (variable)	Baseline	1.5–3.5× improvement
Throughput (uniform)	Baseline	~1.0× (no improvement)

For new LLM serving deployments, continuous batching frameworks like vLLM, TensorRT-LLM, or TGI provide mature implementation paths. The decision becomes whether to adopt these frameworks or build custom serving infrastructure. For organizations with existing traditional batching systems, the migration cost must be weighed against the workload's output length variance using table 10.11.

Even with mature serving frameworks, diagnosing tail latency anomalies requires systematic investigation across the system stack. A representative P99 latency regression shows how the fleet stack methodology applies across infrastructure, execution, and serving layers.

Example 10.3: Debugging high P99 latency

Scenario: An LLM serving system exhibits unexpectedly high tail latency. P50 latency is 100 ms and P95 is 180 ms, both within SLO, but P99 spikes to 500 ms against a 200 ms target. GPU utilization appears healthy at 85 percent.

Setup: The system runs on 4 A100-80 GB GPUs connected via PCIe Gen4 (32 GB/s per GPU) rather than NVLink. Memory bandwidth per GPU is 2.04 TB/s (HBM2e), adequate for decode operations, but PCIe limits tensor-parallel communication. For a batch requiring 100 MB activation transfers between GPUs, raw bandwidth math gives approximately 3.1 ms per synchronization point over PCIe versus 0.17 ms over NVLink. The scheduling policy is the real problem: dynamic request-level batching uses max batch size 32 and timeout 50 ms, and the largest 5 percent of batches experience head-of-line blocking because FIFO batching holds short requests behind long-sequence requests.

Systems lesson: The root cause is not the hardware layer but the scheduling policy. Implement priority-aware scheduling with separate batch size limits by request type:

1. **Request classification**: Tag requests by expected output length (short: under 50 tokens, medium: 50 to 200 tokens, long: over 200 tokens) based on prompt patterns or user-provided hints
2. **Differentiated batching**: Limit short-request batches to 8, medium to 16, long to 32

3. **Priority preemption**: Allow short requests to preempt long-running sequences when P99 approaches SLO

After implementing these changes, P99 dropped to 185 ms. High average GPU utilization can mask head-of-line blocking that affects only a small percentage of requests but drives P99; section 10.7 develops the full isolation framework for keeping one workload's tail from degrading another.

The batching strategies examined so far divide along a fundamental constraint boundary. Vision workloads are compute-bound with fixed-shape tensors: every image in a batch undergoes identical arithmetic, so the batch formation problem reduces to packing a GPU's compute pipeline as tightly as possible. LLM workloads are memory-bound with variable-length sequences: the KV cache grows per-request and per-token, so the batch formation problem shifts to memory accounting, eviction policies, and iteration-level scheduling that continuous batching provides. Each regime produced a different dominant strategy because the scarce resource differs: compute throughput for vision, memory capacity for language.

Recommendation systems present a third constraint regime that resembles neither. *Sparse embedding lookups*, not *dense matrix multiplications*, dominate both compute time and memory traffic. A single recommendation request may touch millions of embedding table entries scattered across shards, while the dense ranking head that follows is comparatively small. This access pattern demands a batching strategy organized around feature types and embedding locality rather than around request shape or sequence length.

10.3.7 Feature-parallel batching for recommendation systems

Recommendation batching starts by grouping work around embedding locality rather than request shape. Figure 10.6 shows how the request is split into feature-specific paths before the dense ranking head recombines the retrieved representations.

Recommendation systems expose the same scheduling principle under a different bottleneck. Their computation pattern involves four stages:

1. **Sparse feature lookup**: Retrieve embeddings for user, item, and context features
2. **Dense feature processing**: Transform and normalize dense features
3. **Feature interaction**: Compute interactions between features (often via attention or factorization)
4. **Ranking head**: Produce final scores

The sparse embedding lookup often dominates latency and determines batching strategy. Feature-parallel batching processes different feature types in parallel rather than batching entire requests:

```
Request 1: [user_id_1, item_ids_1, context_1]
Request 2: [user_id_2, item_ids_2, context_2]
Request 3: [user_id_3, item_ids_3, context_3]

Feature-parallel view:
User embeddings:  [lookup(user_1), lookup(user_2), lookup(user_3)]  → parallel
Item embeddings:  [lookup(items_1), lookup(items_2), lookup(items_3)]  → parallel
Context features: [process(ctx_1), process(ctx_2), process(ctx_3)]  → parallel

Then: Combine features per request for ranking
```

Feature-parallel batching is natural when embeddings are sharded across servers: each embedding server handles lookups for its shard across all requests in the batch. At Meta-scale request volumes, this turns feature sharding into a serving strategy rather than a storage detail.

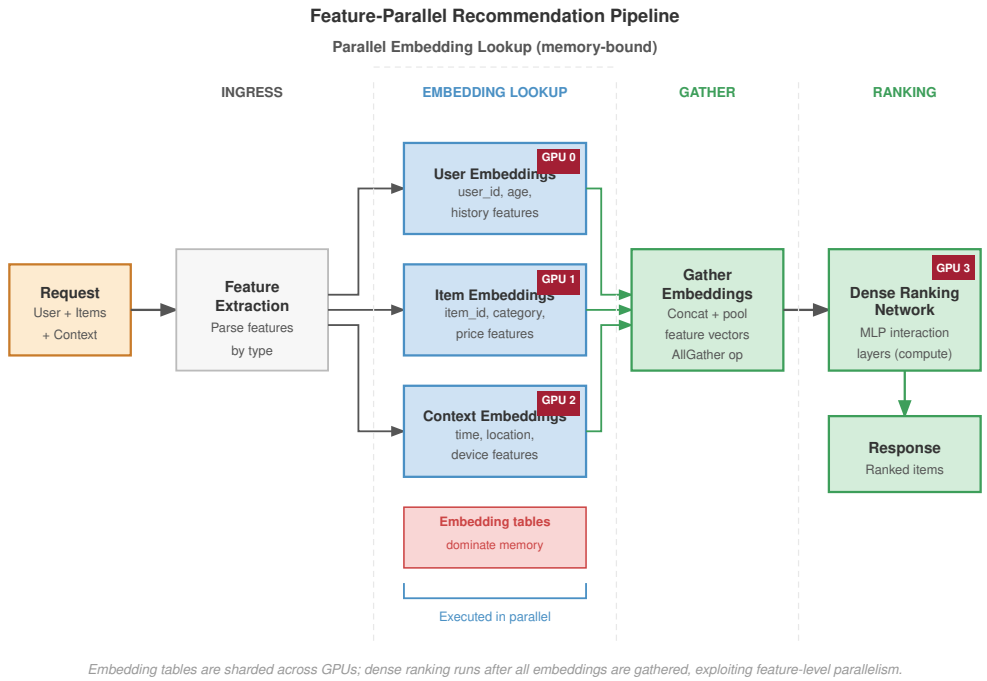


Figure 10.6: Feature-Parallel Batching Pipeline: Requests are batched by feature type (Users, Items, Context) and dispatched to specialized embedding servers in parallel. The retrieved embeddings are then concatenated and processed by the dense ranking head. This architecture enables scaling to trillions of parameters by decoupling embedding storage from dense compute.

Example 10.4: RecSys batching at Meta scale

Consider Meta's recommendation infrastructure serving 10 million QPS across the platform:

Request characteristics:

- Each request queries approximately 100 items (candidate ranking)
- Each item requires 50 embedding lookups (user features, item features, cross features)
- Total per request: 5,000 embedding lookups
- Embedding table size: 100 TB across 1,000 shards

Batching strategy:

With 10 million QPS and 1,000 shards, each shard receives:

$$\text{Lookups per shard} = \frac{\text{QPS} \times \text{lookups/request}}{\text{shards}}$$

In this scenario, the rate is 50 million lookups per second per shard.

Single-threaded processing cannot sustain that rate. Instead, the system uses a batch accumulation window of 1 ms, which yields batches of 10,000 requests at 10 million QPS and about 50K lookups per shard per batch.

Each embedding shard processes 50K lookups in a batched operation, achieving high memory bandwidth utilization through sequential memory access patterns.

As table 10.13 shows, the per-phase latency composition includes request routing, batch accumulation, embedding lookup, feature processing, and ranking:

Table 10.13: Latency breakdown for a recommender request: Per-phase contribution to end-to-end latency for a single recommender request at the request volume and shard count in the batching example.

Phase	Duration	Notes
Request routing	0.2 ms	Consistent hashing to shard
Batch accumulation	0.5 ms (avg)	1 ms window
Embedding lookup	2 ms	Batched, SSD-backed
Feature processing	1 ms	Dense computation
Ranking model	1.5 ms	Final scoring
Total	**5.2 ms**	Within 10 ms SLO

Systems insight: Recommendation batching is organized around sparse feature access, not around identical request tensors. The system wins by batching embedding lookups at the shard where memory bandwidth is consumed.

10.3.8 Streaming inference for real-time applications

Streaming workloads define the boundary where batching itself becomes the wrong response. Real-time speech recognition, video analysis, and robotics require processing inputs as they arrive with minimal latency.

Streaming inference processes inputs incrementally without waiting for batch formation. In speech recognition, it processes audio frames (10–20 ms chunks) as they arrive from the microphone. In video analysis, it processes frames at the capture rate (30–60 FPS) without buffering. In robotics, it processes sensor readings at the control loop frequency (100–1000 Hz).

For streaming applications, the relevant metric is not throughput but time to process each input:

$$T_{\text{streaming}} = T_{\text{capture}} + T_{\text{transfer}} + T_{\text{inference}} + T_{\text{action}}$$

where all components must complete within the inter-frame interval. A streaming speech-to-text pipeline shows how these latency components compose under tight real-time constraints.

Example 10.5: Streaming speech recognition pipeline

Consider a streaming speech-to-text system with 20 ms audio frames:

Latency budget: 100 ms end-to-end (5 frames of delay)

Table 10.14 traces the per-stage latency, including feature extraction, that the pipeline must hit to stay within the 100 ms budget:

Table 10.14: Streaming speech recognition pipeline latency: Per-stage latency for a streaming speech-to-text pipeline operating on 20 ms audio frames against a 100 ms end-to-end budget.

Stage	Duration	Notes
Audio capture	0 ms (continuous)	Microphone buffer
Network to server	20 ms	Including jitter buffer
Feature extraction	5 ms	MFCC computation
Encoder inference	30 ms	Streaming Conformer
Decoder step	15 ms	CTC or transducer decoding
Text formatting	5 ms	Capitalization, punctuation
Network to client	15 ms	Response transmission
Total	**90 ms**	Within 100 ms budget

Here, a Streaming Conformer is the low-latency speech encoder, while CTC and transducer decoding are incremental decoders that emit text from frame-level acoustic states without waiting for the full utterance.

Key constraints:

- No batching: Each frame processes individually
- Stateful model: Encoder maintains context across frames
- Pipeline parallelism: While frame N is in decoder, frame N+1 is in encoder

GPU utilization is typically 30–50 percent for streaming workloads, traded for latency guarantee.

Systems insight: Streaming inference deliberately sacrifices accelerator utilization to preserve end-to-end latency. The binding constraint is the inter-frame deadline, not average throughput.

10.3.9 Adaptive batching strategies

Once batching is workload-specific, fixed parameters become fragile. Production systems adapt batching behavior based on current conditions:

Traffic-adaptive batching adjusts the batch window based on arrival rate:

$$T_{\text{window}} = \min\left(T_{\max}, \frac{B_{\text{target}}}{\lambda_{\text{current}}}\right)$$

When traffic is high, the window shrinks because the target batch size fills quickly. When traffic is low, the window extends but is capped to bound maximum latency.

SLO-adaptive batching takes a complementary approach, monitoring latency percentiles and adjusting parameters aggressively:

```
if P99_latency > 0.9 * SLO:
    reduce B_max by 20%
    reduce T_window by 20%
elif P99_latency < 0.5 * SLO:
    increase B_max by 10%
    increase T_window by 10%
```

The feedback loop maintains latency headroom while maximizing throughput during normal operation. Request-aware batching adds a third dimension by considering request characteristics when forming batches. For LLMs, this means grouping requests by expected output length (inferred from prompt type), grouping them by prompt length to minimize padding, and prioritizing latency-sensitive requests in smaller batches.

Production serving infrastructure embodies these adaptive principles. The NVIDIA Triton Inference Server implements a configurable adaptive batching system that illustrates how SLO-aware and

request-aware strategies operate in practice. Triton exposes three knobs: **max_batch_size** (the upper bound on batch size), **batching_timeout_ms** (the maximum time to wait for batch formation), and **preferred_batch_size** (target batch sizes that align with kernel efficiency). Internally, the scheduler maintains separate queues for each preferred batch size and routes requests to minimize total latency:

$$\text{Queue selection} = \arg\min_q \left(\text{wait}_q + \text{exec}(|q|+1)\right)$$

This optimization weighs both the current queue length and the kernel-efficiency of the resulting batch size. Table 10.15 shows the result for ResNet-50 on V100: the scheduler automatically increases batch size to maintain throughput as traffic grows, with measured throughput tracking offered load until saturation near 2,000 QPS.

Table 10.15: Triton Adaptive Batching: ResNet-50 on V100: Observed batch size, average latency, and sustained throughput for the Triton inference server's adaptive batcher under increasing traffic on ResNet-50 (V100).

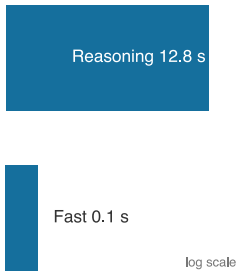
Traffic Level	Avg Batch Size	Avg Latency	Throughput
100 QPS	2.1	8 ms	100 QPS
500 QPS	6.3	12 ms	500 QPS
1000 QPS	12.4	18 ms	1000 QPS
2000 QPS	24.1	28 ms	1980 QPS

Every batching strategy to this point assumes the work per request is fixed once a request is admitted: continuous batching and adaptive windows tune *when* requests join a batch and how long the system waits, but the number of decode steps each request needs is treated as given. Reasoning-heavy inference breaks that assumption. When a model can spend more tokens thinking before it answers, the work per request becomes a variable the scheduler must set, not a property it can only observe.

10.3.10 The logic wall: Test-time compute scaling

Test-time scaling turns reasoning into a serving resource. A request may issue more generated tokens, internal chain-of-thought (CoT) tokens, or search steps before producing the answer. Large-model capability work describes emergent behaviors (Wei et al. 2022), while later analysis cautions that some apparent emergence can be an artifact of metric choice (Schaeffer et al. 2023).

From a serving-systems perspective, the relevant shift is the work issued per request. Extra reasoning consumes latency budget, KV-cache residency, and accelerator time. Models that move from "fast thinking" (instant pattern matching) to "slow thinking" (deliberative reasoning) push pressure from HBM bandwidth toward **Test-Time Compute**: the scheduler must decide how many search steps or CoT tokens a request may consume. The **Logic Wall** is the resulting serving constraint: for complex problems, compute per request grows with task difficulty, and a fleet optimized for tokens per second also needs a policy for allocating thinking time. This is a different control than the batch-window tuning of the previous section: adaptive batching decides when requests run together, while test-time compute decides how much work each request is allowed to do.



Reasoning expands latency about 128 times over fast pattern-matching.

Napkin Math 10.4: Scaling reasoning depth

Problem: Calculate the latency impact of a model that uses 128 "Thinking Tokens" to solve a complex math proof vs. a standard answer.

1. **Standard response**: 1 token answer = 100 ms.
2. **Reasoning response**: 128 of internal search/CoT before the answer.
3. **The latency**: 128 × 100 ms = 12.8 seconds.

Systems insight: Test-time scaling transforms the serving architecture from a throughput factory to a search engine. While standard serving optimizes for tokens per second, reasoning-heavy models are constrained by steps per second. This creates a "reasoning SLO": users may

tolerate 12 seconds for a correct proof, but not for a simple greeting. In the Machine Learning Fleet, this motivates dynamic compute allocation, where the scheduler grants more thinking time to harder prompts.

Variable work per request closes the batching story: continuous and adaptive batching handle variation the system observes, while test-time compute is variation the system chooses. **Dynamic compute allocation** assigns per-request compute budgets based on task difficulty, latency tolerance, and fleet load rather than applying a fixed decode budget to every prompt. With both the batching mechanism and the per-request compute budget now on the table, the remaining task is to decide which combination a given workload should run.

10.3.11 Quantitative summary: Batching strategy selection

The selection problem is to match the batching mechanism to model shape, traffic variance, and latency budget. That match depends on where in the request path the latency budget is actually spent.

Checkpoint 10.4: Batching strategy trade-offs

Verify your understanding of different batching mechanics:

- ☐ **Static vs. continuous**: For a vision service with highly uniform input sizes and predictable traffic, select static batching or continuous batching.
- ☐ **Waste ratio**: Explain why the "Waste Ratio" $(1 - \bar{S}/S_{\max})$ collapses to zero when an LLM serving chat requests moves from static to continuous batching.
- ☐ **Adaptive batching**: Explain why adaptive batching (shrinking the window when traffic is high) reduces P99 latency without sacrificing throughput.
- ☐ **RecSys bottleneck**: In feature-parallel batching for RecSys, identify whether the primary fleet-stack bottleneck is the GPU's Tensor Cores or the distributed Embedding Store's memory bandwidth.
- ☐ **Batching limit**: Explain why free HBM, not Tensor Core compute, often caps the achievable batch size for long-context LLM serving.

Before selecting a batching strategy, it is essential to understand where latency accumulates across the full request lifecycle. Figure 10.7 maps each stage from client to response, revealing the "serving tax" that serialization, routing, and coordination impose outside of GPU compute.

As figure 10.7 shows, noncompute stages (serialization, queuing, routing) can consume a substantial fraction of the latency budget, meaning batching strategy selection must account for the full pipeline, not GPU execution time alone. A decision tree guides strategy selection.

```
Is it autoregressive text generation?
  Yes → Continuous batching with chunked prefill
  No → Is it real-time streaming speech/video/robotics?
          Yes → Streaming pipeline with minimal or no batching
          No → Do embedding lookups dominate latency?
                  Yes → Feature-parallel batching (RecSys)
                  No → Dynamic batching with adaptive parameters
```

Table 10.16 shows that each strategy tunes toward a different objective, so the parameter worth tuning follows from the binding goal: throughput for static, the latency-throughput balance for dynamic, decode-variance for continuous, shard capacity for feature-parallel, and the real-time deadline for streaming.

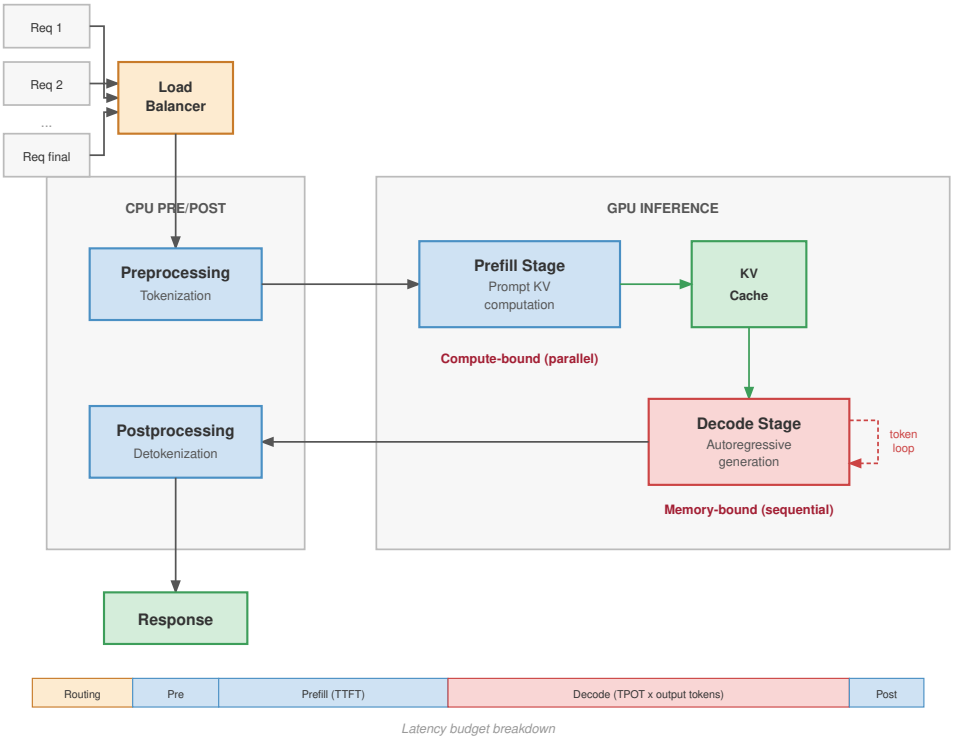


Figure 10.7: End-to-End Inference Pipeline: A detailed view of the request lifecycle: client requests arrive at a load balancer, then undergo preprocessing (tokenization) on the CPU. The data moves to the GPU for the compute-bound prefill stage and the memory-bound decode stage, which uses the KV cache. Finally, the response is postprocessed on the CPU (detokenization) and sent back. This visualization highlights the critical "Serving Tax" components (serialization, routing, coordination) that consume latency budget outside of the actual GPU compute time.

Table 10.16: Batching Strategy Parameters: Each strategy has distinct parameters requiring tuning for the specific deployment.

Strategy	Key Parameters	Tuning Goal
Static	Batch size	Maximize throughput
Dynamic	Window, max batch	Balance latency vs. throughput
Continuous	Chunk size, max batch	Minimize decode latency variance
Feature-parallel	Accumulation window	Match embedding shard capacity
Streaming	Pipeline depth	Meet real-time deadline

Even a well-tuned continuous batching strategy cannot eliminate a deeper, architecture-specific bottleneck for large language models. The context window of every active request must be stored in GPU memory, making KV cache management the next binding constraint on serving throughput.

10.4 Memory and Decode-Time Management

As an LLM generates a 2,000-word essay, it must constantly recall every word it has previously written. It does this by storing intermediate attention states in a rapidly expanding memory buffer known as the KV Cache. Unmanaged, this cache will aggressively fragment GPU memory, causing out-of-memory crashes even when 40 percent of the VRAM is technically free.

Two classes of techniques meet at this boundary. KV-state capacity techniques, such as PagedAttention and prefix caching, decide where attention state lives and how much fragmentation the

memory manager tolerates. Decode-time latency techniques, such as speculative decoding, change how much target-model work is needed per emitted token. Speculative decoding is not KV-cache compression; it trades extra draft-model state, target-model verification, and scheduler complexity for lower time per output token under the same memory budget.

10.4.1 The KV cache wall: Memory-bound capacity

Increasing the batch size to maximize throughput in LLM serving is bounded by the KV cache wall. As figure 10.8 shows, model weights represent a fixed "static tax" on GPU memory, while the KV cache grows linearly with both batch size and sequence length.

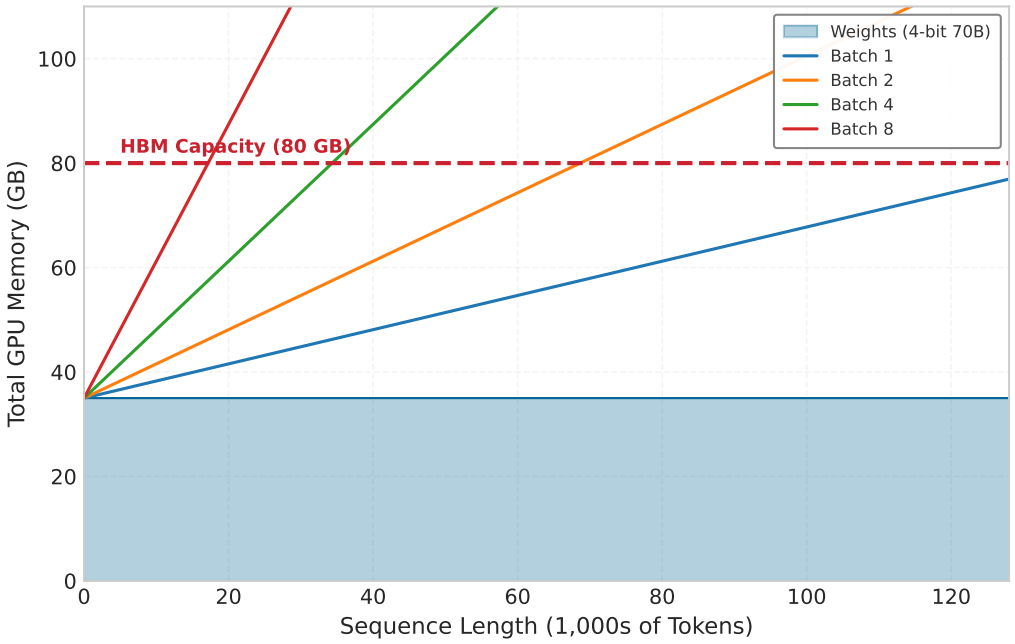


Figure 10.8: The KV Cache Wall: Total GPU memory usage for a 4-bit quantized 70B model (35 GB weights) on an 80 GB H100. While the model fits easily at small context lengths, the KV cache (sloped lines) eventually consumes all remaining HBM. At 128K context, a batch size of 2 is physically impossible on a single GPU (35 GB + 84 GB > 80 GB), forcing the system to either reduce batch size (killing throughput) or shard the model.

The visualization reveals why the replica level must sometimes shard models that would otherwise fit on a single GPU. Sharding provides the memory headroom needed to maintain high batch sizes for long-context requests. Without sharding, a 128K context request effectively "evicts" all other users from the GPU.

The same formula can be turned into an explicit batch-size limit for production hardware.

Example 10.6: KV-cache capacity estimator

Problem: A deployment serves Llama-3-70B (FP16 weights $\approx$ 141.2 GB) on an 8$\times$ H100 node (640 GB total HBM). The goal is to determine the maximum batch size for a context length of 128K tokens.

Formula:

$$M_{\text{KV}} = 2 \times N_L \times H_{\text{KV}} \times d_{\text{head}} \times s_{\text{elem}}$$

$$\text{Total Memory} = M_{\text{weights}} + (\text{Batch} \times \text{Context} \times M_{\text{KV}})$$

Parameters:

- $N_L = 80$, $H_{\text{KV}} = 8$ for Llama-3-70B GQA, $d_{\text{head}} = 128$.

- $s_{\text{elem}} = 2$ bytes (FP16).
- Context $= 131,072$ tokens.

Step 1: Calculate memory per token.

$$M_{\text{KV}} = 2 \times 80 \times 8 \times 128 \times 2 = 327,680 \text{ bytes} \approx 0.33 \text{ MB/token}$$

Step 2: Calculate cache per request.

$$131,072 \text{ tokens} \times 0.33 \text{ MB/token} \approx 42.9 \text{ GB/request}$$

Step 3: Determine max batch size. Available Memory for KV = 640 GB (Total) - 141.2 GB (Weights) - 20 GB (System) = 478.8 GB.

$$\text{Max Batch} = \left\lfloor \frac{478.8}{42.9} \right\rfloor = 11$$

Systems insight: Even with GQA, the shared-KV-head attention variant from section 9.4.3 that the 8-head count above already reflects, 128K-context requests consume tens of gigabytes of KV cache, so the system serves only a small number of concurrent long-context requests before memory becomes the binding constraint. Addressing this requires PagedAttention (to reduce fragmentation) and KV-cache quantization or sharding.

10.4.2 The fragmentation problem

Traditional memory allocation for KV cache preallocates contiguous memory for each sequence based on maximum expected length. This creates two forms of waste.

Internal fragmentation wastes memory within each allocation. Sequences shorter than the maximum allocation leave the unused portion idle. If maximum length is 4,096 but average output is 100 tokens, 97.5 percent of allocated memory is wasted.

External fragmentation compounds this problem across allocations. As sequences complete and new ones start, memory becomes fragmented into noncontiguous free blocks. Even with sufficient total free memory, no single block may be large enough for a new maximum-length allocation.

Consider a simplified example with 8 memory slots and maximum sequence length of 4. Fixed-size reservations first create internal fragmentation:

```
Time 0: Allocate Seq A (slots 0-3), Seq B (slots 4-7)
        [A][A][a][a][B][B][B][b]
        lowercase = reserved but unused
```

External fragmentation appears after sequences complete and new sequences start:

```
Time 1: Active Seq C and Seq D leave two 2-slot gaps
        [ ][ ][C][C][ ][ ][D][D]

Time 2: Try to allocate Seq E (needs 4 contiguous slots)
        [ ][ ][C][C][ ][ ][D][D]  <- Total free slots = 4, largest block = 2

Result: enough total free capacity exists, but no contiguous block is large enough.
```

Production systems report 60–80 percent memory waste from fragmentation under realistic workloads, severely limiting batch sizes and throughput.

10.4.3 PagedAttention

The fragmentation example exposes the allocator failure: enough KV-cache capacity can exist in aggregate while no contiguous block is available for the next sequence. PagedAttention fixes that serving problem by treating KV-cache storage like virtual memory rather than a single preallocated slab.

Definition 10.3: PagedAttention

PagedAttention is an LLM serving memory management technique that applies virtual memory principles to KV Cache allocation, storing attention states in noncontiguous, fixed-size physical blocks.

1. **Significance**: It eliminates Internal and External Fragmentation, which can waste 60–80 percent of the memory allocated to the KV cache under contiguous preallocation. By allowing sequences to grow dynamically, it enables 2–4× higher Concurrent Throughput (X) on the same hardware.
2. **Distinction**: Unlike Contiguous Allocation (which requires prereserving the maximum context length), PagedAttention uses a Block Table to map logical sequence indices to physical memory pages, allocating only what is currently used.
3. **Common pitfall**: A frequent misconception is that PagedAttention speeds up *individual token math*. In reality, it is a Capacity Optimization: it improves *system-level efficiency* by allowing larger batch sizes, though it adds a small Indirection Overhead (L_{lat}) for the pointer lookups.

The virtual-memory shift changes the allocation problem: capacity no longer has to be reserved as one contiguous block before the request starts.

Systems Perspective 10.1: Analogy: The inefficient hotel

Imagine a hotel where every guest might stay anywhere from 1 to 10 days, but they do not know in advance.

Under contiguous allocation, the hotel manager blocks out a 10-day suite for every guest just in case. A 100-room hotel becomes "fully booked" with only 10 guests, wasting 90 percent of its capacity (Internal Fragmentation).

Under PagedAttention (Virtual Memory), the manager assigns a guest just 1 room. If the guest stays another day, they are given whatever room is available next, even if it is on a different floor. The front desk maintains a "Block Table" to keep track of which rooms belong to which guest. The hotel can now accommodate 100 guests simultaneously, recovering the 90 percent wasted capacity.

PagedAttention (Kwon et al. 2023), introduced in vLLM, applies virtual memory concepts to KV cache management. Instead of contiguous allocation, the KV cache is divided into fixed-size pages, using 16-token blocks in the default vLLM configuration, and sequences are allocated pages on demand. Figure 10.9 illustrates the key concepts including page tables that map logical sequence positions to physical memory pages, block size that defines the number of tokens per page (typically 16 tokens), and physical blocks that provide fixed-size memory allocations assignable to any sequence.

PagedAttention provides four capacity benefits:

- **Internal fragmentation**: It allocates only the pages needed for actual tokens.
- **External fragmentation**: Any free page can be used by any sequence.
- **Dynamic growth**: Sequences can grow without preallocation.
- **Prefix sharing**: Common prefixes can share physical pages.

Together, these properties turn KV cache capacity into a schedulable resource rather than a fixed per-request reservation.

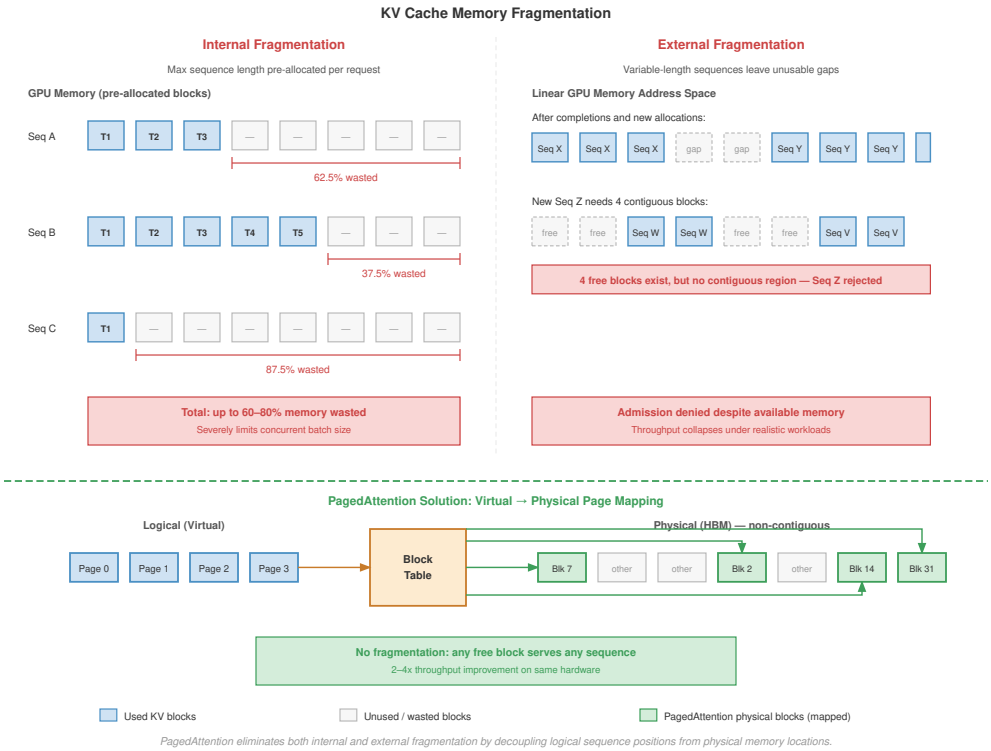


Figure 10.9: KV Cache Fragmentation and the PagedAttention Solution: Top left: internal fragmentation wastes memory by preallocating for the maximum sequence length. Top right: external fragmentation leaves unusable memory gaps. Bottom: PagedAttention solves this by mapping contiguous logical pages of a sequence to noncontiguous physical memory blocks via a block table, allowing the system to fill fragmentation gaps with small blocks from any sequence.

Example 10.7: PagedAttention implementation details

Memory layout:

```
Physical blocks (16 tokens by hidden_dim by 2 by precision):
Block 0:  [K ...K , V ...V ]
Block 1:  [K ...K , V ...V ]
...
Block N:  [K ...K , V ...V ]
```

Page table per sequence:

```python
class PageTable:
    def __init__(self, max_blocks):
        self.block_map = {}  # logical_block -> physical_block

    def allocate_block(self, logical_idx, physical_block):
        self.block_map[logical_idx] = physical_block

    def get_physical(self, logical_idx):
        return self.block_map[logical_idx]
```

Attention kernel modification:

Standard attention: `output = softmax(Q @ K.T/sqrt(d)) @ V`

PagedAttention:

```
def paged_attention(Q, page_table, physical_blocks, block_size):
    # Gather K, V from noncontiguous physical blocks
    for logical_idx in range(num_logical_blocks):
        physical_idx = page_table[logical_idx]
        K_block = physical_blocks[physical_idx].K
        V_block = physical_blocks[physical_idx].V
        # Compute attention for this block
        attention_scores = Q @ K_block.T / sqrt(d)
        output += softmax(attention_scores) @ V_block
    return output
```

Systems insight: The throughput impact of gather operations is small compared to the memory savings; table 10.17 shows the 2.5–4× throughput improvement coming from fitting more concurrent sequences in the same memory.

Table 10.17: PagedAttention vs. Contiguous KV Cache: KV cache pool utilization and relative serving throughput for contiguous allocation against PagedAttention.

Approach	KV Cache Pool Utilization	Throughput (relative)
Contiguous	30-40%	1.0× (baseline)
PagedAttention	95%+	2.5–4×

10.4.4 Prefix caching

Many LLM workloads share common prefixes across requests. System prompts like "You are a helpful assistant…" are prepended to every request. Few-shot examples use the same examples for many queries. Document context involves multiple questions about the same document. Recomputing these shared prefixes wastes both compute (prefill) and memory (duplicate KV cache entries).

Prefix caching shares KV cache entries across requests with common prefixes. Figure 10.10 demonstrates how shared system prompts avoid redundant computation.

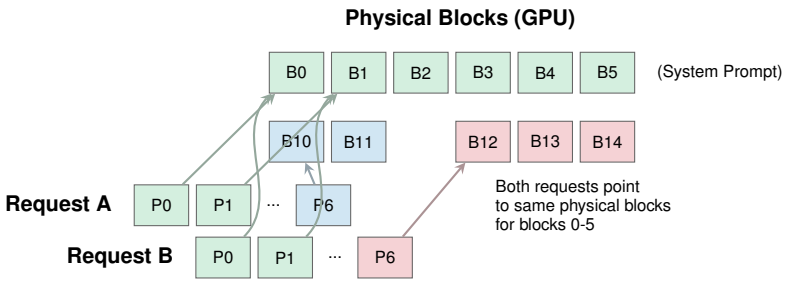


Figure 10.10: Prefix Caching via Block Sharing: PagedAttention enables efficient prefix caching by allowing multiple sequences' block tables to point to the same physical blocks for shared content. In this example, the System Prompt is stored in blocks 0-5. Request A and Request B maps their first 6 logical pages to these same physical blocks, storing only their unique suffixes in new blocks. This dramatically reduces memory usage and prefill computation for workloads with shared context.

With PagedAttention, prefix caching integrates naturally through copy-on-write semantics:

```
System prompt → Physical blocks [0, 1, 2, 3, 4, 5]

Request A page table: [0, 1, 2, 3, 4, 5, 10, 11]  <- shares prefix blocks
Request B page table: [0, 1, 2, 3, 4, 5, 12, 13, 14]  <- shares prefix blocks
Request C page table: [0, 1, 2, 3, 4, 5, 15]  <- shares prefix blocks
```

All three requests reference the same physical blocks for the system prompt. Only when generating unique tokens do they allocate new blocks. The savings can be substantial when many concurrent requests share the same system prompt; the following example quantifies them for a typical chatbot deployment.

Example 10.8: Prefix caching at scale

Scenario: A chatbot service uses a 2,000-token system prompt and serves 1,000 concurrent users.

Without prefix caching:

- KV cache per user: $2000 + 500$ (avg response) $=$ 2,500 tokens
- Total KV cache: 2,500 tokens $\times 1000 \times 2 \times 80 \times 8192 \times 2 =$ 6.6 TB

With prefix caching:

- Shared prefix: 2,000 tokens (once)
- Unique per user: 500 tokens
- Total: $(2000 \times 1) + (500 \times 1000) =$ 502,000
- Memory: 502,000 $\times 2 \times 80 \times 8192 \times 2 =$ 1.3 TB

Result: Prefix caching yields a 79.9 percent memory reduction in KV cache, enabling 5× more concurrent users.

Systems insight: Prefix caching pays off when the prefix hit rate is high because many requests share a long prefix. Table 10.18 contrasts hit rates and memory savings across workloads, showing where the technique changes serving capacity and where it does not.

Table 10.18: Prefix Caching Effectiveness by Workload: Typical prefix hit rates and resulting KV-cache memory savings for chatbot, document-QA, and general API workloads, showing where prefix caching pays off and where it does not.

Workload	Prefix Hit Rate	Memory Savings
Chatbot (same system prompt)	95%+	70-80%
Document QA (same doc)	80-90%	50-70%
General API (diverse)	20–40%	10–30%

10.4.5 KV cache compression and architectural optimization

The capacity techniques above tolerate the cache; compression shrinks it. The calculation here uses a 64-KV-head multi-head attention baseline so the grouped query attention subsection can isolate the head-count reduction as a separate architectural lever. For a 70B parameter model with 80 layers, 64 heads, head dimension 128, and sequence length 4096 in FP16, the KV cache requires:

$$\text{KV cache} = 2 \times 80 \times 64 \times 128 \times 4096 \times 2 \approx 10.7 \text{ GB (FP16)}$$

A single request's KV cache alone consumes a substantial fraction of the H100's 80 GB of HBM. For a batch of 8 concurrent requests, the KV cache would require 85.9 GB, exceeding single-GPU capacity. Reducing this footprint requires two complementary strategies: quantization and architectural optimization.

10.4.5.1 KV cache quantization

Reducing the size of cached values through lower precision provides direct memory savings. Weight-only quantization (GPTQ, AWQ) reduces weight precision while keeping activations in high precision. While effective for storage, this does not reduce the KV cache. **KV cache quantization** explicitly targets the activations, storing cached keys and values in INT8, FP8, or even INT4.

Compressing the KV cache to INT4 reduces it to approximately 2.7 GB per request, a 4× reduction. This freed memory directly translates into larger batch sizes and higher serving throughput. KV cache values exhibit different distributions than model weights: KIVI observes channel-wise outliers in keys and uses per-channel key quantization, while values lack the same channel-wise pattern and are quantized per-token.

10.4.5.2 Grouped query attention (GQA)

Grouped query attention (Ainslie et al. 2023), the shared-KV-head architecture established in section 9.4.3, is the second lever on cache size, complementary to quantization: where quantization shrinks the bytes per cached element, GQA shrinks how many key-value heads are cached per layer. The serving consequence is the head-count arithmetic. For our earlier 70B model, switching from the multi-head attention (MHA) baseline of 64 KV heads to GQA with 8 KV heads shrinks the KV cache by 8×:

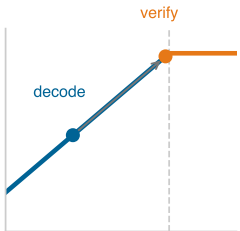
$$\text{KV cache (GQA)} = 2 \times 80 \times 8 \times 128 \times 4096 \times 2 \approx 1.3 \text{ GB (FP16)}$$

At 1.3 GB, the GQA cache is dramatically more manageable than the 10.7 GB required by MHA. Combining GQA with INT8 KV cache quantization yields sub-gigabyte per-request cache sizes, often enabling substantially larger batches on a single GPU. GQA has become a common architectural choice for inference-optimized LLMs because it reduces KV-cache bandwidth and capacity pressure with comparatively small quality impact in many deployments.

> **Napkin Math 10.5: The precision dividend**
>
> **Problem**: A team serves a 70B model on 4× H100 GPUs. Weights in FP16 consume 140 GB (35 GB/GPU). KV cache at FP16 consumes 10.7 GB per request. How does quantizing weights to INT4 and KV cache to INT8 change the maximum batch size?
>
> **Before optimization** (all FP16):
>
> - Weights: 35 GB/GPU
> - Available for KV cache: 80 GB − 35 GB = 45 GB/GPU
> - KV cache per request: 10.7 GB ÷ 4 ≈ 2.7 GB/GPU
> - Maximum batch size: ⌊ 45 GB divided by 2.7 GB ⌋ ≈ 16 requests
>
> **After optimization** (INT4 weights, INT8 KV cache):
>
> - Weights: 35 GB $\times(4/16) = 8.75$ GB/GPU (INT4)
> - Available for KV cache: 80 GB − 8.75 GB = 71.25 GB/GPU
> - KV cache per request (INT8): 5.4 GB ÷ 4 ≈ 1.3 GB/GPU
> - Maximum batch size: ⌊ 71.25 GB divided by 1.3 GB ⌋ ≈ 53 requests
>
> **Systems insight**: Precision engineering fundamentally changes serving economics by enabling larger batch sizes. Larger batches amortize the fixed cost of weight loading, shifting operations from memory-bound toward compute-bound.

10.4.6 Speculative decoding as a serving policy

Section 9.6.1 established speculative decoding as an algorithmic optimization: a small draft model proposes K tokens, the target model verifies them in one parallel forward pass, and rejection sampling guarantees the emitted tokens follow the target model's distribution exactly, so the technique is lossless and bounded by the draft acceptance rate α_{acc}. The serving question is different. Speculation becomes a resource-admission policy under SLA pressure: the scheduler must decide when to enable it, how its memory cost interacts with KV-cache admission, and what it does to the SLA the endpoint is bound by.



Decode is memory-bound; parallel verification moves work toward the ridge.

The latency win is real but conditional, and the condition is the acceptance rate. The expected number of tokens emitted per round, including the correction token, is $\frac{1-\alpha_{\text{acc}}^{K+1}}{1-\alpha_{\text{acc}}}$, which collapses

toward 1 as acceptance falls. For $K = 5$ draft tokens and a draft model 20× faster than the target, that sensitivity is steep:

- **High acceptance** ($\alpha_{\text{acc}} = 0.9$): Expected 4.69 tokens per round, speedup of 3.7×.
- **Medium acceptance** ($\alpha_{\text{acc}} = 0.7$): Expected 2.94 tokens per round, speedup of 2.4×.
- **Low acceptance** ($\alpha_{\text{acc}} = 0.5$): Expected 1.97 tokens per round, speedup of 1.6×.

Predictable text achieves $\alpha_{\text{acc}} > 0.9$, while creative reasoning or code generation may fall below 0.5; well-aligned model pairs that share training data and architecture commonly land at 0.6–0.8. Because acceptance is a property of the workload, not of the hardware, a single global setting is wrong for a mixed fleet: the same speculation that halves time per token on a chat endpoint can slow a code-generation endpoint that sits below the break-even acceptance rate. The serving system must therefore decide per endpoint, and three serving concerns govern that decision.

10.4.6.1 The speculation tax and KV-cache admission

The first concern is memory. The draft model is not free state: it requires its own GPU allocation for weights, and each in-flight request that uses speculation reserves a small additional KV cache for the draft model alongside the target's. Pairing a 7B draft model with a 70B target adds roughly 14 GB of weights per replica plus per-request draft cache. This *speculation tax* competes directly with the KV-cache admission budget developed in section 10.4.1: every gigabyte the draft model holds is a gigabyte the admission controller cannot allocate to a concurrent request. Speculation therefore shrinks the maximum batch size the node can admit, trading aggregate throughput for the latency of individual requests. The admission controller and the speculation policy cannot be tuned independently; enabling speculation lowers the concurrency ceiling that the KV-cache wall already imposes.

10.4.6.2 The latency-throughput tension under an SLA

The second concern is which service-level objective the endpoint is bound by. Speculation improves time per output token but, by lowering the concurrency ceiling, reduces the requests per second a replica can sustain. An endpoint with a tight time-to-first-token or per-token latency SLA, such as interactive chat, benefits: the speculation tax buys headroom against the latency target that matters. An endpoint bound by a throughput SLA, such as bulk document processing or batch summarization, is harmed: speculation spends batch capacity to improve a latency metric no one is measuring, and the lost concurrency directly threatens the throughput target. The decision is therefore an SLA question. The scheduler enables speculation on latency-bound endpoints and disables it on throughput-bound ones, and on a shared replica serving both it must account for the draft model's resident cost against the stricter of the two budgets.

10.4.6.3 Per-endpoint scheduling

The third concern is that these decisions are dynamic, not static configuration. Acceptance rate drifts with the traffic mix, and the value of speculation collapses at high load: when a replica is already near its concurrency ceiling, the decode phase is closer to compute-bound and the marginal latency benefit of speculation shrinks even as its memory tax stays fixed. Production schedulers therefore treat speculation as a runtime knob gated on observed load and acceptance. They enable it for latency-sensitive endpoints during normal operation, disable it under traffic spikes to reclaim KV-cache pages for admission, and may switch draft strategies per endpoint. **Self-speculative decoding** uses early exit from the target model itself as the draft, eliminating the separate-model memory tax at the cost of lower acceptance; **Medusa** (Cai et al. 2024) adds lightweight prediction heads to the target backbone; **Lookahead decoding** uses Jacobi iteration to avoid a draft model entirely. Each variant trades a different slice of the speculation tax against acceptance, and the scheduler chooses among them per endpoint based on which SLA binds and how much KV-cache budget remains.

10.4.7 KV cache in distributed settings

When a conversation is moved, rebalanced, or split across devices, its KV state must either move with it or be rebuilt token by token. Section 10.5 develops the tensor and pipeline parallelism strategies

that distribute a model across devices; each strategy carries a different consequence for KV-cache management:

Under tensor parallelism, the KV cache is sharded across devices along with attention heads. Each device stores cache for its subset of heads.

```
8-way tensor parallelism:
Device 0: KV cache for heads 0-7
Device 1: KV cache for heads 8-15
...
Device 7: KV cache for heads 56-63
```

Cross-device sharing adds a constraint: prefix caching across tensor-parallel devices requires cache to be sharded identically on all devices. This is automatic when prefixes are processed with the same tensor-parallel configuration.

KV cache migration presents a further challenge. When the router moves a conversation to a different replica because of failure or rebalancing (routing keyed on a hash of the session ID, the consistent-hashing scheme developed in section 10.6.6), the KV cache must be migrated:

```
Migration options:
1. Rebuild: Re-run prefill on new replica (500 ms+ for long context)
2. Transfer: Send KV cache over network (100 MB at 100Gbps = 8 ms)
3. Hybrid: Transfer if small, rebuild if large

Decision threshold:
if cache_size_bytes/network_bandwidth < prefill_time:
    transfer()
else:
    rebuild()
```

For Llama-70B GQA with 4K context, KV cache is about 1.3 GB total per sequence, or about 168 MB per device under 8-way tensor parallelism. At 100 Gbps (12.5 GB/s), transferring one shard takes roughly 13 ms, while transferring the full cache would take about 100 ms. Both are often preferable to a ~500 ms prefill rebuild, so transfer remains the better choice when bandwidth is available.

10.4.8 Memory management best practices

Effective KV-cache management begins with capacity accounting rather than an eviction policy. The serving process first reserves memory for weights, activations, runtime overhead, and safety headroom, then treats the remaining HBM as a managed pool for active sequences:

```
Available GPU memory = Total - Weights - Activations - Overhead
KV pool size = 0.9 * Available  # Leave 10% headroom

Max concurrent sequences = KV pool size / (avg_seq_length * per_token_cache)
```

That budget determines how many sequences can coexist before the scheduler must choose between rejection, preemption, or paging. When the cache is full, LRU eviction removes the sequence with the oldest access, size-based eviction frees the longest sequence first, and priority-based eviction protects paid-tier or latency-critical requests. Continuous batching then turns eviction into a scheduling decision: a high-priority request that cannot fit triggers victim selection, swaps the victim's KV cache to CPU memory, allocates GPU memory to the new request, and restores the victim later if its service-level objective still permits the delay.

Example 10.9: KV cache memory hierarchy

Production systems manage KV-cache memory pressure across the three-tier hierarchy in table 10.19, paging from GPU HBM down to CPU DRAM and NVMe SSD as the active working set exceeds each tier's capacity:

Table 10.19: KV cache memory hierarchy: Capacity, swap-in latency, and intended use case for the three tiers production inference servers use to manage KV-cache memory pressure.

Tier	Capacity	Latency	Use Case
GPU HBM	80 GB	0 ms	Active sequences
CPU DRAM	1 TB	1–5 ms	Swapped sequences
NVMe SSD	10 TB	10–50 ms	Long-term cache

Swap implementation:

```
async def swap_to_cpu(sequence_id):
    kv_cache = gpu_cache[sequence_id]
    cpu_cache[sequence_id] = kv_cache.cpu()  # Async transfer
    gpu_cache.free(sequence_id)


async def swap_to_gpu(sequence_id):
    cpu_kv = cpu_cache[sequence_id]
    gpu_cache[sequence_id] = cpu_kv.cuda()  # Async transfer
    cpu_cache.free(sequence_id)
```

Observed performance:

- GPU-only (no swapping): 50 concurrent sequences
- GPU+CPU swapping: 500 concurrent sequences (10×)
- Average swap latency: 3 ms (acceptable for nonurgent requests)

Systems insight: Treating CPU DRAM and NVMe as overflow tiers behind HBM raises concurrency by an order of magnitude at the cost of millisecond swap latency, so paging is a throughput lever for non-urgent traffic but not a substitute for HBM capacity on the latency-critical path.

The mismatch swapping papers over remains structural: a single GPU must handle both the compute-heavy prompt processing and the bandwidth-heavy token generation. Separating those phases onto different hardware pools eliminates this mismatch entirely.

10.4.9 Disaggregated serving: Splitting the workload

LLM inference consists of two distinct phases with different computational characteristics, requiring different batching strategies within the same request. Figure 10.11 illustrates how this architectural split separates the two phases onto dedicated hardware pools, and table 10.20 contrasts their resource profiles.

Definition 10.4: Prefill and decode phases

Prefill and Decode Phases are the two distinct computational regimes of transformer-based LLM inference.

1. **Significance**: The Prefill Phase (processing the prompt) is compute bound (R_{peak}) with high arithmetic intensity, while the Decode Phase (generating tokens) is bandwidth

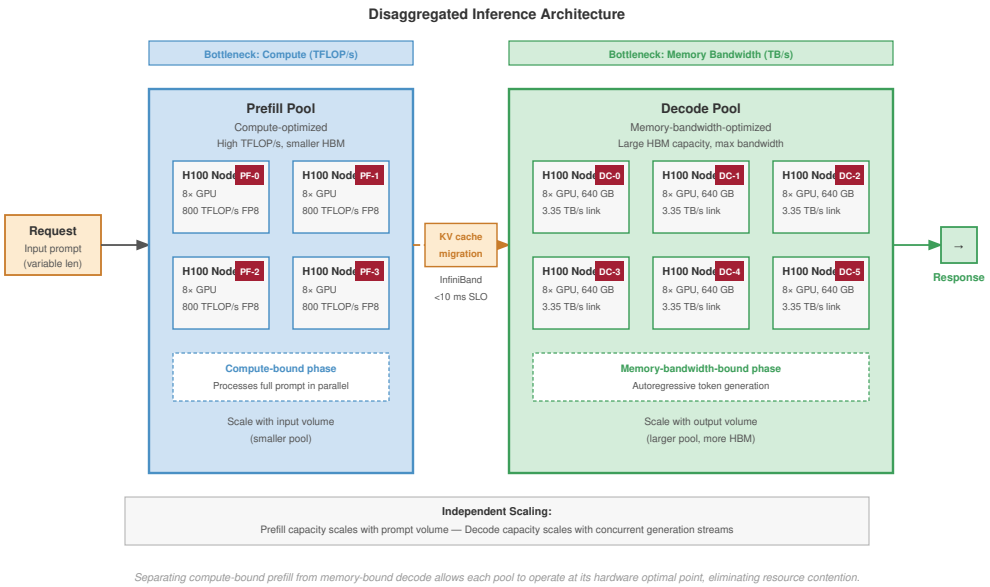


Figure 10.11: Disaggregated Serving Architecture: Separating the compute-bound prefill phase from the bandwidth-bound decode phase. Requests enter the Prefill Pool for prompt processing, and the resulting KV cache is migrated to the Decode Pool for token generation. This allows each phase to run on hardware optimized for its specific bottleneck, improving overall fleet efficiency.

bound (BW) with extremely low arithmetic intensity. This mismatch means a single request's efficiency (η_{hw}) varies wildly during its lifecycle.

2. **Distinction**: Unlike Single-Pass Inference (for example, ImageNet), where the resource bottleneck is constant, LLM inference switches between these regimes at every request, requiring Iteration-Level Scheduling to maintain utilization.
3. **Common pitfall**: A frequent misconception is that both phases should be batched together naively. In reality, because they have opposing hardware requirements, mixing them in the same batch without Chunked Prefill or similar techniques can lead to significant queuing delays (L_{lat}) for decoding tokens.

The prefill phase processes the entire input prompt in parallel. Computation scales with prompt length, and the memory access pattern is compute-bound with high arithmetic intensity[9].

The decode phase, by contrast, generates output tokens one at a time. Each token requires loading the entire model weights, making the memory access pattern bandwidth-bound with low arithmetic intensity. Section C.3.1 places these two phases on the roofline: decode sits far below the ridge point of equation C.4, where throughput is set by HBM bandwidth rather than peak FLOP/s, while prefill's high arithmetic intensity (equation C.2) pushes it past the ridge into the compute-bound regime.

Table 10.20: Prefill vs. Decode Characteristics: The two phases have opposite optimization requirements.

Phase	Computation	Memory Access	Bottleneck	Optimal Batch
Prefill	$\mathcal{O}(\text{prompt length}^2)$	Activation and attention tiles	Compute	Moderate request batches or large chunks

9 **Arithmetic Intensity (Prefill vs. Decode)**: Measured in FLOP/byte of memory accessed, this ratio determines whether a workload is compute-bound or bandwidth-bound. Prefill achieves 100+ FLOP/byte (compute-bound); decode achieves 1–10 FLOP/byte (bandwidth-bound). This 10–100× gap is why mixing the two phases in a single batch wastes either compute or bandwidth, motivating disaggregated serving architectures.

Phase	Computation	Memory Access	Bottleneck	Optimal Batch
Decode	$\mathcal{O}(1)$ dense weight work plus $\mathcal{O}$(context length) attention over KV cache per token	Weight and KV-cache reads	Bandwidth	Large continuous batches of active tokens

Because decode is bandwidth-bound, hardware that shrinks the bytes moved per token attacks that phase's binding constraint directly.

> Systems Perspective 10.2: FP4 as a bandwidth lever for decode
>
> Native hardware support for narrow formats is the current manifestation of this approach. The Blackwell (B200) architecture introduces native **FP4 support**, making 4-bit inference a first-class hardware target rather than a framework-specific workaround. Halving weight and KV-cache bytes relative to FP8 raises token throughput and concurrent-sequence capacity without increasing clock speed, which is what loosens the decode-pool sizing in a disaggregated deployment. The product is point-in-time, and the exact cost-per-token gain depends on model architecture, kernel maturity, batch shape, and the fraction of time spent in prefill versus decode, so production systems should validate FP4 with workload-specific benchmarks rather than assuming a fixed multiplier.

The dichotomy creates a scheduling challenge: prefill operations are long-running and compute-intensive, while decode operations are short and bandwidth-limited. Mixing them in the same batch can cause interference. The GPU remains powered and allocated even when decode steps expose little useful compute, so low-utilization decode traffic can dominate energy cost unless the scheduler keeps the hardware busy with compatible work.

Chunked prefill[10] addresses this by breaking long prompts into fixed-size chunks that interleave with decode operations:

$$\text{Chunk latency} = \frac{\text{Chunk size}}{\text{Prefill throughput}}$$

With chunk size chosen to match decode iteration time, prefill and decode can share GPU resources without decode latency spikes.

Prefill-decode disaggregation takes this separation further by running prefill and decode on separate GPU pools. The prefill pool is optimized for compute intensity, using nodes with high TFLOP/s, moderate request batches, or large token chunks to maximize throughput without blocking decode. The decode pool is optimized for memory bandwidth, using nodes with maximum HBM capacity, the Tier 0 storage resource discussed in section 4.3.2, to handle thousands of concurrent autoregressive streams.

Independent scaling follows directly: prefill capacity scales with input volume while decode capacity scales with output volume. Crucially, this architecture relies on the high-speed, RDMA-enabled networking fabric established in Chapter 3. When a prefill finishes, the resulting KV cache (often megabytes of data) must be migrated to a decode node within the inter-token latency budget, typically under 10 ms. InfiniBand's low-microsecond latency and high bisection bandwidth are the physical enablers of this logical disaggregation, a displacement of overhead that trades communication bandwidth for specialized prefill and decode hardware.

Chunked prefill can be formalized into a practical scheduling algorithm that limits decode stalls from long prompts.

> Example 10.10: Sarathi: Chunked prefill implementation
>
> The Sarathi-Serve system (Agrawal et al. 2024) implements chunked prefills with stall-free scheduling:

[10] **Chunked Prefill**: Without chunking, a long prompt can block decode while prefill completes, violating inter-token latency SLOs for concurrent users. Chunking divides the prompt into smaller work units processed between decode iterations, bounding decode stalls at the cost of slightly longer total prefill time. The trade-off is a classic latency-vs.-throughput decision: interactive applications favor smaller chunks; batch workloads can favor larger ones.

Chunk sizing: Chunks are sized so prefill work can be interleaved with decode iterations. For example, if a deployment targets a 20 ms scheduling quantum and observes 10,000 prefill tokens/second, one chunk would process about 200 tokens. The right chunk size is workload- and hardware-dependent.

Interleaving schedule: Each GPU iteration processes either:

- One prefill chunk for a new request, OR
- One decode step for all active sequences

The schedule reduces decode stalls caused by long incoming prompts.

KV-cache handoff: When a prefill chunk completes, the scheduler has enough KV state to continue the request through later decode work. In disaggregated prefill/decode systems, any physical KV-cache transfer is a separate networking and placement budget rather than a guarantee supplied by Sarathi-Serve itself.

Performance impact:

- Without chunking: long prompts can create decode latency spikes for other active requests.
- With chunking: prefill work is broken into bounded units, so decode iterations can continue to make progress while long prompts enter the system.

Systems insight: Chunked prefill turns a long prompt from one blocking request into bounded work units that can coexist with decode traffic.

10.4.10 Adapter state and locality

KV-cache management makes one GPU safe for many concurrent requests; multi-tenant adapter serving asks the same serving process to manage another kind of resident state. The KV cache is per-token attention state. A LoRA adapter is per-tenant weight delta. Both must be available at the moment a decode step runs, and both can erase batching gains if the scheduler ignores locality. As serving platforms scale to support thousands of concurrent users on a single foundation model, personalization introduces a new memory bottleneck. Low-Rank Adaptation (LoRA) stores a small set of user- or task-specific adapter weights that modify a shared base model without copying the full model. When 10,000 users each have a unique LoRA adapter applied to the same 140 GB base model, replicating the base weights for each user is physically impossible. Instead, the base model remains pinned in HBM, while user-specific adapters are brought into the fast on-chip working set as needed.

Systems Perspective 10.3: The context switch of machine learning

This multi-tenant serving pattern shifts the fundamental performance constraint from compute throughput to adapter locality. As the continuous batching scheduler interleaves requests from different users, adapter weights may have to move repeatedly from HBM into SRAM or registers before each generation step. The effect is similar to an operating-system context switch: the accelerator has work to do, but it first waits for the state belonging to the next tenant. If the adapter swap latency exceeds the compute time of the generation step, the GPU compute cores stall. Efficient multi-tenancy therefore batches requests by active adapter when possible and coordinates adapter placement with KV-cache paging so personalization does not erase the benefit of sharing the base model.

The same locality problem reappears at a larger scale when the model itself exceeds single-GPU capacity. For a 175-billion-parameter foundation model, even the weights must be split across multiple devices, so inference becomes a sharding problem before it becomes a scheduling problem.

10.5 Model Sharding for Inference

A 70-billion-parameter model requires over 140 GB of VRAM to hold its weights in FP16, exceeding a single 80 GB GPU's capacity. To serve this model, we must slice its architecture across multiple GPUs that act together as a single logical replica. Adapting distributed training parallelism techniques, specifically tensor and pipeline parallelism, for low-latency inference introduces a distinct set of constraints.

10.5.1 When sharding becomes necessary

Model sharding for inference is driven by two distinct requirements. Table 10.21 identifies the memory and latency constraints that necessitate sharding:

The first driver is memory capacity. A model that cannot fit in single-GPU memory must be sharded regardless of performance considerations. For a model with P parameters at precision b bits, the weight memory is calculated by equation 10.6:

$$\text{Memory}_{\text{weights}} = P \times \frac{b}{8} \text{ bytes} \tag{10.6}$$

A 70-billion-parameter model in FP16 (16 bits) requires:

$$\text{Memory} = 70 \times 10^9 \times \frac{16}{8} = 140 \text{ GB}$$

The result exceeds the 80 GB capacity of an H100 GPU, requiring at minimum 2-way sharding.

The second driver is latency. Even when a model fits in memory, sharding can reduce latency by parallelizing computation. Equation 10.7 formalizes the potential speedup as a function of parallelization efficiency:

$$T_{\text{parallel}} = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}} \tag{10.7}$$

where N is the number of devices in the sharding group, $T_{\text{comm}}(N)$ is the communication overhead, $T_{\text{sync}}(N)$ is synchronization overhead, and T_{overlap} is communication hidden behind useful compute. In the common no-overlap simplification with negligible extra synchronization, this reduces to $T_{\text{compute}}/N + T_{\text{comm}}(N)$. Sharding provides latency benefit only when the nonoverlapped overhead is smaller than the time saved through parallelization.

Table 10.21: Sharding Triggers: Different constraints lead to different sharding requirements and strategies.

Sharding Trigger	Model Examples	Minimum Sharding	Strategy
Memory (weights)	Llama-70B (140 GB)	2-way	Tensor or pipeline
Memory (KV cache)	GPT-4 (long context)	4–8 way	Tensor (for cache)
Memory (embeddings)	DLRM (100 TB)	1000+ way	Embedding sharding
Latency	Any large model	Varies	Tensor parallelism

10.5.2 Tensor parallelism

Tensor parallelism (Shoeybi et al. 2019) distributes individual layers across multiple devices, enabling parallel computation within each layer. The column-row partitioning scheme introduced for training in Chapter 5 applies here: splitting the first linear layer by columns and the second by rows requires only one AllReduce per transformer block. For transformer models, the primary target is the attention mechanism and feed-forward layers, which contain the majority of computation.

The multi-head attention computation naturally partitions across attention heads. For a model with N_{heads} attention heads distributed across tensor-parallel degree t, each device computes N_{heads}/t heads:

$$\text{Attention}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \text{ for heads } i \in \{1, \ldots, N_{\text{heads}}/t\}$$

After computing local attention, an AllReduce operation (covered in Chapter 6) combines results across devices, adding communication overhead proportional to the activation size divided by the interconnect bandwidth.

The feed-forward layer (typically two linear transformations with activation) partitions along the hidden dimension. For the first linear layer, columns are distributed; for the second, rows are distributed. This column-row partitioning requires only one all-reduce per feed-forward block.

The communication pattern for tensor-parallel inference follows a two-phase synchronization per layer. Figure 10.12 illustrates this flow:

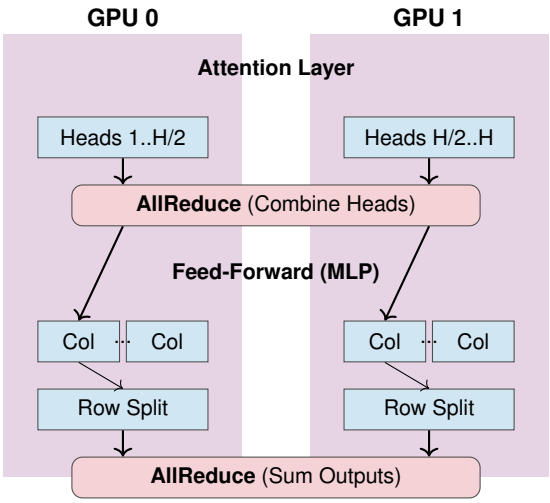


Figure 10.12: Tensor Parallelism for Inference: Computation is distributed across devices by splitting tensor operations. Attention heads are partitioned across GPUs, requiring an AllReduce operation to synchronize results. Feed-forward networks use a column-row splitting strategy that requires only one AllReduce synchronization per block. This approach reduces latency for large models but introduces communication overhead that demands high-bandwidth interconnects like NVLink.

The inference time with tensor parallelism follows equation 10.8:

$$T_{\text{inference}} = \frac{T_{\text{compute}}}{t} + 2 \times T_{\text{allreduce}}\left(\frac{A}{t}\right) \tag{10.8}$$

where T_{compute} is the sequential compute time, t is the tensor-parallel degree, and A is the activation size being reduced. The factor of 2 accounts for the two all-reduce operations per transformer layer (attention and feed-forward).

Example 10.11: Tensor parallelism for Llama-70B

Consider serving Llama-70B with the following configuration: (Touvron et al. 2023)

Model specifications:

- Parameters: 70B
- Hidden dimension: 8,192
- Attention heads: 64
- Layers: 80

Memory per GPU (weight only, FP16):

$\text{Memory}_{70B} = 70 \times 10^9 \times 2 = 140$ GB

Minimum sharding: 2-way (140 GB/80 GB per H100)

Recommended sharding: 8-way for optimal latency

Table 10.22 decomposes the per-layer cost of 8-way tensor parallelism on 8× H100 (NVLink interconnect) into compute and AllReduce phases, exposing the realized 6.9× speedup as compute scaling minus a small communication tax.

Table 10.22: 8-Way Tensor Parallelism Per-Layer Breakdown: Sequential vs. 8-way tensor-parallel timings for attention, feed-forward, and AllReduce phases of one Llama-70B transformer layer, decomposing the realized 6.9× speedup into compute scaling and communication tax.

Component	Sequential (1 GPU)	8-way TP	Speedup
Attention compute	12 ms	1.5 ms	8×
AllReduce (attention)	0 ms	0.3 ms	N/A
Feed-forward compute	18 ms	2.25 ms	8×
AllReduce (FF)	0 ms	0.3 ms	N/A
Total per layer	**30 ms**	**4.35 ms**	**6.9×**

Layer-stack estimate (80 layers on the same prefill workload):

- Sequential: 2,400 ms for the full prefill pass
- 8-way TP: 348 ms for the full prefill pass

The 6.9× speedup (vs. theoretical 8×) reflects communication overhead. At 900 GB/s, moving a 8 MB activation payload would take only about 0.01 ms by raw bandwidth math; the 0.3 ms budget shown here includes collective launch, synchronization, ring protocol, and framework overheads.

Time-to-first-token (1024-token prompt):

- Prefill compute: ~348 ms on 8-way TP (compute-bound, near-linear scaling minus communication overhead)
- Total TTFT: ~400 ms with preprocessing and serving overhead

Systems insight: Tensor parallelism buys latency by spending communication. It works for Llama-70B only because the interconnect keeps the collective tax small relative to the compute saved by sharding.

10.5.3 Pipeline parallelism for inference

Pipeline parallelism distributes layers across devices sequentially, with each device handling a subset of layers. Unlike tensor parallelism, there is no synchronization within a layer, only between pipeline stages.

For inference, pipeline parallelism creates bubbles differently than in training. Figure 10.13 contrasts single-request latency (bubble-dominated) with pipelined throughput (bubble-amortized):

For a single request, pipeline parallelism provides no latency benefit: the request must traverse all stages sequentially. The pipeline fill time equals the sequential execution time.

However, pipeline parallelism enables throughput scaling through pipelining multiple requests:

```
Time →
Device 0: [Req1] [Req2] [Req3] [Req4] ...
Device 1:        [Req1] [Req2] [Req3] [Req4] ...
Device 2:               [Req1] [Req2] [Req3] [Req4] ...
Device 3:                      [Req1] [Req2] [Req3] [Req4] ...
```

Once the pipeline is full, throughput equals p times single-stage throughput, where p is the number of pipeline stages. The steady-state latency remains approximately the single-device latency (sum of all stage times), but throughput scales with parallelism.

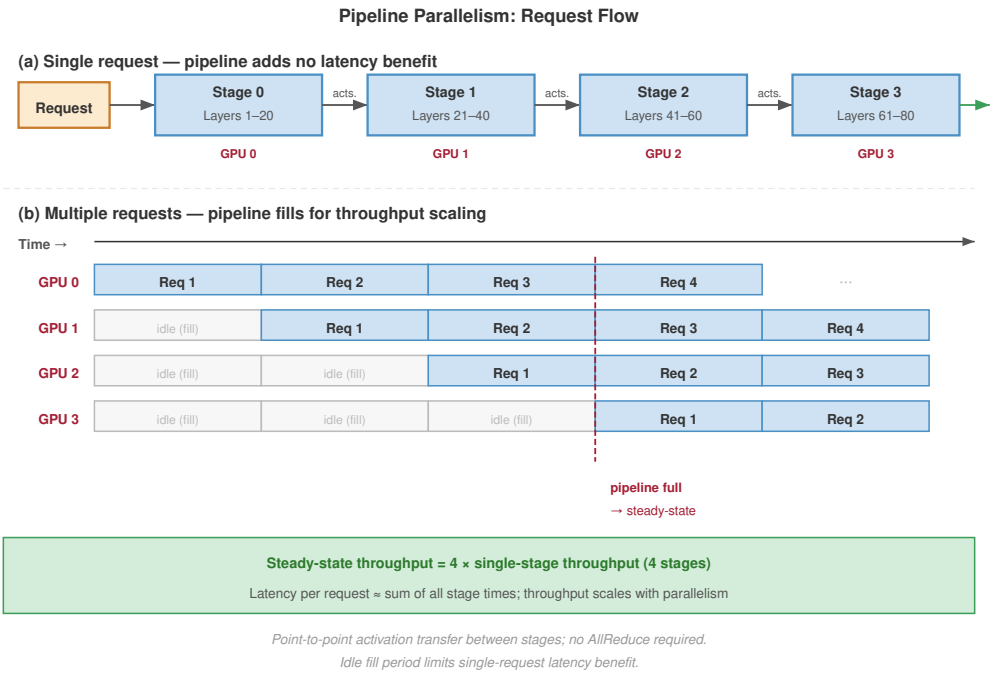


Figure 10.13: Pipeline Parallelism Bubbles: For a single inference request (top), pipeline parallelism offers no latency benefit as the request must traverse all four stages sequentially. When processing multiple concurrent requests (bottom), the pipeline fills and throughput scales with the number of stages. This makes pipeline parallelism ideal for high-throughput batch processing but less suitable for latency-critical interactive serving.

Pipeline parallelism is appropriate for inference in three scenarios:

- Memory constraints require sharding but latency requirements are relaxed
- Throughput matters more than individual request latency
- Network bandwidth between devices is limited (only point-to-point communication)

Table 10.23 captures the trade-offs between these two sharding approaches:

Table 10.23: Pipeline vs. Tensor Parallelism: Each strategy has distinct trade-offs in latency, throughput, and implementation complexity.

Aspect	Tensor Parallelism	Pipeline Parallelism
Single-request latency	Reduced by $\sim t\times$	No improvement
Throughput	$t\times$	$p\times$ (when pipelined)
Communication pattern	AllReduce (bandwidth-intensive)	Point-to-point (latency-sensitive)
Memory efficiency	Activations replicated	Activations passed along
Complexity	Higher (requires custom kernels)	Lower (layer-level partitioning)

The decision rule is latency first, topology second. Tensor parallelism is the right default when single-request latency matters and NVLink-class bandwidth is available; pipeline parallelism is a throughput tool for relaxed-latency workloads or placements where only point-to-point stage communication is affordable.

10.5.4 Expert parallelism for MoE models

Mixture-of-experts (MoE) models, such as DeepSeek-V3 or Mixtral, introduce unique serving challenges beyond those of dense models. In an MoE transformer layer, the feed-forward network (FFN) is replaced by multiple parallel "expert" sub-networks and a lightweight router that selects a subset of experts for each token.

The MoE architecture enables scaling the total parameter count while keeping the per-token compute constant. However, serving such models at scale requires expert parallelism (EP), where different experts are hosted on different GPUs.

10.5.4.1 MoE economics and capacity planning

The economics of MoE serving create a paradox: *total parameter count* determines the minimum hardware count (for memory), while *active parameter count* determines the compute cost per token. This makes MoE models efficient in compute but expensive in terms of VRAM "rent."

The performance advantages are striking. During autoregressive decode at batch size 1, the dominant cost is reading weights from HBM. A dense 400B model in FP16 reads 800 GB per step. DeepSeek-V3, despite having more total parameters, reads only the 37B active parameters per step (approximately 74 GB in FP16), a 10.8× reduction in per-token bandwidth. The compute savings are proportional: 10.8× fewer FLOPs per token.

The trade-off is memory capacity: all experts must reside in memory even though only a fraction are active at any time. DeepSeek-V3's full model in FP16 requires approximately 1342 GB, necessitating distribution across many GPUs.



MoE keeps every expert resident but activates few per token.

> **Napkin Math 10.6: MoE capacity planning**
>
> **Problem**: A team deploys DeepSeek-V3 (671B total parameters, 37B active per token) for a chatbot application. The model uses FP8 weights (1 byte per parameter). The cluster has 8-GPU nodes, each with 8× H100 GPUs (80 GB HBM per GPU, 640 GB per node). How many nodes are needed, and what is the per-token decode latency?
>
> **Math**:
>
> 1. **Memory**: Total weight memory = 671 GB. A single node (640 GB) is insufficient. 2 provide 1,280 GB, leaving enough room for KV caches.
> 2. **Latency**: Each decode step reads 37B active parameters (37 GB). Distributed across 16 GPUs, each reads about 2.3 GB. At 3.35 TB/s HBM bandwidth, $t_{\text{read}} \approx 0.69$ ms. AllToAll routing adds about 0.3 ms.
> 3. **Floor**: Estimated decode latency ≈ 1.0 ms/token, or 1,000 tokens/s at batch size 1.
>
> **Systems insight**: MoE allows a 671B model to approach the per-token bandwidth cost of a much smaller dense model, but capacity planning is still driven by the full parameter set that must remain resident.

10.5.4.2 Expert parallelism and load balancing

Distributing MoE models across GPUs introduces expert parallelism: each GPU holds a subset of experts, and tokens are routed to the GPU holding their assigned expert. In an MoE layer, a gating network selects k experts (out of E total) for each token:

$$\text{Output} = \sum_{i \in \text{top-}k} g_i \cdot \text{Expert}_i(\text{input})$$

Expert parallelism distributes experts across devices, with each device hosting E/N experts, where N is the number of expert-parallel devices. Figure 10.14 traces the routing, dispatch, and gather operations:

The communication pattern differs from tensor parallelism: instead of all-reduce (*same data to all devices*), expert parallelism uses all-to-all (*different data to different devices* based on routing). This AllToAll pattern creates a critical system challenge: load balancing. If the router consistently sends

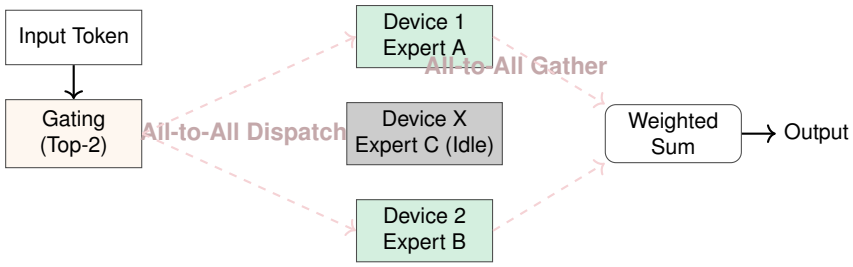


Figure 10.14: Mixture-of-Experts (MoE) Routing: Expert parallelism distributes "experts" across different devices. For each token, a gating mechanism selects the top-k experts. An AllToAll communication step dispatches tokens to the devices hosting their selected experts (1). Experts process the tokens in parallel (2). A second AllToAll step gathers the results back to the original device (3). This pattern enables massive model capacity but introduces all-to-all communication overhead.

more tokens to certain experts than others, those experts' GPUs become bottlenecks while other GPUs sit idle. Three mechanisms address load imbalance:

- **Auxiliary loss**: An additional term in the training loss penalizes uneven expert utilization by rewarding uniform routing probabilities.
- **Capacity factor**: Each expert has a maximum number of tokens it will accept per batch (typically 1.25× the fair share). Tokens exceeding this are dropped or rerouted.
- **Expert buffering**: Tokens are buffered and processed in subsequent iterations if experts are at capacity, smoothing out load spikes at the cost of latency variance.

These mechanisms keep expert utilization balanced enough that routing remains a throughput advantage instead of a new straggler source.

10.5.4.3 Routing failure modes

Router behavior directly impacts both system performance and model quality. Expert collapse is a training-time router failure in which the router converges to a small subset of experts, leaving others undertrained; the served model then behaves like a small dense model with a massive memory footprint. Routing instability is another training-time failure: assignments oscillate, preventing experts from specializing. Serving introduces runtime overload conditions as well. Distribution shift can make a router trained on formal text overload specific "grammar experts" when served colloquial chat data, creating hotspots. Token dropping occurs when capacity-limited experts drop tokens, skip their computation, and rely only on the residual connection, degrading output quality.

These failure modes follow from the basic MoE serving trade-off[11]: computation is dynamically routed to different experts based on input, so the routing pattern becomes a systems workload rather than only a model-internal choice. Popular models like Mixtral (A. Q. Jiang et al. 2024) use MoE to achieve high capacity with lower inference cost.

11 **MoE (Mixture-of-Experts) Serving Trade-Off**: Only a subset of experts activate per token (typically 2 of 8–16), keeping per-token FLOPs low while total parameter count reaches hundreds of billions. The serving constraint: all expert weights must reside in memory even though most are idle on any given token, inflating memory footprint 4–8× relative to the active compute. This forces expert parallelism across devices with AllToAll communication at every MoE layer.

Example 10.12: Expert parallelism for Mixtral-8x7B

Mixtral-8x7B uses 8 experts per MoE layer with top-2 routing:

Model characteristics:

- Total parameters: 46.7B (but only ~12.9B active per token)
- Experts per layer: 8
- Active experts per token: 2 (top-k = 2)
- MoE layers: Every feed-forward layer (each FFN replaced by a 8-expert MoE block)

Sharding strategy (4-way expert parallelism):

- Experts 0-1 on Device 0
- Experts 2-3 on Device 1
- Experts 4-5 on Device 2

- Experts 6-7 on Device 3

Mechanism: The per-token communication pattern has four steps:

1. Gating: Determine which 2 experts to use (~0.1 ms)
2. AllToAll dispatch: Send token to devices hosting selected experts (~0.2 ms)
3. Expert compute: Process token through selected experts (~1 ms each, parallel)
4. AllToAll gather: Collect results back (~0.2 ms)

Total MoE layer time: ~1.5 ms (vs. ~4 ms for equivalent dense layer)

Systems insight: Routing imbalance directly erodes GPU utilization and throughput across the eight experts; table 10.24 quantifies the degradation. Production systems monitor routing statistics and may retrain or fine-tune gating to improve balance.

Table 10.24: MoE Routing Load Balance vs. Throughput: GPU utilization and throughput impact as routing imbalance grows across the eight experts of a Mixtral-8x7B MoE layer with 4-way expert parallelism.

Routing Distribution	GPU Utilization	Throughput Impact
Perfectly balanced	100%	Baseline
Moderate imbalance (20%)	83%	-17%
Severe imbalance (50%)	67%	-33%

Expert parallelism represents the peak of model sharding complexity, requiring tight integration between the model architecture, the routing algorithm, and the physical interconnect topology. The next section examines how recommendation systems use similar sharding techniques for massive embedding tables.

10.5.5 Embedding sharding for recommendation systems

Recommendation systems typically contain embedding tables that dwarf dense model weights in size. Meta's DLRM reference architecture uses model parallelism over embedding tables to mitigate memory constraints while scaling dense layers separately (Naumov et al. 2019). At production scale, this embedding-heavy structure requires sharding strategies distinct from tensor or pipeline parallelism.

Row-wise sharding partitions embedding tables by row (entity ID):

$$\text{Shard}_i = \{e_j : \text{hash}(j) \bmod N_{\text{shards}} = i\}$$

Each shard contains approximately $N_{\text{entities}}/N_{\text{shards}}$ embeddings, where N_{entities} is the total number of entities and N_{shards} is the shard count.

Column-wise sharding partitions each embedding vector across devices:

$$e_j = [e_j^{(0)}, e_j^{(1)}, ..., e_j^{(N_{\text{shards}}-1)}]$$

Each device stores a slice of every embedding. Hybrid sharding combines both approaches: frequently accessed embeddings are column-sharded for faster access, while the long tail uses row sharding. The choice of embedding sharding strategy depends on lookup patterns and communication overhead. Table 10.25 compares row-wise, column-wise, and hybrid approaches:

Table 10.25: Embedding Sharding Strategies: Different strategies trade off lookup locality against load balance.

Sharding Strategy	Lookup Pattern	Communication	Best For
Row-wise	Single device per lookup	AllToAll gather	Uniform access patterns
Column-wise	All devices per lookup	AllGather	Hot embeddings

Sharding Strategy	Lookup Pattern	Communication	Best For
Hybrid	Varies by embedding	Mixed	Production RecSys

The trade-off is locality against load balance. Row-wise sharding keeps each embedding vector whole on one device, which localizes the lookup but needs an AllToAll gather to collect vectors scattered by entity ID; column-wise sharding parallelizes every lookup across devices at the cost of an AllGather; and hybrid sharding splits the difference, replicating hot embeddings column-wise while row-sharding the cold tail. Figure 10.15 shows the three layouts, where the row-wise "network gather" is the AllToAll collective named in table 10.25.

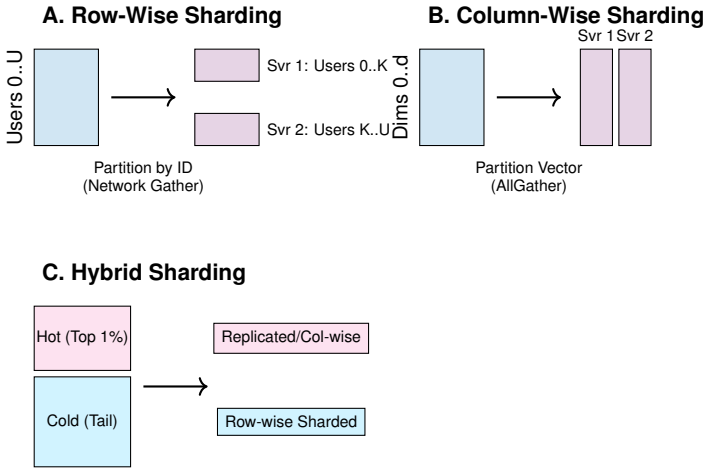


Figure 10.15: Embedding Sharding Strategies: Row-wise sharding places complete embedding vectors on specific servers based on entity ID, requiring a network gather for lookup. Column-wise sharding splits each vector across all servers, allowing parallel local lookups followed by an AllGather, which is efficient for popular "hot" embeddings. Hybrid sharding combines these approaches, using column sharding for hot items and row sharding for the "cold" long tail to balance load and memory.

All three sharding strategies operate at massive scale in production. Meta's recommendation infrastructure provides a concrete example of how row-wise, column-wise, and hybrid approaches combine to serve trillion-entity embedding tables.

Lighthouse 10.1: Archetype B (DLRM at Scale): Embedding sharding at Meta

For the Archetype B recommendation workload, Meta's infrastructure demonstrates embedding sharding at extreme scale:

Scale:

- Embedding tables: 100+ TB total
- Unique entities: 10+ trillion
- Embedding dimension: 128–256
- Shards: 1,000+ servers

These scale bullets describe infrastructure-level totals, not one dense resident table in which every unique entity has a 128–256-dimensional vector. A dense INT8 table with 10 trillion entities and 128 dimensions would require at least 1,280 TB, while 100 TB across the same entity count averages only 10 bytes per entity. The production system therefore relies on compression, tiering, cold storage, and uneven entity coverage rather than a single fully materialized dense matrix.

Sharding strategy:

- **Hot embeddings** (top 1 percent by access frequency): Replicated across all shards
- **Warm embeddings** (next 10 percent): Column-sharded with 8-way parallelism
- **Cold embeddings** (remaining 89 percent): Row-sharded with consistent hashing

Each inference request requires approximately 5,000 embedding lookups. Without optimization, this would require 5,000 network round trips. Instead, the system applies several optimizations. Batch accumulation collects lookups for 1 ms. Lookup deduplication removes duplicate entities across requests. Shard-aware batching groups lookups by destination shard. Parallel dispatch sends batched requests to all shards simultaneously. Streaming assembly reconstructs embeddings as responses arrive.

The order-of-magnitude effect of these optimizations on round-trip count, lookup latency, and bandwidth appears in table 10.26:

Table 10.26: Meta embedding sharding lookup performance: Network round-trip, lookup-latency, and bandwidth measurements for the recommendation lookup path before and after batching plus shard-aware dispatch, showing the order-of-magnitude effect of these optimizations.

Metric	Without Optimization	With Optimization
Network round trips	5,000	1 (batched)
Lookup latency	50 ms	2 ms
Network bandwidth	10 Gbps	40 Gbps (burst)

The lighthouse lesson is that DLRM-scale serving is dominated by sparse embedding placement and lookup coordination, not by dense accelerator FLOP/s.

10.5.6 Hybrid sharding strategies

Production systems often combine multiple sharding strategies because no single axis satisfies every inference constraint. Hybrid sharding composes memory capacity, latency, and communication requirements: one axis may keep weights resident, another may reduce per-token latency, and a third may route sparse components without forcing dense layers onto the wrong device. Three recurring combinations illustrate the pattern:

Tensor and pipeline parallelism combine when models require both memory distribution and latency reduction:

```
8 GPUs organized as 2 by 4 (pipeline stages by tensor parallel):

Stage 0 (Layers 1-40):  TP across GPUs 0,1,2,3
Stage 1 (Layers 41-80): TP across GPUs 4,5,6,7
```

The combination achieves 4× latency reduction (from TP) while handling models requiring 8-way sharding for memory.

Expert and tensor parallelism combine for MoE models where individual experts are large:

```
Mixtral with large experts:

- Expert parallelism: Distribute 8 experts across 8 GPU groups
- Tensor parallelism: Each expert spread across 2 GPUs
- Total GPUs: 16
```

Embedding and dense parallelism serve recommendation models with both large embeddings and large dense components:

```
DLRM-scale model:

- Embedding sharding: 1,000 shards across CPU servers
```

```
- Dense model: 8-way tensor parallel across GPUs
- Communication: Embeddings gathered to GPU, processed, returned
```

Hybrid sharding buys fit or latency by adding new communication patterns at every boundary between shards. The next question is whether the communication tax stays inside the latency and throughput budget.

10.5.7 Communication overhead analysis

The practical speedup from sharding depends critically on communication efficiency. Each sharding strategy has characteristic communication patterns with different bandwidth and latency requirements.

Equation 10.9 quantifies AllReduce communication time for tensor parallelism, where data is combined from all devices with the result available on all devices.

$$T_{\text{allreduce}} \approx 2(N-1)\alpha + \frac{2(N-1)}{N} \times \frac{M}{\beta} \tag{10.9}$$

where N is the number of devices, M is the collective payload size, α is startup latency, and β is the interconnect bandwidth. The factor of 2 accounts for the reduce-scatter and all-gather phases. Section D.2.1 derives this ring-AllReduce cost and its bandwidth-optimal $2(N-1)/N$ scaling, establishing why the per-layer communication tax stays bounded as the tensor-parallel degree grows.

Equation 10.10 expresses the simpler point-to-point communication for pipeline parallelism, where data flows from one device to the next.

$$T_{\text{p2p}}(n) = \alpha + \frac{n}{\beta} \tag{10.10}$$

where n is the point-to-point payload size, α is the network latency, and n/β is the transfer time. Equation 10.11 captures the more complex AllToAll communication for expert parallelism, where each device exchanges distinct data with every other device.

$$T_{\text{alltoall}} = (N-1) \times \left(\alpha + \frac{M/N}{\beta}\right) \tag{10.11}$$

Both equations make clear that the achievable bandwidth β and latency α of the underlying interconnect determine real-world sharding performance. Production hardware differs enough on both axes that the interconnect choice can determine whether a sharding plan is viable.

Communication overhead depends heavily on the interconnect technology. Table 10.27 records the physical bandwidth, latency, and intended scope of each candidate interconnect:

Table 10.27: Datacenter interconnect specs: Bandwidth, latency, and intended scope for NVLink, PCIe Gen5, InfiniBand HDR, and 100G Ethernet, establishing the physical limits that bound achievable sharding speedups.

Interconnect	Bandwidth	Latency	Use Case
NVLink (H100)	900 GB/s	500 ns	Intra-node TP
PCIe Gen5	64 GB/s	1 µs	Intra-node (no NVLink)
InfiniBand HDR	200 Gb/s (25 GB/s)	7 µs	Inter-node
Ethernet 100G	100 Gb/s (12.5 GB/s)	50 µs	Inter-node (commodity)

Table 10.28 translates those bandwidths into AllReduce time for an 8-way tensor-parallel layer (activation 8 MB per all-reduce; batch=1, hidden=8,192) as a fraction of a 30 ms transformer-layer budget:

Table 10.28: AllReduce cost by interconnect: AllReduce time for an 8 MB activation across NVLink, InfiniBand, and 100G Ethernet, with each as a fraction of a 30 ms transformer-layer budget, showing why intra-node TP requires NVLink-class bandwidth.

Interconnect	AllReduce Time	Share of 30 ms Layer
NVLink	0.03 ms	0.10%
InfiniBand	0.56 ms	1.9%
100G Ethernet	1.12 ms	3.7%

The engineering boundary is local: NVLink makes tensor parallelism efficient within a node, InfiniBand can make cross-node tensor parallelism acceptable for carefully chosen workloads, and commodity Ethernet is usually too slow for latency-sensitive inference.

NVLink bandwidth[12] has evolved significantly over GPU generations.

12 **NVLink Bandwidth Evolution**: From 160 GB/s bidirectional on early NVLink implementations to 900 GB/s on Hopper-class GPUs to 1.8 TB/s on Blackwell-class GPUs—roughly a 10× improvement across three hardware generations. This bandwidth growth is what made intra-node tensor parallelism across 8 GPUs practical with less than 5 percent communication overhead, directly determining the maximum model size servable without crossing the slower cross-node interconnect.

10.5.8 Sharding strategy selection

The choice of sharding strategy depends on model architecture and deployment priorities. Table 10.29 evaluates each approach across four critical factors:

Table 10.29: Sharding Strategy Selection Guide: Match strategy to deployment priorities and constraints.

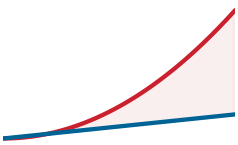
Factor	Tensor Parallel	Pipeline Parallel	Expert Parallel	Embedding Shard
Latency priority	Best	Worst	Moderate	N/A
Throughput priority	Good	Best (pipelined)	Good	Best
Interconnect limited	Poor fit	Good fit	Moderate	Good fit
Implementation effort	High	Low	Moderate	High

Strategy selection starts from the deployment bottleneck. Use tensor parallelism when latency dominates and the interconnect can sustain frequent collectives, pipeline parallelism when throughput matters more than per-request latency, expert parallelism when MoE routing defines the model structure, and embedding sharding when large sparse tables dominate capacity.

Once we have constructed these massively sharded, multi-GPU replicas, dozens of them must run in parallel to handle global traffic. The challenge shifts from the internal mechanics of a single replica to the traffic control layer that routes millions of user queries across the fleet.

10.6 Load Balancing and Request Routing

If ten model replicas are actively serving traffic, and a new request arrives asking for a 5,000-word document summarization, sending it to a replica that is already overloaded will trigger a massive latency spike for every user assigned to that node. Simple round-robin load balancing fails catastrophically for generative ML. We must employ intelligent request routing that understands the internal memory and queue states of every replica in the fleet.



Two-choice routing flattens the tail imbalance of random placement.



Lighthouse 10.2: Archetype B (DLRM at Scale): The tail at scale

Archetype B (DLRM at Scale) is the canonical victim of tail latency (Dean and Barroso 2013). Processing 10 million QPS means that a 1-in-10,000 latency spike happens 1,000 times every second. For Archetype B (DLRM at Scale), sophisticated load balancing, such as routing each request to the less-loaded of two sampled replicas, is mandatory to suppress these outliers; simple Round-Robin would allow queues to build up, causing the 99th percentile latency to collapse into unacceptable slowness.

10.6.1 Load balancing principles

Load balancing serves two primary goals that sometimes conflict: latency minimization and utilization maximization. Latency minimization routes requests to the replicas that can serve them fastest, considering current queue depth and processing time; utilization maximization spreads load evenly to avoid both idle replicas and overloaded replicas. The tension arises because latency-optimal routing may concentrate load on fast replicas, reducing their performance and leaving other replicas underutilized.

Evaluation therefore needs four signals that cover both tail latency and load balance:

- **Maximum queue length**: This determines worst-case latency.
- **Load variance**: This measures how evenly work is distributed.
- **Utilization spread**: This shows whether some replicas are idle while others are saturated.
- **Decision overhead**: This captures the cost of making the routing choice.

A policy that optimizes only one of these signals can look healthy in aggregate while sending unlucky requests into the tail.

10.6.2 Round-robin and random assignment

The simplest load-balancing strategies assign requests without considering server state. Round-robin sends request 1 to server 1, request 2 to server 2, and so on, which gives perfect distribution only when servers are homogeneous and request processing times are identical. Random assignment selects a server uniformly at random for each request; with many requests it converges to an even mean distribution, but its variance is higher than round-robin. Both strategies are reasonable baselines for identical servers, yet production serving rarely satisfies that assumption. Heterogeneous GPU generations, different memory configurations, variable request sizes, warmup behavior, and replicas nearing memory limits all make the state of the server fleet matter.

Under these realistic conditions, uninformed strategies perform poorly because unlucky assignments accumulate in the tail. The maximum queue length under random assignment follows the classical balls-into-bins result (Mitzenmacher 2001), shown in equation 10.12:

$$E[\text{max queue}] = \Theta\left(\frac{\log R}{\log \log R}\right) \tag{10.12}$$

where R is the number of servers or replicas. Evaluating the ratio at $R = 1{,}000$ gives $\log R / \log \log R \approx 6.9/1.9 \approx 3.6$, which the leading constant lifts to roughly 4–5 requests in the worst-case queue. This seems small, but the unlucky requests in long queues experience significantly higher latency. The power of two choices (principle 16) provides the theoretical response: one extra queue-length probe changes the tail behavior from $\mathcal{O}(\log R / \log \log R)$ to $\mathcal{O}(\log \log R)$.

10.6.3 The power of two choices

A foundational result in load balancing theory (Mitzenmacher 2001) shows that querying just two random servers before making a routing decision provides exponentially better load distribution than random assignment.[13]

The procedure is deliberately small, but the load signal must match the serving workload. For a homogeneous vision fleet, current queue length may be enough. For an LLM fleet, active tokens, estimated remaining decode work, and KV-cache pressure better approximate the work a replica has already accepted. Algorithm 10.2 states a representative routing loop: probe two replicas, compare normalized work, and route without polling the whole fleet.

[13] **Balls-into-Bins (Power of Two Choices)**: With random placement, maximum bin load is $\Theta(\log R / \log \log R)$; with two random choices and greedy selection, it drops to $\Theta(\log \log R)$. For a 1,000-server inference fleet, this reduces worst-case queue depth from roughly 5 to roughly 2, cutting P99 tail latency by nearly half while adding only the overhead of a single extra queue-length probe per request.

Algorithm 10.2 Power-of-two choices request routing

Require: replica set $\mathcal{S}$; request r; capacity weights w_s; per-replica work signal W_s (queue length, active tokens, predicted work, or KV-cache occupancy)
Ensure: a selected replica for r
1: sample two candidate replicas s_a, s_b from $\mathcal{S}$ ▷ *capacity-weighted*
2: query W_{s_a}, W_{s_b}; estimate incremental work $\hat{c}(r)$ (expected output tokens or KV pages)
3: $u_s \leftarrow (W_s + \hat{c}(r))/w_s$ for $s \in \{s_a, s_b\}$ ▷ *post-admission load*
4: $s^\star \leftarrow$ candidate with smaller u_s ▷ *ties: cache affinity*
5: route r to $s^\star$
6: update $s^\star$'s local admission state
7: **return** $s^\star$

The two probes and the one state update per request are $\mathcal{O}(1)$ coordination rather than an all-replica scan, and the tail-latency win materializes only when the load signal reflects true work: in LLM serving, request count alone can hide a 10-token request behind a 500-token one, while active-token and KV-cache signals expose the risk. The queue-length bound in equation 10.13 formalizes why this small change matters, reducing maximum queue length from $\mathcal{O}(\log R/\log\log R)$ to $\mathcal{O}(\log\log R)$:

$$E[\max \text{ queue}]_{\text{two choices}} = \Theta(\log\log R) \tag{10.13}$$

For 1,000 servers, random assignment produces a maximum queue of roughly 4–5 requests, while two choices reduces the maximum to roughly 2. The improvement is exponential: two choices with 1,000 servers achieves better balance than random assignment with just 10 servers.

Theorem 10.2: Power-of-two choices load-balancing bound

For R servers receiving independent requests, random assignment produces a maximum queue length of $\mathcal{O}(\log R/\log\log R)$ with high probability, while routing each request to the shorter of two randomly sampled queues reduces the maximum queue length to $\mathcal{O}(\log\log R)$.

The practical implication is that near-optimal load balancing does not require polling the whole fleet. Two probes avoid the R-probe cost of exact least-loaded routing, the improvement grows with system size, and the method remains simple enough to implement inside ordinary load balancers. Variants of power-of-two-choices appear in large-scale systems because they offer strong tail behavior without global polling.

The mechanism is intuitive: random assignment occasionally makes poor choices by routing to an already-busy server, and these mistakes compound. With two choices, the algorithm almost never makes the worst choice, avoiding the tail behavior that creates long queues. Mathematically, when m servers have queue length k, random assignment grows queue length $k+1$ with probability proportional to m/R. With two choices, that probability drops to $(m/R)^2$, creating a super-exponential decay in queue length distribution.

10.6.4 Weighted and adaptive load balancing

When servers have different capacities, naive load balancing creates imbalance. A mix of A100 GPUs and lower-capacity T4 GPUs receiving equal request rates will overload the T4 servers while leaving A100 capacity idle. Weighted round-robin corrects the mean assignment rate by routing requests proportional to server capacity:

$$\Pr(\text{route to server } i) = \frac{w_i}{\sum_j w_j}$$

where w_i is the weight, or capacity, of server i. Weighted two-choices applies the same idea to tail control: sample two servers with probability proportional to their weights, compare current load relative to capacity, and route to the lower relative load. The weighted-routing example makes this policy concrete for mixed GPU hardware.

Example 10.13: Heterogeneous GPU cluster

Scenario: A serving cluster mixes two GPU tiers and must route requests without overloading the slower tier.

Given:

- 10 NVIDIA H100 GPUs at 1000 QPS each.
- 20 NVIDIA A100 GPUs at 600 QPS each.
- Total capacity is 22,000 QPS; target traffic is 15,000 QPS.

At target traffic of 15,000 QPS, weighted assignment gives each H100 a weight of 4.5 percent and each A100 a weight of 2.7 percent. The expected per-server load becomes 681.8 QPS for H100s and 409.1 QPS for A100s, leaving the two server types at 68.2 percent and 68.2 percent utilization.

Without weighting: Equal distribution would send 500 QPS to every server, leaving H100s underutilized at 50 percent and A100s with reduced latency headroom at 83.3 percent.

Systems insight: Heterogeneous clusters need capacity-weighted routing. Equal request counts are not equal load when each hardware tier has a different service rate.

Static weights handle known capacity differences, but production replicas also change over time. Adaptive load balancing adjusts weights dynamically based on observed performance:

```
For each server i:
    latency[i] = exponential_moving_average(observed_latency)
    weight[i] = 1 / latency[i]  # Inverse latency weighting
```

Inverse latency weighting lowers the probability of routing to servers whose observed latency has risen, whether the cause is memory pressure, thermal throttling, request mix, or background work consuming resources.

10.6.5 Least-connections load balancing

An alternative to random selection is routing to the server with the fewest active connections or shortest queue. Least-connections maintains an active-request count for each server, routes a new request to the minimum count, increments on dispatch, and decrements on completion. For long-running requests, common in LLM serving, this outperforms round-robin because it accounts for work still in progress rather than only historical assignment order.

The challenge is maintaining accurate connection counts in a distributed system. A centralized counter gives a single source of truth but can become a bottleneck. Distributed counters with gossip scale better but may route using stale information. Sampled least-connections combines the idea with two choices by probing only a subset of servers and choosing the minimum.

Example 10.14: Least-connections for LLM serving

LLM inference has highly variable request durations because output length changes service time. A short 10-token response might take 500 ms, while a 500-token response might take 25 seconds, a 50× difference. With round-robin at 100 QPS across 10 servers, each server receives 10 requests per second regardless of whether it is already handling long responses. A server that receives several long requests falls behind while other servers sit idle. Least-connections routes new work away from those busy servers, so replicas that finish short requests receive new work immediately and load balances around actual work remaining.

Table 10.30 reports the observed improvement in production LLM serving across routing algorithms:

Table 10.30: LLM routing algorithm comparison: Observed p99 latency and load variance under round-robin, least-connections, and least-connections-plus-two-choices routing for highly variable LLM workloads, quantifying how routing choices reshape tail latency.

Algorithm	P99 Latency	Load Variance
Round-robin	45s	3.2 requests
Least-connections	28s	0.8 requests
Two-choices + LC	26s	0.5 requests

Systems insight: Least-connections reduces P99 by 38 percent; combining with two-choices provides additional improvement because routing follows current work rather than request count alone.

10.6.6 Consistent hashing for stateful routing

Many inference workloads maintain state that benefits from routing affinity. LLM conversations reuse KV cache from previous turns, recommendation sessions carry user context and recent interactions, and streaming inference may depend on model state from previous frames. For these workloads, routing the same user or session to the same server improves performance by avoiding cache misses and state reconstruction.

Consistent hashing[14] (Karger et al. 1997) maps requests to servers based on a hash of the routing key (user ID, session ID):

$$\text{server}(request) = \arg\min_{s \in \mathcal{S}_{\text{srv}}} \text{distance}(\text{hash}(key), \text{hash}(s))$$

where $\mathcal{S}_{\text{srv}}$ is the set of servers mapped onto the ring, and each request routes to the nearest server clockwise.

The important properties are determinism, minimal disruption, and balance. The same key always routes to the same server, adding or removing servers remaps only K/N_{servers} keys on average, and virtual nodes distribute load evenly enough to prevent one physical server from owning a disproportionate key range. For LLM serving, the most direct payoff is keeping each user's requests on the server that already holds their session state.

14 **Consistent Hashing**: A ring-based load distribution algorithm originally deployed at Akamai for CDN routing. Servers are mapped onto a virtual ring so that adding or removing a node remaps only K/N_{servers} keys on average. For LLM serving, this minimal-disruption property preserves KV cache locality during autoscaling events: without it, every scale-up would invalidate cached conversation state across the fleet, forcing expensive recomputation.

Example 10.15: Consistent hashing for KV cache affinity

Scenario: An LLM serving system maintains user-specific KV cache state across conversation turns.

Without affinity:

- User sends message, routed to Server A, KV cache built
- Next message routes to Server B (random)
- KV cache rebuilt from scratch, 500 ms penalty
- Average conversation: 10 turns, 4.5 seconds wasted on cache rebuilds

With consistent hashing:

- User ID hashed to Server A
- All messages from this user route to Server A
- KV cache reused across turns
- Rebuild only on server changes or cache eviction

Implementation with virtual nodes:

Each physical server has 100 virtual nodes on the hash ring, ensuring even distribution despite server heterogeneity.

```
Hash ring positions:
Server A: [0.01, 0.03, 0.07, 0.12, ...]  (100 positions)
Server B: [0.02, 0.05, 0.09, 0.15, ...]  (100 positions)
...

Request for user "alice":
hash("alice") = 0.0834
Nearest server clockwise: Server A (at 0.09)
```

Handling server failures: When Server A fails, its 100 virtual nodes drop from the ring and the affected requests reroute to the next server clockwise, so only $\sim 1/N_{\text{servers}}$ of all requests are touched.

Systems insight: Stateful serving turns load balancing into a cache-locality problem. The routing layer must preserve affinity without making failure recovery disruptive.

The same stateful routing machinery becomes a correctness boundary when cached state is user-specific.

War Story 10.1: When cache state crossed users

Context: ChatGPT used Redis Cluster through the redis-py client to cache user information and reduce database lookups in a high-volume serving system (OpenAI 2023).

Failure mode: Under a specific Asyncio connection-reuse bug, a request could receive cached data associated with another active user because a connection was recycled with unexpected response state.

Consequence: On March 20, 2023, OpenAI took ChatGPT offline while fixing the issue, and reported that some users could see another user's conversation title data and limited payment-related information.

Systems lesson: Stateful inference infrastructure needs isolation guarantees around caches, connection pools, and routing keys. A cache is part of the serving correctness boundary, not merely an optimization.

10.6.7 Request routing for sharded models

The sharding strategies examined in section 10.5 introduce routing complexity: a single inference request may require computation on multiple devices, necessitating coordination. The routing pattern depends on the sharding strategy.

Under tensor parallelism, each request is broadcast to all devices in the shard group. Each device processes its portion of each layer, and results are synchronized via AllReduce. Figure 10.16 shows this fan-out, compute, and gather pattern.

The key constraint visible in figure 10.16 is that every request requires an AllReduce synchronization across all devices in the shard group, making interconnect bandwidth the latency-determining factor rather than model size.

Under pipeline parallelism, each request traverses stages sequentially, with each stage forwarding activations to the next. Figure 10.17 illustrates how pipelining multiple requests achieves throughput scaling even though single-request latency equals the sum of all stage times.

The critical insight from figure 10.17 is that pipeline parallelism offers no single-request latency benefit: one request still traverses all stages sequentially. Throughput scales only when multiple concurrent requests fill the pipeline, reaching steady-state throughput proportional to the number of stages.

Under expert parallelism, each request is dispatched to the devices hosting its selected experts based on the gating decision. Two AllToAll communication steps bookend the expert compute, as shown in figure 10.18. Load imbalance is the critical challenge: when the gating function routes disproportionately to some experts, communication overhead becomes a bottleneck.

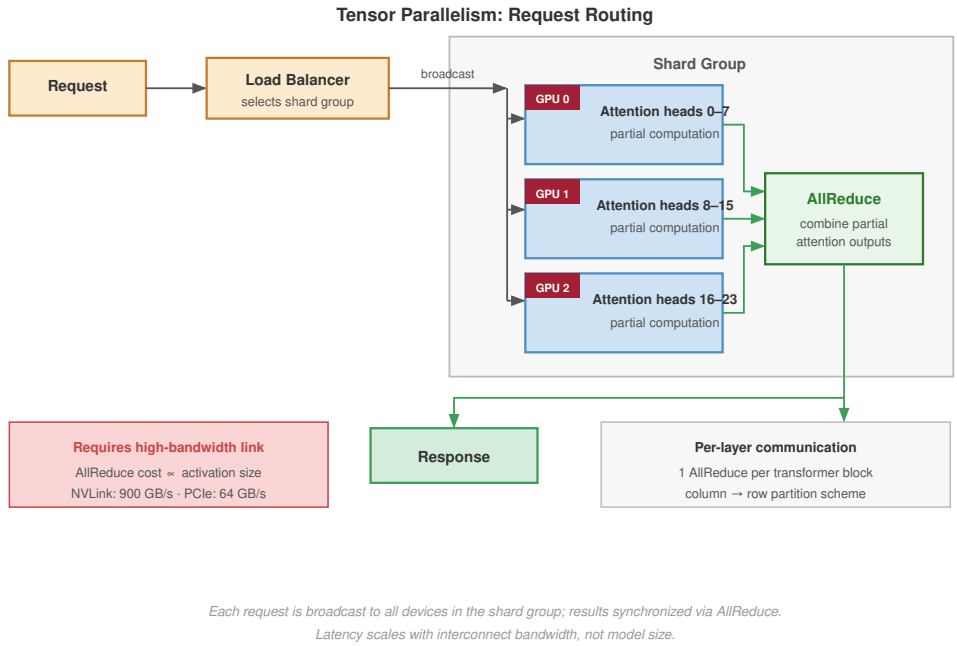


Figure 10.16: Tensor Parallelism Request Routing: A request is broadcast from the load balancer to all devices in the shard group. Each GPU computes its assigned attention head partition, then a single AllReduce synchronizes results before the response is returned. Interconnect bandwidth—not model size—determines latency.

As figure 10.18 illustrates, the two AllToAll communication steps make expert parallelism uniquely sensitive to load imbalance: if the gating function concentrates tokens on a few experts, those devices become bottlenecks while others sit idle.

When multiple shard groups provide horizontal scaling, the load balancer routes to groups rather than individual devices. Figure 10.19 shows this two-level hierarchy: standard load-balancing algorithms (round-robin, two-choices, consistent hashing) operate at the group level, while each group handles its own internal AllReduce locally.

The two-level hierarchy in figure 10.19 separates concerns: the load balancer treats each shard group as a single logical server using standard algorithms, while internal AllReduce coordination remains local to each group. Adding shard groups scales throughput linearly; adding GPUs per group reduces per-request latency.

10.6.8 Health checking and failover

Load balancers can route around failure only if the health signal matches the failure mode. For ML serving, a process that is alive may still be unusable because weights are not loaded, the GPU memory pool is exhausted, or the first request will trigger a long warm-up. Production systems therefore layer health checks from cheapest to most realistic.

Liveness probes verify that the server process is running:

```
GET /health/live
Response: 200 OK (process alive) or timeout (process dead)
```

Readiness probes go further, verifying the server can handle requests (model loaded, GPU initialized):

```
GET /health/ready
Response: 200 OK (ready to serve) or 503 (not ready)
```

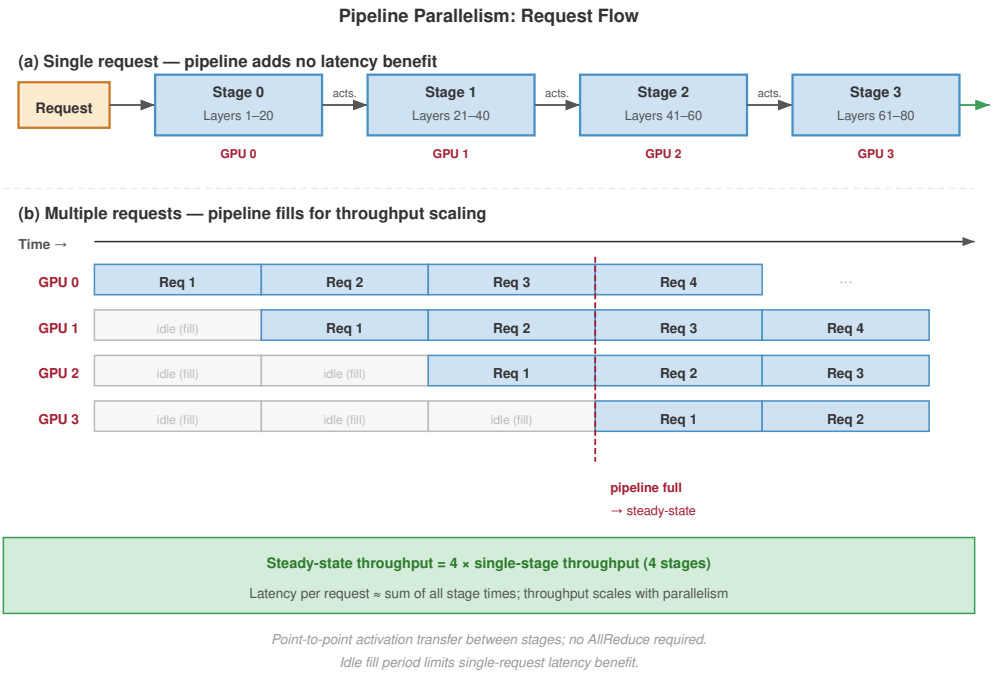


Figure 10.17: Pipeline Parallelism Request Flow: (a) A single request flows sequentially through four stages—no latency benefit from parallelism. (b) Multiple requests fill the pipeline: once steady state is reached, throughput scales with the number of pipeline stages, while per-request latency remains approximately constant.

Deep health checks verify that actual inference works by running a test request:

```
POST /health/inference
Body: {"prompt": "test"}
Response: 200 OK with valid output, or error
```

Deep health checks are the primary defense against a failure mode that does not exist in traditional web servers: silent hardware degradation. A GPU with degraded HBM modules may continue to respond to process-level probes while producing output corrupted by NaN propagation or silent bit errors in activation tensors. The replica returns HTTP 200 and passes every liveness and readiness probe, yet every generated sequence is nonsense. Deep health checks catch this by validating the tensor output of a known probe input against expected bounds—checking, for example, that logit distributions stay within a plausible entropy range and that no NaN or Inf values appear in the output. Standard health checks are necessary but insufficient for GPU-accelerated servers, which face these hardware-specific failure modes.

Example 10.16: Health checks for GPU inference

Scenario: GPU inference servers can pass ordinary process health checks while still being unable to serve real requests.

GPU memory pressure: Server may be alive but unable to allocate memory for new requests.

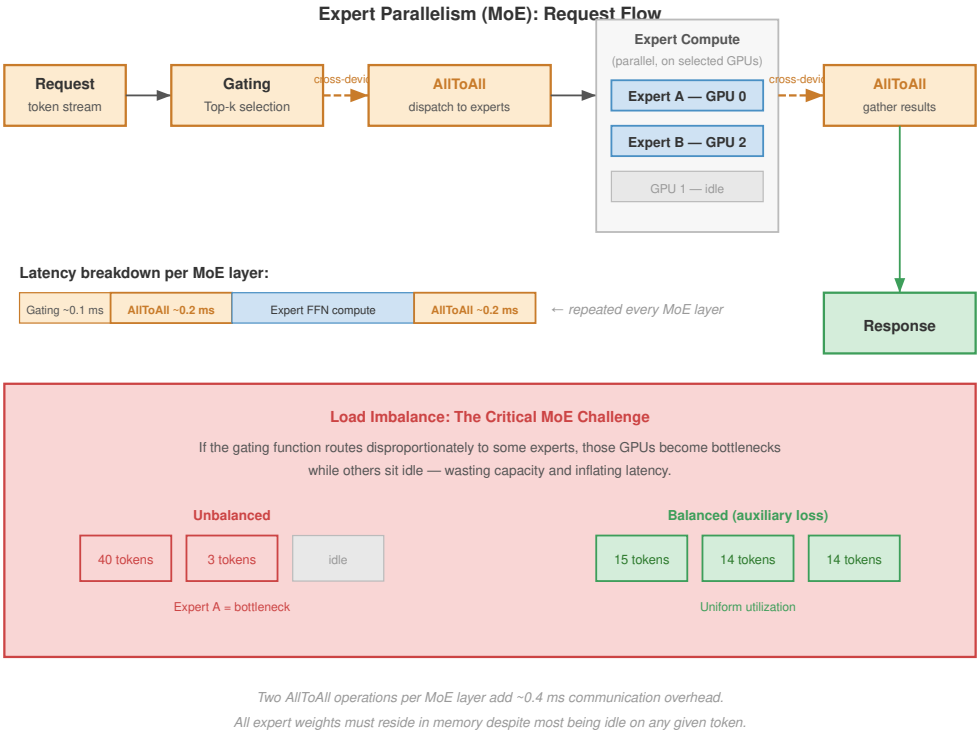


Figure 10.18: Expert Parallelism (MoE) Request Flow: Gating selects the top-k experts; an AllToAll dispatch sends tokens to the GPUs hosting those experts; experts compute in parallel; a second AllToAll gathers results. The lower panel contrasts unbalanced routing (one expert bottlenecks) against balanced routing (uniform utilization via auxiliary loss).

```
def readiness_check():
    free_memory = (
        torch.cuda.memory_reserved() - torch.cuda.memory_allocated()
    )
    if free_memory < MIN_REQUEST_MEMORY:
        return {
            "status": "not_ready",
            "reason": "insufficient GPU memory",
        }
    return {"status": "ready"}
```

Model warm-up: First inference after load is slower. Mark ready only after warm-up.

```
async def startup():
    model = load_model()
    # Warm up with dummy requests
    for _ in range(10):
        model.generate(dummy_input)
    # Now mark as ready
    global ready
    ready = True
```

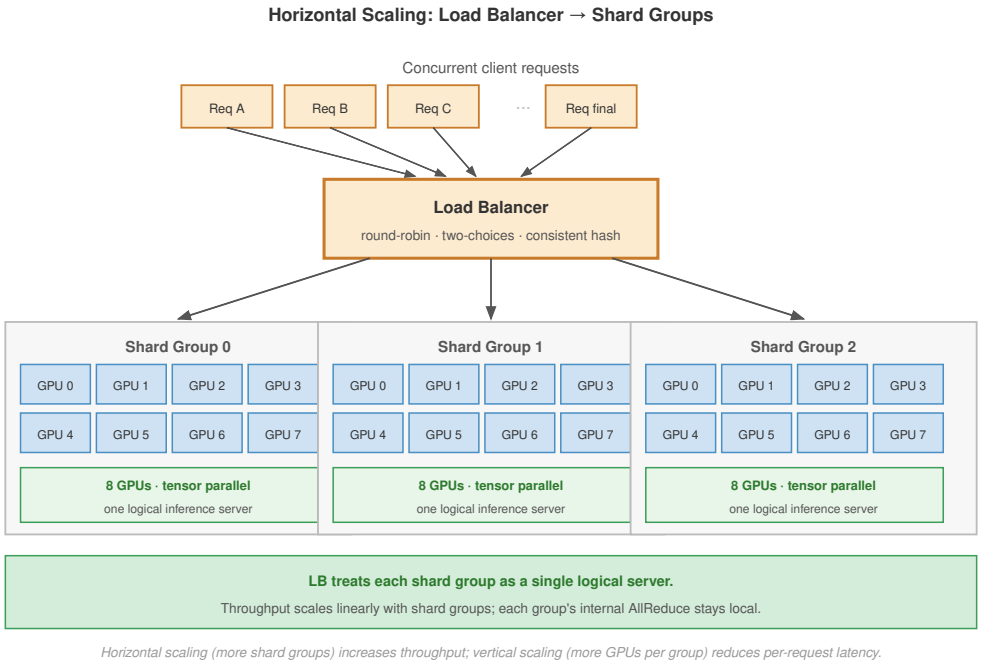


Figure 10.19: Horizontal Scaling with Shard Groups: The load balancer treats each shard group as a single logical inference server. Within each group, 8 GPUs coordinate via tensor parallelism. Adding groups scales throughput linearly; adding GPUs per group reduces per-request latency.

The three probe tiers each balance fast outage detection against GPU-resident probing cost; table 10.31 fixes interval, timeout, and failure-threshold values for liveness, readiness, and deep-health probes. The cadences form a gradient: readiness probes run fastest because routing decisions depend on them, liveness probes run slower because they only catch a dead process, and deep health checks run rarest because they consume GPU resources to validate model output.

Table 10.31: GPU Inference Health-Check Cadences: Liveness, readiness, and deep-health probe intervals, timeouts, and failure thresholds for GPU inference servers, balancing fast outage detection against the cost of GPU-resident probes.

Check Type	Interval	Timeout	Failure Threshold
Liveness	10s	5s	3 failures
Readiness	5s	3s	2 failures
Deep health	30s	10s	1 failure

Systems lesson: A serving process being alive is not the same as a model replica being ready. GPU memory, warm-up state, and model-output validity must be part of the health contract.

10.6.9 Quantitative analysis: Load balancing impact

The choice of load balancing algorithm has quantitative impact on system performance. Consider a system with 100 servers, 10,000 QPS, and variable request sizes (CV = 0.5). Table 10.32 quantifies the latency and overhead trade-offs:

Table 10.32: Load Balancing Algorithm Comparison: More sophisticated algorithms reduce queue lengths and latency at the cost of increased overhead.

Algorithm	Max Queue	P99 Latency	CPU Overhead
Random	4.2 requests	45 ms	Minimal
Round-robin	2.8 requests	32 ms	Minimal
Two-choices	1.9 requests	24 ms	2 probes/request
Least-connections	1.4 requests	19 ms	Global state
Two-choices + LC	1.2 requests	17 ms	2 probes + state

Table 10.32 shows the familiar serving trade-off: random and round-robin routing keep CPU overhead near zero but leave longer queues and higher latency variance; two-choices roughly halves p99 latency with only two probes per request; least-connections improves further when request cost varies but requires global state; and combining two-choices with least-connections produces the shortest queues at the highest implementation complexity.

For many serving systems, two-choices provides a strong trade-off between performance improvement and implementation complexity. Least-connections adds value for workloads with high request size variance (LLM serving, recommendation ranking).

10.6.10 Circuit breakers and backpressure

When a GPU inference server slows down, routing more requests to it can turn a local slowdown into a fleet-wide failure. Thermal throttling is a typical trigger: a replica that normally serves a request in 50 ms begins taking 500 ms, queues fill behind it, client timeouts generate retries, and those retries push load onto neighboring replicas. The load balancer needs a way to stop treating that replica as merely slow and start treating it as unavailable.

The **circuit breaker pattern**[15] (Nygard 2007) provides that control. In the closed state, the server receives normal traffic. When error rate, timeout rate, or latency exceeds a threshold, the breaker opens and requests fail fast or route elsewhere instead of waiting behind a saturated queue. After a recovery interval, the breaker becomes half-open: it admits a small number of probe requests, closes if they succeed, and reopens if they fail. The important systems property is bounded blast radius. The failed replica loses traffic quickly enough that its queue cannot export overload to the rest of the fleet.

[15] **Circuit Breaker**: Named after the electrical safety device that cuts current before wires overheat. In inference serving, the three-state machine (closed, open, half-open) prevents a single overloaded GPU replica from cascading failure to the entire fleet: once error rate exceeds threshold, the breaker opens and fails requests fast (microseconds) rather than waiting for timeout (seconds), protecting upstream callers and preserving capacity for healthy replicas.

Backpressure propagation complements the breaker by making overload visible upstream. When queue depth crosses a threshold, the server returns a signal such as `503 Service Unavailable`; the load balancer marks the replica as degraded and routes fewer requests to it. If all replicas become degraded, the system shifts from routing control to admission control, rejecting some requests at the edge rather than accepting work it cannot serve within the latency SLO. Circuit breakers isolate unhealthy replicas; backpressure tells the rest of the system to slow down before retries amplify the failure.

A saturated LLM fleet under KV cache pressure has three relief valves that generic backpressure cannot provide:

- **Context eviction**: Evict older conversational turns from the KV cache to reduce context window and free memory.
- **Speculative decoding**: Disable speculative decoding to reclaim the GPU memory its draft model occupies.
- **Fallback routing**: Route new requests to a smaller quantized fallback model until the primary queue drains.

Each response trades output quality or latency for continued availability, which is often preferable to rejection from the user's perspective. The system designer must specify which degradation modes are acceptable at each pressure tier, because the correct trade-off depends on the application—a real-time assistant tolerates context truncation better than silent quality degradation.

Example 10.17: Cascading failure prevention

Consider a scenario where one server becomes slow (thermal throttling):

Without circuit breaker:

1. Server A slows down (processing 500 ms instead of 50 ms)
2. Load balancer continues routing to Server A
3. Requests queue on Server A, timeouts begin
4. Retry logic sends failed requests to other servers
5. Other servers overload from retry traffic
6. System-wide failure

With circuit breaker:

1. Server A slows down
2. Error rate on Server A rises above 50 percent
3. Circuit breaker opens for Server A
4. All requests route to Servers B, C, D
5. System operates at reduced capacity but remains stable
6. After 30 s, the breaker admits probe requests and closes if they succeed

Systems lesson: For GPU inference, the canonical thresholds and recovery settings appear in table 10.33. Circuit breakers preserve partial capacity by stopping retries from amplifying one slow replica into a system-wide overload.

Table 10.33: Circuit-Breaker Settings for GPU Inference: Error and latency thresholds, open duration, and half-open probe count chosen so that a single throttling GPU is isolated quickly without prematurely failing healthy nodes.

Parameter	Value	Rationale
Error threshold	50%	GPU OOM failures are serious
Latency threshold	2× baseline	Detect throttling early
Open duration	30 s	Probe only after the queue drains
Half-open requests	5 requests	Careful testing before full reopening

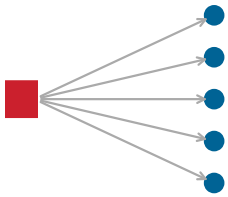
Efficient routing assumes that our inference servers own the underlying hardware. In enterprise environments, however, our critical production models are often running on the exact same physical clusters as experimental models and developer endpoints. To prevent a rogue developer query from crashing our flagship service, we must enforce strict multi-tenancy and resource isolation.

10.7 Multi-Tenancy and Isolation

Consider a critical customer-support chatbot running on the same GPU server as an experimental computer vision script. If the experimental workload accidentally monopolizes the PCIe bus or memory bandwidth, the customer-facing chatbot begins timing out, violating SLAs. Multi-tenancy and isolation ensure that co-located workloads save money without sacrificing the predictability of flagship services.

10.7.1 The multi-tenancy challenge

Multi-tenancy saves cost through cost efficiency, operational simplicity, and statistical multiplexing. Sharing infrastructure raises resource utilization, reduces the number of clusters to manage, and makes aggregate traffic more predictable than any one tenant's traffic.



One tenant's burst can degrade every workload sharing the pool.

The same sharing creates the isolation problem and multi-tenancy trade-offs that table 10.34 frames: noisy neighbors can make one tenant's burst degrade others, resource contention spans GPU memory, network bandwidth, and CPU cycles, security boundaries must protect tenant data, and SLO complexity increases when tenants have different requirements.

Table 10.34: Single vs. Multi-Tenant Tradeoffs: Multi-tenancy reduces cost but requires careful isolation engineering.

Aspect	Single-Tenant	Multi-Tenant
Resource utilization	30–50%	70-90%
Cost per request	Higher	Lower
SLO guarantees	Simple	Complex
Isolation	Complete	Requires engineering
Operational overhead	Higher (many clusters)	Lower (fewer clusters)

10.7.2 Noisy neighbor problems

The noisy neighbor problem occurs when one tenant's workload degrades performance for others sharing the same infrastructure. The interference manifests across three resource dimensions simultaneously.

GPU memory contention is the most severe: a tenant with unexpectedly long sequences can consume a disproportionate share of the KV cache pool. Consider three tenants sharing a 60 GB KV cache pool with equal 20 GB allocations. When one tenant begins issuing long-context requests, its allocation can swell to 45 GB, forcing evictions that reduce the other tenants from 200 concurrent sequences each to 75 – a 62 percent batch size reduction that directly degrades their throughput. Network bandwidth saturation compounds this effect when a tenant streaming many large responses consumes the available egress capacity. GPU time-sharing between tenants introduces context-switching overhead and unpredictable latency variance. Measuring noisy-neighbor impact requires capturing all three interference dimensions simultaneously.

Example 10.18: Quantifying noisy neighbor impact

Problem: How does one tenant's burst change latency for every tenant in an unprotected shared pool?

Consider an inference platform serving 10 tenants on shared H100 GPUs:

Baseline (even load):

- Each tenant: 100 QPS, 10 ms P99 latency
- GPU utilization: 70 percent
- All SLOs met

Tenant 3 bursts to 500 QPS (5× baseline). Without per-tenant quotas, the burst cascades into SLO violations for all ten tenants on the shared pool (table 10.35):

Table 10.35: Noisy neighbor without isolation (unprotected pool): All tenants violate their latency SLO when one tenant bursts.

Tenant	QPS	P99 Latency	SLO Status
Tenant 1	100	25 ms	Violated
Tenant 2	100	28 ms	Violated
Tenant 3	500	45 ms	Violated
Tenants 4-10	100 each	22-30 ms	Violated

Systems insight: Shared capacity raises utilization only if the serving system also enforces isolation. Without quotas, the effective SLO becomes the worst burst any tenant can generate.

Example 10.19: Noisy neighbor with per-tenant quotas

Enabling per-tenant quotas contains the impact to the offending tenant alone (table 10.36):

Table 10.36: Noisy neighbor with per-tenant quotas: Per-tenant QPS, p99 latency, and SLO status after enabling per-tenant resource quotas in the same workload.

Tenant	QPS (actual)	P99 Latency	SLO Status
Tenant 1	100	10 ms	Met
Tenant 2	100	10 ms	Met
Tenant 3	120 QPS (throttled)	50 ms	Violated (only for them)
Tenants 4-10	100 each	10 ms	Met

Systems insight: Quotas turn a platform-wide failure into a local admission-control decision: the bursting tenant is throttled, while protected tenants keep their latency contract.

10.7.3 Resource quotas and fair sharing

Resource quotas limit what each tenant can consume, preventing any single tenant from monopolizing shared resources. Hard quotas turn shared serving into admission control: the system checks concurrency, KV-cache memory, and request rate before admitting each request. Listing 10.1 uses those three gates so a tenant can be throttled at the scarce resource boundary instead of degrading the whole batch.

Listing 10.1: Resource Quotas: Admission control that enforces per-tenant limits on concurrency, KV cache memory, and request rate.

```
class TenantQuota:
    max_concurrent_requests: int  # e.g., 100
    max_kv_cache_mb: int  # e.g., 20,000
    max_qps: int  # e.g., 1,000
    max_batch_tokens: int  # e.g., 50,000


def admit_request(tenant_id, request):
    quota = get_quota(tenant_id)
    usage = get_usage(tenant_id)

    if usage.concurrent >= quota.max_concurrent:
        return RateLimitError("concurrent request limit")
    if usage.kv_cache_mb >= quota.max_kv_cache_mb:
        return RateLimitError("memory limit")
    if usage.qps >= quota.max_qps:
        return RateLimitError("rate limit")

    return admit(request)
```

The invariant is deny before interference. A request that would exceed any tenant quota never enters the scheduler, which keeps protected tenants from paying for another tenant's burst with higher queueing delay or KV-cache pressure.

Soft quotas with fair sharing provide a more flexible alternative. Each tenant has a nominal token generation quota (for example, 10,000 tokens/sec), but when the cluster is underutilized (50 percent capacity), tenants can burst to 2× their quota. When the cluster saturates (90 percent capacity), quotas are enforced. This approach maximizes utilization during low-traffic periods while protecting tenants during contention.

When total demand exceeds capacity, max-min fairness allocates resources to maximize the minimum allocation across tenants. The algorithm first gives each tenant an equal share, then redistributes unused capacity from tenants whose demand falls below their equal share. Raw request counts are an inadequate unit for this calculation because, as section 10.7 establishes, equal request counts are not equal load—a tenant issuing long-context summarization requests consumes far more KV cache and decode compute than a tenant issuing short classification queries. Token generation rate (tokens/sec) serves as the appropriate bounding metric, capturing both throughput and memory pressure. For three tenants with demands of 30,000, 20,000, and 80,000 tokens/sec competing for a GPU pool capable of 100,000 tokens/sec, max-min fairness yields allocations of 30,000, 20,000, and 50,000—each tenant receives up to its demand or its fair share, whichever is less, with excess capacity redistributed proportionally.

10.7.4 Priority scheduling

When tenants have different SLO requirements, priority scheduling ensures that high-priority requests receive resources first. Table 10.37 shows that preemption rights and resource guarantees move together down the three classes: critical traffic both preempts any lower class and reserves capacity, while best-effort never preempts and gets no guarantee, so an SLO tier maps directly onto a position in the preemption ladder.

Table 10.37: Priority Scheduling Classes: Three-tier priority hierarchy for multi-tenant inference. Critical traffic gets 100 percent reserved capacity and can preempt any lower class; standard traffic shares capacity proportionally and can preempt best-effort; best-effort runs on leftover capacity with no guarantees.

Class	Use Case	Preemption	Resource Guarantee
Critical	Revenue-generating	Can preempt lower	100% reserved
Standard	General traffic	Can preempt best-effort	Weighted share
Best-effort	Background, batch	Cannot preempt	No guarantee

Class	Use Case	Preemption	Resource Guarantee

The scheduler sorts incoming requests by priority class, then by arrival time within each class. When a new critical request arrives, it jumps ahead of all standard and best-effort requests in the queue, ensuring that revenue-generating traffic never waits behind background batch jobs.

Preemption extends this principle to requests already in flight. When a critical request arrives and all GPU slots are occupied by lower-priority work, the scheduler selects a victim from the lowest priority class, saves its KV cache state to CPU memory, and yields the GPU slot. Once the critical request completes, the preempted request resumes from its saved state rather than restarting from scratch. This checkpoint-and-resume mechanism makes preemption practical for autoregressive generation, where discarding partial output would waste all tokens generated so far.

10.7.5 Bulkhead pattern

The bulkhead pattern[16] (Nygard 2007) physically isolates tenant workloads, preventing failures from propagating across tenants. The pattern is named after ship compartments that contain flooding to isolated sections.

The strongest form of bulkhead dedicates entire replicas to specific tenants or tenant groups. Figure 10.20 illustrates this deployment-level isolation between gold and standard tiers:

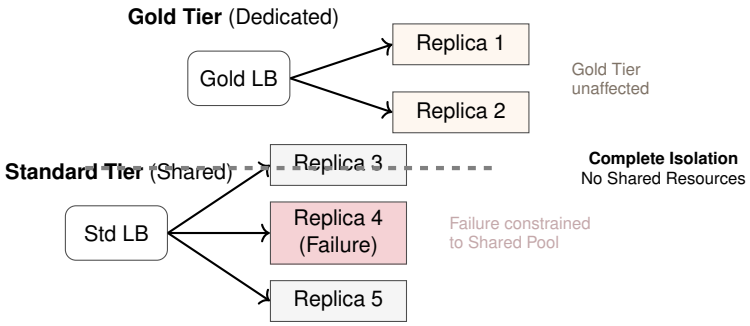


Figure 10.20: Bulkhead Isolation Patterns: To prevent cascading failures in multi-tenant systems, bulkheads isolate resources. Deployment-level bulkheads (shown) assign dedicated physical replicas to high-priority tenants, ensuring complete isolation. Request-level bulkheads enforce strict concurrency limits within shared processes. Like ship compartments, these boundaries ensure that a failure or resource exhaustion in one segment cannot sink the entire platform.

Deployment-level bulkheads provide complete isolation for premium tenants, ensuring that their performance is never affected by other workloads. The trade-off is lower overall resource utilization and increased operational overhead from managing dedicated infrastructure.

Request-level bulkheads complement this physical separation by limiting what any single request can consume within a shared process. Capping input length (for example, 8,000 tokens), output length (2,000 tokens), and execution time (30 seconds) prevents a single pathological request from monopolizing a GPU for minutes while other requests queue behind it.

Failure isolation follows naturally from bulkhead boundaries. When a tenant submits malformed input that triggers a model error, the bulkhead confines the failure to that tenant's request; other tenants continue processing normally rather than sharing a crashed inference worker. In multi-tenant deployments, these boundaries typically align with service tiers, each with distinct resource guarantees.

Example 10.20: Bulkhead configuration for API tiers

Scenario: A typical LLM API has enterprise, professional, and free service tiers.

Setup: The enterprise tier runs on a dedicated GPU pool with hardware isolation and a 99.9 percent availability SLO – no other tenant's traffic can affect it. The professional tier shares

16 **Bulkhead Pattern**: Named after ship compartments that contain flooding to isolated sections. The Titanic's bulkheads failed because they did not extend to the top of the hull, allowing water to cascade over as the ship tilted. The lesson for multi-tenant inference: shared GPU pools with soft quotas provide only partial isolation, and a single tenant's burst can exhaust memory or saturate bandwidth for all others. Effective bulkheads require dedicated hardware resources per critical tenant.

a GPU pool but receives 70 percent of that pool's capacity through resource quotas and can preempt the free tier when contention arises. The free tier receives the remaining 30 percent on a best-effort basis with rate limiting (for example, 10 QPS) and no SLO guarantee.

Systems lesson: Bulkheads concentrate isolation cost where the business requirement justifies it while still allowing shared infrastructure to run at high utilization.

10.7.6 Model isolation

When multiple models run on shared infrastructure, isolation must span memory, compute, and model loading. Memory isolation partitions GPU memory so that one model's allocation cannot encroach on another's. On an 80 GB H100, for example, reserving 40 GB for one model and 30 GB for a second leaves a 10 GB shared pool for transient needs – each model's KV cache growth is bounded regardless of the other's request pattern.

Compute isolation addresses the latency variance that GPU time-sharing introduces. MIG (Multi-Instance GPU)[17] provides the strongest guarantee by spatially partitioning the GPU into hardware-isolated instances, each with dedicated SMs, memory bandwidth, and L2 cache. Time-slicing offers a softer alternative with higher overhead, while dedicating entire GPUs per model trades utilization for complete isolation.

Model loading presents a subtler isolation challenge: loading a new model should not evict a running model from GPU memory. Listing 10.2 implements that policy as a fail-closed memory check: evict only lower-priority, evictable models, and reject the load when the remaining memory is protected.

17 **MIG (Multi-Instance GPU)**: Introduced with NVIDIA's A100 (NVIDIA Corporation 2020), MIG spatially partitions a single GPU into up to 7 isolated instances, each with dedicated compute SMs, memory bandwidth, and L2 cache. Unlike time-slicing, which shares resources and introduces latency variance of 2–5×, MIG provides hardware-enforced isolation with predictable performance, making it a strong mechanism for serving multiple small models on a single expensive data center GPU when partition sizes match model requirements.

Listing 10.2: Model Loading Isolation: Priority-aware memory management that prevents model eviction from violating tenant isolation constraints.

```
class ModelManager:
    def load_model(self, model_id, priority):
        required_memory = get_model_size(model_id)
        available = get_free_gpu_memory()

        if required_memory > available:
            # Check if eviction would violate isolation
            evictable = get_evictable_memory(priority)
            if required_memory > evictable:
                raise InsufficientMemoryError(
                    "Cannot load without violating isolation constraints"
                )
            evict_lower_priority_models(priority, required_memory)

        load_to_gpu(model_id)
```

That rejection path is the isolation guarantee. It may lower utilization in the moment, but it prevents a control-plane load request from turning into an unplanned model eviction or a tenant-visible outage.

10.7.7 Observability for multi-tenancy

Effective multi-tenancy requires per-tenant visibility into resource consumption and performance. The essential metrics span four dimensions: request volume and latency distribution (P50, P95, P99), GPU memory and KV cache utilization, throttling and preemption event counts, and error rates broken down by error type. Together, these metrics reveal whether each tenant is receiving its contracted quality of service.

Three alerting thresholds transform these metrics into actionable signals:

- **SLO violation**: An alert fires when a tenant's P99 latency exceeds its target by more than 10 percent for five consecutive minutes.
- **Quota exhaustion**: An alert triggers when usage reaches 90 percent of the tenant's allocation, providing advance warning before hard limits cause request rejection.

- **Noisy neighbor detection**: An alert identifies tenants consuming more than twice their fair share for five minutes, flagging interference before it cascades to other tenants.

Chargeback and attribution close the operational loop by mapping resource consumption – GPU-seconds, KV cache GB-hours, network egress – to individual tenants for billing and capacity planning. Without per-tenant attribution, organizations cannot distinguish between tenants that need more capacity and tenants that are simply inefficient, making informed provisioning decisions impossible.

Proper isolation allows us to securely pack workloads onto a fixed set of servers. Traffic on the internet, however, is rarely fixed. To handle viral product launches or deep nocturnal lulls without burning cash, our isolated inference services must dynamically grow and shrink across global data centers through intelligent autoscaling.

10.8 Autoscaling and Global Infrastructure

A viral social media post drives a 10× spike in traffic to a generative AI application in under five minutes. If the infrastructure team relies on manual scaling or slow, CPU-based metrics to spin up new GPU instances, the service collapses before new replicas even finish downloading the model weights. Autoscaling for ML inference requires predictive, custom-metric strategies to handle these massive load dynamics.

That response has four coupled dimensions: how capacity is added, how long new replicas take to become useful, how predictive signals avoid cold-start lag, and how global routing moves traffic when capacity or regions fail. Treating autoscaling as a replica-count knob misses the interaction between model loading, cache warmup, spot capacity, and user geography.

10.8.1 Scaling dimensions

Inference systems can scale along three dimensions, each with distinct cost, latency, and speed characteristics. Table 10.38 contrasts these approaches.

A common approach is horizontal scaling: adding or removing model replicas to adjust aggregate throughput. Total capacity scales linearly as Capacity = Replicas × Per-replica throughput, but each new replica requires provisioning time measured in minutes.

When replica count is fixed, vertical scaling replaces existing GPUs with more powerful ones to increase per-replica throughput. Cost efficiency is Throughput/GPU cost, but vertical scaling requires redeployment and is the slowest scaling response.

The fastest response comes from batch size scaling, which adjusts how many requests the system processes simultaneously. Increasing batch size trades higher per-request latency for greater throughput, with no provisioning delay. This makes batch size the first knob to turn during a traffic spike, buying time for horizontal scaling to take effect. These scaling dimension trade-offs are summarized in table 10.38.

Table 10.38: Scaling Dimension Tradeoffs: Horizontal scaling preserves per-request latency but waits on provisioning, batch-size scaling reacts immediately by trading latency for throughput, and vertical scaling changes the hardware envelope but requires redeployment.

Scaling Type	Latency	Cost	Speed
Horizontal (add replicas)	Unchanged	Linear	Slow (minutes)
Batch size	Increases	Unchanged	Instant
Vertical (better GPU)	Unchanged	Nonlinear	Very slow (redeployment)

10.8.2 The cold start problem

Cold start latency represents a significant challenge for serverless inference because model loading often dominates the total startup time. Unlike stateless web services that start in seconds, inference services must provision hardware, load weights, and warm runtime kernels before serving traffic, as equation 10.14 defines:

$$T_{\text{cold start}} = T_{\text{provision}} + T_{\text{load}} + T_{\text{warmup}} \tag{10.14}$$

The provisioning phase ($T_{\text{provision}}$) acquires a GPU instance, which takes 30 seconds to several minutes depending on cloud provider and GPU type.[18] This delay alone exceeds the total cold start time of a stateless web service.

Once the instance is available, model loading (T_{load}) transfers weights from storage to GPU memory. Loading time scales with model size and depends heavily on storage tier, as table 10.39 shows:

[18] **Cold Start (GPU vs. Serverless)**: Serverless functions cold-start in 100 ms–1 s; GPU inference cold-starts in 1–10 minutes due to GPU allocation, model weight transfer, and CUDA context initialization. This 100× gap means reactive autoscaling alone cannot absorb traffic spikes, making predictive scaling and warm pools essential design requirements for any GPU serving fleet with latency SLOs.

Table 10.39: Model Load Time by Storage Tier: Local SSD loads weights at roughly 6 GB/s; remote object storage (S3) loads at roughly 0.5 GB/s before additional validation and runtime overhead. The 12× gap matters most for the largest models: a 175B model takes about 1 minute from SSD (350 GB at 6 GB/s) vs. 12 minutes from S3, dominating the cold-start budget for any serving deployment that has to pull weights from object storage on each scale-up.

Model Size	Load Time (SSD)	Load Time (S3)
7B (14 GB)	5s	30s
70B (140 GB)	25s	5min
175B (350 GB)	1min	12min

Even after weights reside in GPU memory, the first inference runs slower than steady state because just-in-time (JIT) kernel compilation, CUDA context initialization, memory pool allocation, and cache population must complete. This warmup phase (T_{warmup}) typically requires 10–30 dummy inferences, adding 5–30 seconds. The phase-by-phase timeline for Llama-70B shows how these delays accumulate in practice.

Example 10.21: Cold start timeline for Llama-70B

Table 10.40 traces the per-phase and cumulative duration of bringing up a new replica for Llama-70B on H100:

Table 10.40: Cold-start timeline for a Llama-70B replica (H100): Per-phase and cumulative duration for bringing up a fresh Llama-70B serving replica on an H100 instance.

Phase	Duration	Cumulative
Cloud API request	5s	5s
GPU instance provisioning	60s	65s
Container startup	10s	75s
Model download (S3)	300s	375s
Model load to GPU	25s	400s
CUDA warmup	15s	415s
Readiness probe pass	5s	420s
Total cold start	**7 min**	

Systems insight: Scaling decisions must anticipate demand several minutes in advance. Reactive scaling alone cannot handle sudden traffic spikes.

Figure 10.21 visualizes the cumulative timeline:

10.8.3 Reactive scaling

Reactive scaling adjusts capacity after the system observes pressure. It is useful for gradual demand changes, but it cannot solve the cold-start problem by itself because the signal arrives after traffic has already reached the fleet.

Metric-based scaling is a thresholded control loop. Listing 10.3 separates the target operating point from the scale-up and scale-down thresholds, while the cooldown period prevents the controller from undoing its own last decision before the new replicas affect the metric.

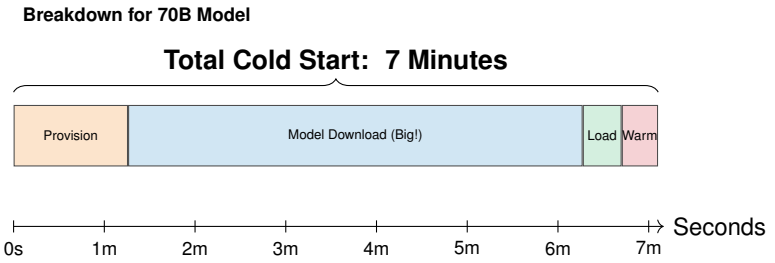


Figure 10.21: Anatomy of a Cold Start: Bringing up a new GPU inference replica is a multi-step process taking minutes. While container startup is fast, provisioning the specialized instance and downloading massive model weights (100 GB+) dominate the timeline. CUDA context initialization and "warmup" inference passes add further delay. This roughly 7-minute lag makes purely reactive scaling dangerous for handling sudden traffic spikes.

Listing 10.3: Metric-Based Scaling: Autoscaling policy driven by utilization thresholds with a cooldown period to prevent oscillation.

```
autoscaling:
  metric: cpu_utilization  # or gpu_utilization, queue_depth
  target_value: 70%
  scale_up_threshold: 80%
  scale_down_threshold: 50%
  cooldown_period: 300s
```

Those fields encode the responsiveness-stability trade-off. Tighter thresholds react sooner but can oscillate under noisy utilization; longer cooldowns dampen oscillation but leave the fleet overloaded longer after a real spike.

A more responsive alternative scales on queue depth rather than utilization, targeting a maximum queue length per replica:

$$\text{Desired replicas} = \left\lceil \frac{\text{Queue depth}}{\text{Queue target}} \times \text{Current replicas} \right\rceil$$

When the primary concern is SLO compliance rather than utilization, latency-based scaling adjusts replicas to maintain a target P99 latency:

$$\text{Desired replicas} = \left\lceil \frac{\text{P99}_{\text{observed}}}{\text{P99}_{\text{target}}} \times \text{Current replicas} \right\rceil$$

All reactive approaches share three fundamental limitations:

- **Cold start vulnerability**: Cold start time prevents rapid response to sudden spikes.
- **Oscillation risk**: Metric-based triggers can create oscillation between scale-up and scale-down cycles.
- **Over-provisioning requirement**: The system must over-provision during the cold start window to absorb load that new replicas cannot yet handle.

These limits show up most sharply during a traffic spike, where replicas must exist before cold-started capacity becomes usable.

Napkin Math 10.7: Reactive scaling response analysis

Consider traffic spike from 1000 to 3000 QPS:
Current state: 10 replicas, 100 QPS each, 70 percent utilization
Target state: 30 replicas for 3000 QPS
Without prewarming:

- T=0: Spike detected, scale-up triggered
- T=0 to T=5min: Cold start for 20 new replicas
- T=0 to T=5min: Existing 10 replicas handle 3000 QPS (300 QPS each)
- Utilization: 210 percent (overloaded)
- P99 latency: 500 ms+ (SLO violated)

With sufficient warm spares (20 replicas):

- T=0: Spike detected, warm spares activated immediately
- T=0: 30 replicas handle 3000 QPS (100 QPS each)
- T=0 to T=5min: Replenish 20 warm spares in the background
- Utilization: 70 percent (headroom preserved)
- P99 latency: 80 ms (SLO maintained)

Warm spares convert cold-start delay into idle-capacity cost: the system pays for extra replicas before the spike so the user does not pay for provisioning latency during the spike.

At internet-launch scale, the same cold-start window turns from a capacity-planning nuisance into an infrastructure survival problem.

Example 10.22: The ChatGPT traffic spike

Context: When ChatGPT scaled from zero to 100 million users in two months, the engineering challenge shifted from model quality to survival.

Failure mode: The service faced an unprecedented cold-start problem: provisioning thousands of GPUs while managing a KV cache memory footprint that grew linearly with context length and request concurrency. Early serving infrastructure had to evolve rapidly toward optimized serving layers, aggressive batching and quantization, and geographic load balancing to keep per-token latency acceptable despite the deluge.

Systems lesson: Successful model launch can create an infrastructure failure mode. At serving scale, demand growth, state memory, and provisioning latency become part of the product architecture.

10.8.4 Predictive scaling

Predictive scaling anticipates demand before it occurs, initiating scaling ahead of traffic changes. The simplest and most effective approach uses time-series forecasting on historical traffic patterns. Listing 10.4 shows a simplified forecasting function combining daily and weekly seasonality with trend estimation. For generative LLM services, however, forecasting request volume alone is insufficient: a volume spike of long-context summarization requests exhausts KV cache far faster than the same volume of short-context chat completions, because each long-context request can hold tens of thousands of tokens in active KV cache while decode completes. Accurate predictive scaling for LLM fleets must therefore forecast the *mix* of request types alongside total volume, treating a shift toward long-context requests as a memory-bound demand increase even when request count is unchanged.

Listing 10.4: Predictive Scaling: Time-series demand forecasting combining daily seasonality, weekly seasonality, and trend estimation.

```
def predict_demand(current_time, history):
    # Seasonal decomposition
    daily_pattern = extract_daily_seasonality(history)
    weekly_pattern = extract_weekly_seasonality(history)

    # Trend estimation
    trend = estimate_trend(history)

    # Forecast
    predicted = (
        daily_pattern[current_time.hour]
        * weekly_pattern[current_time.weekday()]
        * trend
    )
    return predicted
```

Event-driven scaling is useful when demand has an external clock the platform can trust. Listing 10.5 schedules capacity before a product launch or recurring batch window, using the ramp-up field to hide cold start and the duration field to release extra replicas after the event.

Listing 10.5: Event-Driven Scaling: Scheduled scaling rules that preprovision capacity for product launches and recurring traffic spikes.

```
scheduled_scaling:
  - event: "product_launch"
    time: "2024-03-15 09:00 UTC"
    target_replicas: 50  # 5x normal
    ramp_up: 30min  # Start scaling 30min before

  - event: "nightly_rag_embedding_refresh"
    cron: "0 2 * * *"  # Every night at 2am
    target_replicas: 20  # 2x normal; batch embedding workload exhausts KV-cache headroom
    duration: 2h
```

This pattern works for launches, batch refreshes, and calendar-driven traffic; it does not replace reactive control for unplanned demand because the schedule is only as good as the event model.

In practice, neither forecasting nor event schedules alone suffice. Production systems combine a predictive baseline with reactive adjustment to handle both anticipated patterns and unexpected deviations:

$$\text{Target replicas} = \max(\text{Predicted}, \text{Reactive}) + \text{Buffer}$$

Daily patterns make the proactive side of this rule concrete.

Example 10.23: Predictive scaling for traffic

Table 10.41 shows a chatbot service's predictable diurnal traffic:

Table 10.41: Daily traffic pattern for a chatbot service: Hourly QPS bands and the number of replicas required to absorb each band for a chatbot with predictable diurnal traffic.

Time (UTC)	Typical QPS	Replicas Needed
00:00-06:00	500	5
06:00-09:00	1500	15 (ramp up)
09:00-17:00	3000	30 (peak)
17:00-20:00	2000	20 (ramp down)
20:00-00:00	1000	10

A predictive autoscaler converts that demand curve into the schedule in table 10.42, where each scale-up action leads its ramp by roughly 30 minutes (10 replicas warming at 05:30 for the 06:00 ramp, 15 more at 08:30 for the 09:00 peak). That deliberate lead time is what absorbs the cold-start delay the reactive knob cannot anticipate.

Table 10.42: Predictive scaling schedule: Per-time-bucket scaling actions, active replicas, and replicas in cold-start warming for the same chatbot service.

Time	Action	Replicas Active	Replicas Starting
05:30	Scale up	5	+10 warming
06:00	Traffic ramp	15	-
08:30	Scale up	15	+15 warming
09:00	Peak traffic	30	-
17:00	Scale down	20	-10 terminating
20:00	Scale down	10	-10 terminating
00:00	Scale down	5	-5 terminating

Systems insight: A reactive-only autoscaler must over-provision during each ramp (about 45 replicas at peak) because it cannot anticipate the curve. Predictive scaling sizes the fleet to the actual peak (30 replicas), recovering roughly 33 percent of GPU spend without sacrificing latency headroom.

The cost savings above depend on anticipating traffic before it arrives. As figure 10.22 shows, proactive provisioning avoids the SLO violations that reactive-only systems suffer during ramp-up periods.

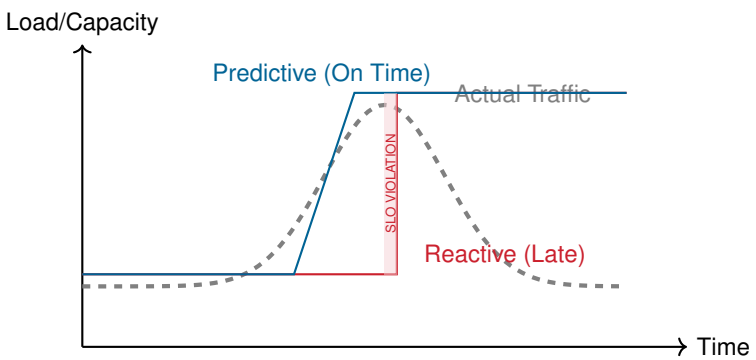


Figure 10.22: Predictive vs. Reactive Scaling: Reactive scaling (red line) responds to traffic spikes after they occur, leading to periods of under-provisioning where SLOs are violated due to cold start latency. Predictive scaling (blue line) anticipates traffic and begins provisioning capacity before the spike arrives, ensuring consistent performance.

10.8.5 Warm pool management

Maintaining a pool of prewarmed replicas reduces effective cold start time by buying idle readiness before demand arrives. Warm pool sizing starts from the expected spike and per-replica throughput:

$$\text{Warm pool size} = \frac{\text{Max expected spike}}{\text{Per-replica throughput}} \times \text{Headroom factor}$$

For example, if the maximum expected spike is 2× normal and the headroom factor is 1.5, the fleet needs three times the minimum pool capacity available before the spike arrives:

$$\text{Warm pool} = 2 \times 1.5 = 3 \times \text{minimum pool capacity}$$

The cost of maintaining warm replicas is Pool size × GPU cost/hour × Idle fraction, creating a direct trade-off between response speed and idle-capacity expense. Tiered warm pools, summarized in table 10.43, resolve this trade-off by maintaining replicas at different readiness levels, each with different activation latency and cost:

Table 10.43: Tiered Warm Pool Readiness Levels: Three readiness tiers trade activation latency against idle-capacity cost. Hot replicas absorb bursts instantly at full cost; warm replicas activate in 30 seconds at 60 percent cost; cold replicas cost nothing but take five minutes to come online. Production deployments combine a small hot pool, a larger warm pool, and unlimited cold capacity.

Tier	State	Response Time	Cost (relative)
Hot	GPU loaded, running	Instant	100%
Warm	GPU allocated, model loaded	30s	60%
Cold	GPU not allocated	5+ min	0%

When a traffic spike hits, the scaling sequence activates hot replicas immediately, promotes warm replicas within 30 seconds, and cold-starts new instances only if demand persists beyond the warm pool's capacity. A typical configuration maintains 2 hot replicas for instant burst absorption and 5 warm replicas for sustained spikes, with unlimited cold capacity available from the cloud provider.

10.8.6 Scaling response time analysis

The scaling response budget is the sum of the control loop's detection, decision, provisioning, and warmup delays. Equation 10.15 makes those phases explicit:

$$T_{\text{response}} = T_{\text{detect}} + T_{\text{decide}} + T_{\text{provision}} + T_{\text{warmup}} \tag{10.15}$$

Table 10.44 decomposes each term and identifies its primary optimization path:

Table 10.44: Scaling Response Time Components: The four phases of a scaling event with typical durations and their primary optimization lever. Provisioning dominates the budget by an order of magnitude, which is why warm pools are a high-leverage optimization for latency-sensitive serving fleets.

Component	Duration	Optimization
Detection	10–60s	Reduce metrics interval
Decision	1–5s	Faster autoscaler
Provisioning	30s–5min	Warm pools
Warmup	5–30s	Precompilation

Each component offers distinct optimization opportunities. Detection speed improves by collecting metrics at 1-second intervals rather than 60-second intervals, at the cost of noisier signals and higher metric volume. Decision speed improves by precomputing scaling plans for predicted scenarios so that when a trigger fires, the system executes a precomputed plan rather than computing one from scratch. Provisioning speed improves most dramatically through warm pools, which eliminate the provisioning phase entirely for anticipated demand. Warmup speed improves through precompiled inference engines that skip JIT compilation, reducing the final phase from 30 seconds to under 5 seconds.

10.8.7 Spot and preemptible instances

Cloud providers may offer discounted GPU instances[19] that can be reclaimed with short notice, so the scheduling decision is whether the workload can drain, reroute, or recompute without violating its SLO. Table 10.45 lays out the three GPU instance classes and the workloads each suits best:

[19] **Spot Instance Economics**: Cloud providers have sold unused GPU capacity at large discounts, sometimes with 30-second to 2-minute termination notice. For inference, termination destroys in-flight requests and KV cache state, requiring graceful drain logic and rapid request re-routing. The savings can justify this complexity for burst and best-effort traffic tiers but are unsuitable for SLO-critical paths where termination would violate latency guarantees.

Table 10.45: GPU Instance Types by Pricing and Reliability: Three GPU instance classes trade discount against interruption risk. On-demand is the reference price with no interruption; reserved instances trade upfront commitment for moderate discount; spot instances trade preemption risk for large savings. Production fleets often run a baseline of reserved + on-demand instances and burst into spot capacity for best-effort traffic.

Instance Type	Discount	Interruption Notice	Use Case
On-demand	0%	Never	SLO-critical
Reserved	30-60%	Never	Steady baseline
Spot/Preemptible	60–90%	30s–2min	Burst capacity

Spot termination handling is a race against the provider's warning window. Listing 10.6 orders the shutdown path so new work stops first, in-flight work drains while time remains, recoverable state is persisted, and the load balancer stops sending traffic before the instance disappears.

Listing 10.6: Spot Termination Handling: Graceful shutdown sequence that drains in-flight requests, persists KV cache state, and deregisters from the load balancer.

```
def handle_spot_termination():
    # Received 2-minute warning
    # 1. Stop accepting new requests
    stop_accepting_requests()

    # 2. Complete in-flight requests (if possible)
    await complete_inflight(timeout=90)

    # 3. Save state for resumption elsewhere
    save_kv_cache_to_storage()

    # 4. Signal load balancer to redirect traffic
    deregister_from_loadbalancer()

    # 5. Terminate gracefully
    shutdown()
```

The sequence matters because the steps are not interchangeable. Deregistering too early strands useful capacity, but saving state after shutdown loses the KV cache; the handler preserves availability by draining first and routing away only after the replica can no longer accept work safely.

A spot-aware architecture splits traffic between on-demand and spot replicas by SLO class, as shown in figure 10.23. SLO-critical requests always route to on-demand capacity, while best-effort requests absorb the cost savings and interruption risk of spot instances.

The split architecture in figure 10.23 achieves approximately 18–27 percent cost reduction compared to a fully on-demand fleet in the representative discount scenario, while absorbing preemption risk only for best-effort traffic and keeping SLO-critical requests on guaranteed capacity.

10.8.8 Global inference infrastructure

Global serving becomes necessary when distance, failure domains, or data-residency constraints exceed what one region can hide. A user in Tokyo expects low-latency responses regardless of where models were trained or where the company headquarters is located; the engineering choice is whether to replicate computation, cache repeated work, or centralize the model and pay the network penalty. The architectural patterns are useful only when tied to that choice.

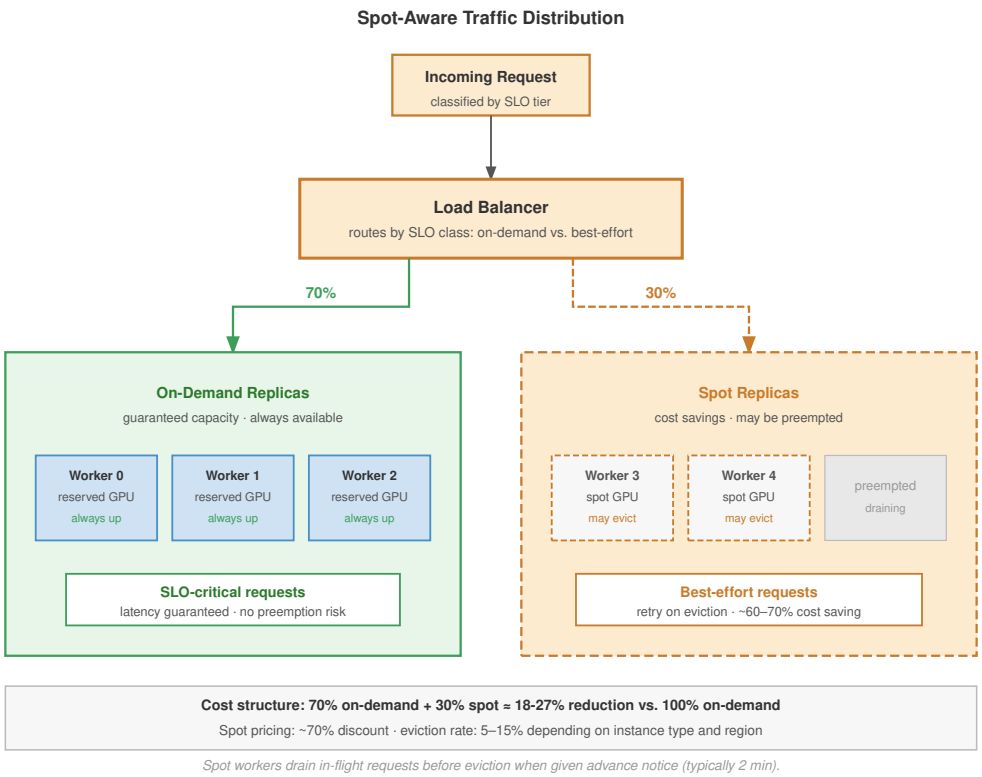


Figure 10.23: Spot-Aware Traffic Distribution: The load balancer routes 70 percent of traffic to on-demand replicas (guaranteed availability) and 30 percent to spot replicas (cost savings, preemption risk). SLO-critical requests always use on-demand; best-effort requests tolerate occasional retries on eviction. In the representative discount scenario, the mixed fleet achieves roughly 18–27 percent cost reduction vs. 100 percent on-demand.

10.8.8.1 Why multi-region matters

Single-region deployment creates three fundamental limitations. The most immediate is the latency floor imposed by network round-trip time (RTT) to distant users, quantified in table 10.46:

Table 10.46: Network RTT by User Location: A single US-East deployment imposes a 75 to 200 ms RTT on European, Asian, and Australian users—an order of magnitude above the local-region floor. For interactive workloads where each request makes several model calls, this latency floor compounds quickly, motivating multi-region deployment for any globally distributed user base.

User Location	RTT to US-East	RTT to Local Region
New York	10 ms	10 ms
London	75 ms	10 ms
Tokyo	150 ms	10 ms
Sydney	200 ms	10 ms

For interactive applications (chatbots, autocomplete), these delays compound across multiple model calls per request. Beyond latency, single-region deployment creates a single point of failure: cloud region outages, while rare, affect all users simultaneously. Regulatory constraints add a third dimension, as data residency requirements (the General Data Protection Regulation (GDPR) and data sovereignty laws) may require processing user data within specific geographic boundaries. The resulting patterns form a decision ladder: regional replicas pay the highest deployment cost but

minimize RTT and isolate regional failures; edge caching pays operational complexity only when queries repeat; and cross-region sharding is a last resort for capacity shortages because inter-region latency overwhelms per-stage compute.

10.8.8.2 Pattern 1: Global load balancing with regional replicas

Each region runs independent inference replicas with identical models. A global load balancer routes each user to the nearest region based on measured round-trip latency, as shown in figure 10.24. A shared model registry synchronizes weights across regions during rollouts.

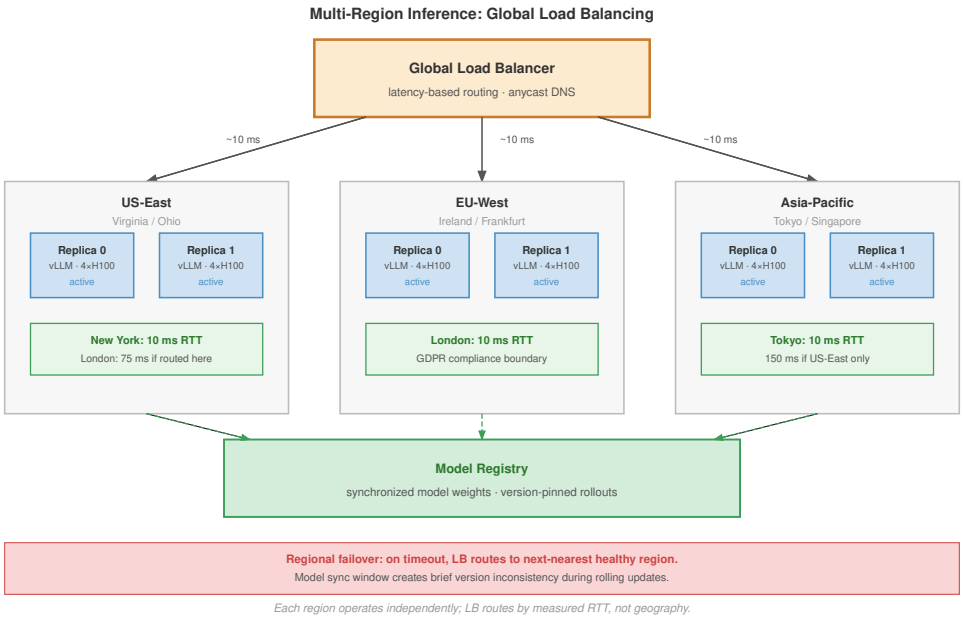


Figure 10.24: Multi-Region Inference: Global Load Balancing: A global load balancer routes requests to three independent regional deployments (US-East, EU-West, Asia-Pacific) based on latency. Each region operates independently; a shared model registry synchronizes weights. During regional failure, the LB reroutes to the next nearest healthy region.

As figure 10.24 shows, independent regional replicas reduce RTT from 150–200 ms to under 10 ms for distant users, while the shared model registry ensures consistency across deployments. Model synchronization is the first architectural concern: updates must propagate to all regions. Push-based synchronization lets a central registry push to every region, which is simple but creates a possible inconsistency window; pull-based synchronization lets regions poll for updates, trading higher rollout latency for stronger consistency; hybrid synchronization combines push notification with pull verification. Version consistency is the second concern. During rollouts, different regions may briefly serve different versions. For most applications this is acceptable, but applications requiring strict consistency need version pinning in request routing.

10.8.8.3 Pattern 2: Edge caching with central inference

For models too large to replicate globally, caching responses at the edge eliminates redundant inference for repeated queries. Figure 10.25 shows the two paths: a cache hit returns in under 10 ms from the nearest edge node; a cache miss traverses the full backend and populates the cache on return.

The two paths in figure 10.25 highlight the order-of-magnitude latency difference: cache hits return in under 10 ms, while cache misses incur the full backend round-trip. Effectiveness depends on request repeatability across workloads such as autocomplete, FAQ chatbots, open-ended chat, and code generation, as table 10.47 quantifies:

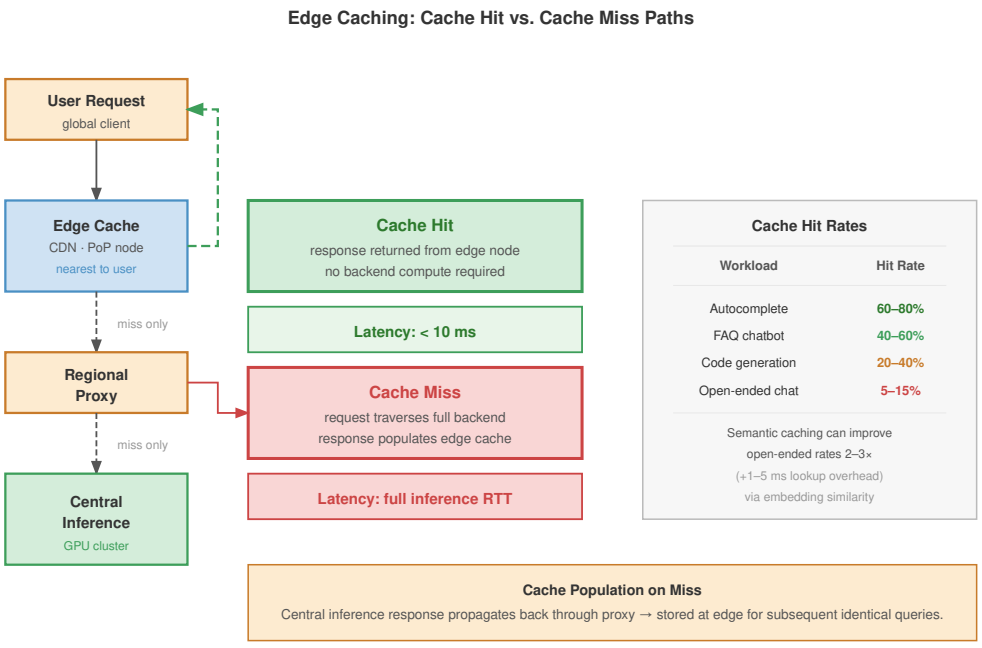


Figure 10.25: Edge Caching: Cache Hit vs. Cache Miss Paths: A cache hit returns immediately from the CDN edge node (<10 ms). A cache miss flows through to central inference and populates the cache on return. Hit rate determines the effective cost saving; open-ended generative workloads have low repeatability and benefit little from this pattern.

Table 10.47: Edge Cache Hit Rates by Workload: Cache effectiveness depends almost entirely on request repeatability. Autocomplete and FAQ workloads have high prefix overlap and benefit dramatically; open-ended chat has near-unique inputs and barely benefits. The hit-rate range determines whether edge caching is worth the operational complexity for a given workload.

Workload	Cache Hit Rate	Suitability
Autocomplete	60–80%	Excellent
FAQ chatbot	40-60%	Good
Open-ended chat	5-15%	Poor
Code generation	20–40%	Moderate

Semantic caching[20] (caching based on embedding similarity rather than exact match) can improve hit rates for open-ended workloads.

[20] **Semantic Caching**: Returns cached responses for queries that are embedding-similar rather than string-identical, requiring a vector database lookup per request. The trade-off: higher hit rates (potentially 2–3× for open-ended workloads) at the cost of added lookup latency (1–5 ms) and the risk of incorrect cache hits when semantically similar queries require different answers. Setting the similarity threshold is a precision-recall trade-off with direct impact on response correctness.

10.8.8.4 Pattern 3: Cross-region model sharding

Cross-region model sharding is a last-resort capacity strategy rather than a normal latency optimization. For the largest models, pipeline parallelism can theoretically span regions: a router assigns early layers to one region and later layers to another. This pattern is rarely practical because inter-region network latency (75–150 ms) dominates compute time per stage (1–5 ms), making cross-region pipeline parallelism 10–100× slower than co-located deployment. It applies only when no single region has sufficient GPU capacity for the full model.

Because cross-region sharding is so costly on the steady-state path, most production designs keep inference co-located and use other regions for resilience. The cross-region question therefore shifts from splitting one request across regions to deciding how quickly traffic can move when a region fails.

10.8.8.5 Cross-region failover

When a region becomes unavailable, traffic must reroute to healthy regions. Listing 10.7 shows active-active failover routing logic that falls back to the next healthy region after timeout or unavailability.

Listing 10.7: Active-Active Failover: Global routing that falls back to the second-nearest healthy region on timeout or unavailability.

region back to the previous model version while preserving the new version for debugging. The key invariant is that no region proceeds to a new version until the prior region has demonstrated healthy metrics for a sufficient monitoring window.

Global deployment health requires monitoring four metrics both per-region and globally, as table 10.48 specifies:

Table 10.48: Global Deployment Health Metrics: Four metrics monitored at both per-region and global granularity during phased rollouts. Per-region thresholds catch localized regressions; global thresholds catch cross-region effects (such as a P99 inflation greater than 2× the single-region target) that no per-region check would surface.

Metric	Per-Region	Global
Error rate	< 0.1%	< 0.1%
P99 latency	< target	< 2× single-region
Throughput	Stable	Stable
Model quality	Within bounds	Consistent across regions

10.8.8.7 Cost optimization across regions

GPU pricing varies by region. Table 10.49 shows representative H100 prices so placement can balance cost against latency requirements:

Table 10.49: H100 Pricing and Latency by Region: Representative spot and on-demand H100 prices alongside RTT to US users. The 20 percent price spread across regions is modest compared to the latency spread, so cost-aware routing typically sends only latency-tolerant workloads (batch inference, background processing) to the cheapest region while keeping interactive traffic local.

Region	H100 Spot Price	On-Demand	Latency to US Users
US-East	$2.50/hr	$4.00/hr	10–50 ms
US-West	$2.30/hr	$3.80/hr	30–70 ms
EU-West	$2.80/hr	$4.20/hr	75–100 ms

Cost-aware routing directs latency-tolerant workloads (batch inference, background processing) to the cheapest available region, as shown in listing 10.8.

Listing 10.8: Cost-Aware Routing: Priority-based routing that sends low-priority batch requests to the cheapest region while keeping interactive traffic local.

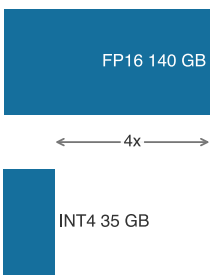
```
def route_batch_request(request):
    if request.priority == "low":
        # Route to cheapest region with capacity
        return get_cheapest_region_with_capacity()
    else:
        # Route to nearest region
        return get_nearest_region(request.user_location)
```

Time-of-day routing can reduce costs by 20–40 percent for batch workloads while maintaining SLOs for interactive traffic.

Scaling global infrastructure aggressively is highly effective, but it is also exceptionally expensive. To further bend the cost curve of our deployments, we must look inward again at the model itself, altering its fundamental mathematical representation to run faster and cheaper through weight quantization.

10.9 Weight Quantization for Serving

Weight quantization for serving is a bottleneck-selection decision: reducing weights from 16-bit floats to 8-bit or 4-bit integers can turn a 140 GB FP16 language model that requires two A100 GPUs



INT4 turns a two-GPU model into a one-GPU candidate.

into a 35 GB INT4 representation that fits on a single GPU. The trade-off is a small accuracy loss in exchange for a 4× reduction in memory bandwidth and capacity demand.

> **Systems Perspective 10.4: Quantization as a serving constraint**
>
> In serving systems, quantization is not merely a model-compression technique; it changes the binding resource. Post-training quantization (PTQ) can shrink a deployed model without retraining, while quantization-aware training (QAT) spends training effort to preserve accuracy at lower precision. The systems question is which constraint the lower-precision representation relaxes: GPU memory capacity, HBM bandwidth during decode, accelerator instruction support, or cost per served token. The precision foundations appear in section 9.4 and section 9.4.2.

Quantization reduces numerical precision of model weights and activations, decreasing memory footprint by 2–4× while increasing decode throughput, which is *memory-bandwidth limited* rather than *compute limited*. The serving decision is which quantity should shrink: resident weights for fit, bytes moved per decode step for bandwidth, activations for prefill throughput, or the precision path supported by target hardware. Serving at scale also introduces distinct challenges: models must be quantized after training without access to training data, quality must be preserved across diverse inputs, and hardware deployment targets vary from data center GPUs to edge accelerators. Production inference therefore needs a decision model, not a method roster. The decision model in table 10.50 keeps the method choice tied to the serving constraint that actually binds the system.

Table 10.50: Serving Quantization Decision Model: Quantization methods are useful only when they relax the resource that actually binds the serving system. The method family follows from the bottleneck: fit, bandwidth, prefill compute, or outlier robustness.

Binding serving constraint	Representation change	Method family	Risk to test before rollout
Model does not fit in memory	Quantize weights only	GPTQ, AWQ	Calibration mismatch and quality regression
Decode is bandwidth-bound	Reduce bytes read per token	W4A16 weight-only paths	Kernel support and KV-cache headroom
Prefill is compute-bound	Quantize weights and activations	SmoothQuant W8A8	Activation outliers and INT8 hardware path
Outliers block low precision	Change the coordinate basis before quantization	Rotation-based methods	Transform overhead and limited runtime path

10.9.1 LLM-specific quantization challenges

Large language models present unique quantization challenges distinct from vision or recommendation models. The outlier activation problem occurs because certain attention heads produce activation magnitudes orders of magnitude larger than typical values. Naive quantization clips these outliers, causing significant quality degradation.

Consider a large language-model layer where most activation values are modest but specific channels produce much larger outliers (Xiao et al. 2023). Symmetric INT8 quantization with range [-127, 127] has no painless scale choice: a wide range preserves outliers but collapses typical values into too few bins, while a narrow range preserves resolution for typical values but clips outliers and introduces large errors.

The outlier distribution explains why the specialized methods that follow differ: each one protects a different part of the serving contract.

10.9.2 GPTQ: Layer-by-layer weight quantization

GPTQ (Frantar et al. 2023) is the weight-only choice when deployment needs 4-bit compression after training and can afford calibration data. Its serving value is that the model gets smaller without a full retraining run. The risk is that rounding one weight changes the layer output in a way later weights must absorb. GPTQ addresses that risk by using calibration activations to estimate which

[21] **Hessian Matrix in Quantization**: The Hessian $H = X^T X$ captures second-order sensitivity: weights with large diagonal entries have outsized impact on model outputs. GPTQ exploits this to quantize insensitive weights aggressively while preserving sensitive ones, achieving 3–4-bit quantization with less than 1 percent perplexity degradation. Without Hessian guidance, naive uniform quantization below 8-bit typically destroys model quality.

weight errors matter most,[21] then compensating the remaining unquantized weights as each column is rounded.

GPTQ processes the model one layer at a time, quantizing each weight matrix column by column[22] and compensating the still-unquantized columns so the layer's output drifts as little as possible. The serving consequence is a two-part deploy-time cost: a calibration pass over 128–256 representative samples that must match production traffic, and a per-layer second-order update that makes quantization minutes-to-hours of offline work rather than a one-shot cast. Both are paid once before deployment, so they do not touch serving latency, but a calibration set that misrepresents the deployed traffic mix is the failure mode to test for before rollout.

Table 10.51 summarizes GPTQ performance across Llama model sizes:

22 **GPTQ column-wise update**: GPTQ factors the inverse Hessian H^{-1} via Cholesky decomposition so the compensation becomes a stable triangular update: after rounding column q, the residual error is propagated to columns $q{+}1{:}$ weighted by $[H^{-1}][:, q{+}1{:}]/[H^{-1}]_{qq}$. The factorization avoids recomputing a full inverse per column, giving $\mathcal{O}(d_{\text{row}} \cdot d_{\text{col}}^2)$ work instead of the naive $\mathcal{O}(d_{\text{row}} \cdot d_{\text{col}}^3)$. Per-group scaling (group size 128) gives outlier channels a finer step than a single per-matrix scale.

Table 10.51: GPTQ Performance Across Llama Sizes: Quantization quality holds remarkably steady as model size scales: the 70B model loses less perplexity than the 7B model despite the same 4-bit target. Memory reduction is uniformly 4×. Quantization time scales near-linearly with model size, dominated by the per-layer second-order updates.

Model	Bits	Perplexity Increase	Memory Reduction	Quantization Time
Llama-7B	4	+0.3	4×	15 min
Llama-13B	4	+0.2	4×	30 min
Llama-70B	4	+0.15	4×	3 hours

GPTQ's strengths include fast quantization without retraining, minimal quality loss for 4-bit weights, and broad hardware compatibility. Its limitations include requiring calibration data, sensitivity to calibration set selection, and per-layer processing that cannot use cross-layer information.

10.9.3 AWQ: Activation-aware weight quantization

AWQ (Lin et al. 2024) attacks the same weight-only serving goal from the activation side. A calibration pass measures per-channel activation magnitudes, and the weights feeding the few high-magnitude channels are scaled up before quantization (with a matching scale folded into the next layer), so those salient weights keep their resolution while the rest are quantized aggressively. Its serving advantage over GPTQ is that protecting salient channels avoids the per-column error-feedback pass, so calibration is lighter and data-free of any reference output; the risk to test is that the salient-channel scaling must fuse into the deployed graph without a separate runtime op, or the memory win is eaten by an extra kernel.

Table 10.52 summarizes how AWQ compares to GPTQ across error compensation, calibration cost, quality, speed, and hardware compatibility.

Table 10.52: AWQ vs. GPTQ: AWQ scales salient channels rather than compensating residual error, achieves higher 4-bit quality at marginally slower calibration speed, and matches GPTQ on hardware compatibility.

Aspect	GPTQ	AWQ
Error compensation	Adjusts remaining weights	Scales salient channels
Calibration data	128–256 samples	128 samples
Quality (4-bit)	Very good	Excellent
Speed	Faster	Slightly slower
Hardware compatibility	Broad	Broad

AWQ can achieve 0.5-1 percent lower perplexity degradation than GPTQ at the same bit-width in the representative setting here, making it a plausible choice when the quality budget is tight.

10.9.4 SmoothQuant: Migrating quantization difficulty

SmoothQuant (Xiao et al. 2023) is the serving path for W8A8 deployments where activation quantization matters, especially compute-bound prefill. Activations carry unpredictable outliers that defeat INT8; weights do not. SmoothQuant divides activations by a per-channel scale and multiplies

the corresponding weights by the same scale, an algebraically identical transform that leaves the layer output unchanged while moving the hard-to-quantize range out of the activations and into the more forgiving weights. Its serving consequence is that the scales fold into the existing weights and a cheap activation division, so the smoothed model runs on a standard INT8 path at near-zero runtime overhead; the risk to test is that the migration only shifts outliers rather than removing them, so a calibration set whose activation extremes differ from production can still clip.

In the W8A8 deployment regime, SmoothQuant enables INT8 quantization for both weights and activations. Table 10.53 shows how W8A8 compares to FP16 baseline and weight-only quantization:

Table 10.53: W8A8 SmoothQuant vs. Weight-Only Quantization: W8A8 quantizes both weights and activations to INT8, doubling memory savings and unlocking compute-bound prefill speedup, but its decode speedup is lower than W8A16's because decode is memory-bound and W8A8's smaller weights are already captured by the simpler weight-only scheme.

Configuration	Memory	Prefill Speedup	Decode Speedup	Quality
FP16 (baseline)	1×	1×	1×	Baseline
W8A16 (weights only)	2×	1.3×	1.8×	<0.5% loss
W8A8 (SmoothQuant)	2×	1.8–2×	1.3–1.5×	<1% loss

The critical distinction is that W8A8 provides near 2× speedup for compute-bound prefill (large batch processing initial prompt) but only 1.3–1.5× speedup for memory-bound decode (generating tokens one at a time). LLM serving is typically decode-heavy, so real-world throughput improvements from W8A8 are often 1.3–1.7× rather than the theoretical 2× compute throughput of INT8 Tensor Cores.

10.9.5 Rotation-based quantization

Traditional quantization methods (GPTQ, AWQ, SmoothQuant) address outliers through compensation or migration. **Rotation-based quantization** (Ashkboos et al. 2024) is the aggressive path when activation outliers block lower precision: it mathematically transforms the weight and activation space to eliminate outliers entirely.

The crucial realization is that outliers are artifacts of the coordinate basis representation. Rotating to a different basis spreads extreme values uniformly, making all values quantization-friendly.

10.9.5.1 QuaRot (quantization with rotation)

QuaRot rotates the weight and activation spaces with an orthogonal Hadamard transform[23] so that any single outlier is spread evenly across all dimensions, leaving a distribution with no dominant value that uniform low-bit quantization can capture. Because the rotation is orthogonal, the layer output is unchanged and the transform needs no calibration data, which is what lets QuaRot reach W4A4 where the compensation and migration methods stop at higher precision. The serving consequence is a runtime cost the other methods do not pay: the rotation runs inline on every forward pass, adding roughly 3 percent overhead, so the test before rollout is whether the W4A4 memory-and-bandwidth win clears that standing tax on the target hardware.

Table 10.54 contrasts QuaRot's rotation-based approach with SmoothQuant's migration. The operating trade-off is calibration and runtime cost: SmoothQuant avoids runtime overhead after calibration but stops at W8A8, while QuaRot reaches W4A4 by paying an inline rotation cost.

Table 10.54: QuaRot vs. SmoothQuant: SmoothQuant migrates outliers from activations into weights and runs at zero runtime overhead but requires calibration data and stops at W8A8. QuaRot eliminates outliers through Hadamard rotation, runs data-free, reaches W4A4, and pays a ~3 percent runtime cost for the inline transforms.

Aspect	SmoothQuant	QuaRot
Calibration data	Required	Not required (data-free)
Minimum precision	W8A8	W4A4
Runtime overhead	~0%	~3% (Hadamard transforms)
Outlier handling	Migration	Elimination

[23] **Hadamard rotation in QuaRot**: QuaRot transforms activations as $X' = X \cdot H$ and weights as $W' = H^T \cdot W$, where H is an orthogonal Hadamard matrix, computed structurally without being stored. Orthogonality preserves $XW = X'W'^T$, so the layer output is unchanged. The outlier-spreading effect is concrete: a vector $[1000, 1, 1, 1]$ with one extreme value becomes roughly $[502, 500, 500, 500]$ after the transform, so no single coordinate dominates the quantization range.

Aspect	SmoothQuant	QuaRot

Table 10.55 compares perplexity across quantization methods and precision choices on LLaMA-2-70B:

Table 10.55: Quantization Perplexity on LLaMA-2-70B: Perplexity degradation across four quantization methods on a 70B model. QuaRot's W4A4 lands within 0.2 perplexity of FP16 baseline while delivering roughly 4× memory reduction for quantized tensors, demonstrating that rotation-based methods make 4-bit activation quantization viable for production inference.

Method	Precision	LLaMA-2-70B Perplexity	vs. Baseline
FP16	W16A16	3.12	Baseline
SmoothQuant	W8A8	3.18	+0.06
GPTQ	W4A16	3.24	+0.12
QuaRot	W4A4	3.31	+0.19

QuaRot achieves 4-bit weights *and* 4-bit activations with quality competitive to GPTQ's 4-bit weights only, enabling approximately 4× memory reduction vs. FP16 for quantized weights and activations. SpinQuant extends QuaRot by learning optimal rotation matrices during a short fine-tuning phase, improving quality at the cost of training compute. Rotation methods therefore extend low-precision reach, but whether the trade-off pays depends on the deployment hardware's support for the target precision.

10.9.6 Hardware-deployment co-design

The quantization method only pays off when the deployment hardware accelerates the chosen precision. Different accelerators support formats such as BF16, INT8, and INT4 with varying performance multipliers.

Table 10.56 summarizes NVIDIA Tensor Core precision support across Ampere and Hopper:

Table 10.56: Tensor Core Precision Support and Speedup: Native precision support across NVIDIA Ampere and Hopper Tensor Cores. The 2× speedup at INT8 and FP8 reflects the doubled throughput of those native paths; INT4 remains framework-dependent on Ampere and Hopper, with native FP4 first appearing in Blackwell.

Format	Ampere (A100)	Hopper (H100)	Speedup vs. FP16
FP16	Yes	Yes	1×
BF16	Yes	Yes	1×
INT8	Yes	Yes	2×
FP8 (E4M3)	No	Yes	2×
INT4	Software/CUTLASS or framework-dependent	Software/framework-dependent on H100; native FP4 is a Blackwell feature	workload-dependent

Memory bandwidth dominates autoregressive LLM decode throughput because each token reads the entire model:

$$\text{Decode throughput} \propto \frac{\text{Memory bandwidth}}{\text{Model size in bytes}}$$

In the ideal memory-bound case, reducing FP16 weights to 4-bit values cuts weight traffic by 4×, but realized decode throughput depends on kernel support, dequantization overhead, KV-cache precision, batch shape, and whether HBM bandwidth is truly the bottleneck. The value of 4-bit serving is therefore strongest when it increases model residency or batch capacity, and throughput gains should be validated on the target serving workload.

Table 10.57 shows that the configuration follows two things the deployment fixes in advance: the precision path the target hardware supports and whether the goal is memory residency or throughput. Memory-constrained consumer GPUs take W4A16, INT8 accelerators take W8A8 for Tensor Core throughput, and H100-class hardware takes its native FP8 path.

Table 10.57: Quantization Deployment Configurations: Mapping from quantization scheme to deployment target and supporting frameworks. W4A16 is common for consumer and memory-constrained deployments; W8A8 unlocks INT8 Tensor Core throughput; FP8 is the native low-precision path on H100/H200-class deployments; W4A4 is an aggressive compression target with more limited framework support.

Quantization	Best For	Framework Support
W4A16 (GPTQ/AWQ)	Consumer GPUs, memory-constrained	vLLM, TensorRT-LLM, llama.cpp
W8A8 (SmoothQuant)	INT8 accelerators, high throughput	TensorRT-LLM, ONNX Runtime
FP8	H100/H200 deployments	TensorRT-LLM
W4A4	Research, extreme compression	Limited

10.9.7 Runtime contract for quantized serving

Production serving frameworks matter because quantized weights reduce cost only when the runtime preserves the theoretical savings. The durable contract is framework-independent: the serving stack must load the quantized artifact without silently dequantizing it, choose kernels that execute the intended low-precision path, allocate KV cache around the smaller weight footprint, and expose enough telemetry to detect quality or latency regressions. Table 10.58 lists these responsibilities alongside the way each one silently leaks the theoretical saving when the runtime fails it, which is why all four together form the minimum contract a framework must satisfy before quantization becomes a production-serving win.

Table 10.58: Quantized Serving Runtime Contract: Framework-specific APIs differ, but every production path has to preserve representation, choose supported kernels, plan memory, and validate quality under the deployed traffic mix.

Runtime responsibility	Why it matters
Preserve the quantized artifact	Silent dequantization restores the FP16 memory and bandwidth cost.
Select compatible low-precision kernels	Unsupported operators fall back to slower or higher-precision execution.
Plan memory around weights and KV cache	The capacity gain matters only if the freed memory becomes useful batch headroom.
Validate quality and latency together	A lower-bit path that meets latency but fails task quality is not a serving win.

Quantized models combine with PagedAttention for maximum memory efficiency:

$$\text{Max batch} = \frac{\text{GPU Memory} - \text{Quantized Weights}}{\text{KV Cache per Sequence}}$$

A 70B model with 4-bit weights requires approximately 35 GB, leaving 45 GB on an 80 GB A100 for KV cache. With FP16 KV cache at about 2.6 GB per 1K-token sequence for the MHA baseline, this supports about 17 concurrent 1K-token sequences before other runtime overheads. With FP16 weights, the same 70B model would *not* fit on one A100 at all, so weight-only quantization changes both feasibility and batch capacity.

10.9.8 Quantization selection guidelines

The final selection matches the method to the binding constraint. Latency-sensitive paths should prefer FP16 or BF16 when quantization overhead would dominate, while throughput-oriented paths are more likely to benefit from quantization. Hardware support narrows the viable choices: H100 and H200 deployments can consider FP8 because the hardware provides native support with minimal quality loss; A100 and A10G deployments usually choose between W8A8 and W4A16 depending on the workload; consumer GPUs often require W4A16 simply to fit the model. The

quality budget then sets the lower precision bound. If less than 0.5 percent degradation is acceptable, AWQ 4-bit is a plausible target; if less than 1 percent degradation is acceptable, GPTQ 4-bit or SmoothQuant W8A8 may be viable; if no degradation is acceptable, FP16 or BF16 remains the safest choice. The binding resource makes the final distinction: compute-bound prefill favors W8A8 because it can provide a 2× speedup, whereas memory-bound decode favors W4A16 because it can provide a 4× effective bandwidth increase.

Table 10.59 shows how quantization reshapes serving cost per million tokens:

Table 10.59: Quantization Impact on Serving Cost: Estimated cost per million tokens at three configurations. Halving the GPU count via 4-bit quantization halves the cost; halving it again brings the per-million-token cost to one quarter of the FP16 baseline. The cost savings come almost entirely from fewer GPUs rather than from per-GPU throughput gains.

Configuration	Cost per 1M tokens (estimated)
FP16 on 8× A100	$2.40
AWQ 4-bit on 4× A100	$1.20
AWQ 4-bit on 2× A100	$0.60

Quantization can reduce serving costs by 2–4× while maintaining acceptable quality, making it an important lever for cost-effective LLM deployment.

These serving optimizations span continuous batching, PagedAttention, global load balancing, and extreme quantization. The case studies that follow show how production systems at global scale combine these techniques to meet specific cost and latency targets.

10.10 Case Studies

Production-scale serving systems force the chapter's techniques to meet real workload variance, latency budgets, and operational constraints. The cases bind on different constraints: embedding scale, variable-length generation, cascade cost, or multimodal freshness. The examples show how systems combine batching, sharding, caching, routing, and quantization to meet specific cost and latency targets. The Orca and vLLM mechanisms developed earlier provide the LLM-serving primitives; the cases below broaden the view to recommendation serving, global request routing, ranking cascades, and multimodal freshness.

10.10.1 Meta recommendation serving

Meta's recommendation infrastructure binds on embedding scale rather than dense forward-pass compute. It serves predictions for feeds, ads, and content ranking across Facebook, Instagram, WhatsApp, and Messenger, making it one of the largest production inference deployments in the world.

The scale numbers explain why embedding locality dominates the design:

- Request volume: Billions of requests per day
- Latency target: <10 ms P99
- Model diversity: Hundreds of model variants
- Feature cardinality: Trillions of unique entities

The serving stack separates sparse lookup from dense ranking:

```
User request → Feature collection → Embedding lookup → Model inference → Response
```

```
Feature Store        Embedding Servers      GPU Inference
(CPU, DRAM)         (CPU + SSD, 1000s)     (GPU, 100s)
```

In Meta's architecture, the binding design constraint is embedding scale: tables total over 100 TB, requiring 1,000+ shards. Meta uses a hybrid sharding strategy:

- Hot embeddings (top 1 percent): Replicated across memory on all inference servers
- Warm embeddings (next 10 percent): Column-sharded with 8-way parallelism
- Cold embeddings (remaining 89 percent): Row-sharded with consistent hashing, SSD-backed

Hybrid sharding reduces embedding lookup latency from 50 ms (naive) to 2 ms through batching and locality optimization.

Instead of batching entire requests, Meta batches at the feature level. Each inference request triggers 5,000+ embedding lookups, but these lookups are batched across requests within a 1 ms window. This achieves 90 percent+ memory bandwidth utilization on embedding servers.

The resulting GPU-CPU hybrid architecture runs dense model computation (ranking towers) on GPUs, while sparse embedding lookups run on CPU servers with large memory and SSD storage. Table 10.60 places each component on the hardware whose strength matches its bottleneck: the memory-bound sparse lookups land on SSD-backed CPU servers, the low-intensity feature processing stays on CPU, and only the compute-dense ranking pass uses the GPU.

Table 10.60: Meta DLRM Hybrid CPU/GPU Architecture: Hardware-workload mapping for Meta's recommendation serving stack. CPU + SSD handles the bandwidth-bound sparse path; CPU handles low-arithmetic-intensity feature processing; GPU handles the compute-dense ranking forward pass. The three latency budgets sum to ~5 ms, well within typical recommendation SLOs.

Component	Hardware	Latency	Throughput
Embedding lookup	CPU + SSD	2 ms	50M lookups/s
Feature processing	CPU	1 ms	10M ops/s
Dense ranking	GPU	1.5 ms	100K infs/s

The serving lesson is that recommendation latency is often dominated by embedding lookup rather than dense model inference. Feature-parallel batching and a hybrid CPU-GPU architecture match the hardware to the sparse and dense halves of the workload, which sets up a contrast with GPT-style API serving where request variance and autoregressive state dominate.

10.10.2 OpenAI API infrastructure

OpenAI-style API infrastructure binds on request variance: short prompts, long-context requests, and different model sizes share capacity while users expect predictable TTFT. The infrastructure serves GPT-class models to large developer and application workloads while maintaining quality of service across diverse workloads.

The scale numbers explain why continuous batching and admission control dominate:

- Request volume: Millions of requests per hour
- Latency target: Time-to-first-token (TTFT) <2s, throughput varies by model
- Model sizes: Billions to hundreds of billions of parameters
- Context lengths: Up to 128K tokens

The serving stack separates routing, scheduling, cache management, and shard groups:

```
API Gateway → Rate Limiting → Request Router → Model Cluster → Response Streaming
```

```
Model Pool

  Large
  8×H100


  Smaller
```

```
4×A100
```

The main design decision is to prevent long prompts from blocking decode service for everyone else.

An Orca-style long-context serving design uses continuous batching to maintain high GPU utilization despite variable output lengths. Chunked prefill bounds decode latency by processing long prompts in chunks that interleave with ongoing generation. Table 10.61 contrasts the three approaches:

Table 10.61: Batching Strategy Comparison: Three batching approaches with the resulting GPU utilization and 128K-prompt behavior. Static batching lets the longest request idle the whole batch; continuous batching lifts utilization to 75 percent but still stalls decode behind long prefills; chunked prefill interleaves prefill chunks with decode, capping decode stalls at a few seconds while reaching 85 percent utilization.

Batching Strategy	GPU Utilization	128K Prompt Behavior
Static batching	45%	30s TTFT; decode blocked
Continuous batching	75%	30s TTFT; decode still blocked by prefill
Continuous + chunked	85%	30s TTFT; decode stalls bounded to ~3s

Large GPT-class models often require 8-way or greater tensor parallelism for memory capacity and latency:

- 8× H100 per large-model shard group
- NVLink for intra-node communication
- Consistent hashing for session affinity (KV cache reuse)

OpenAI implements rate limiting at multiple levels to prevent noisy neighbors:

- Per-API-key request rate limits
- Per-API-key token-per-minute limits
- Organization-level capacity quotas
- Global model capacity limits

During peak demand, OpenAI shifts capacity between models based on queue depth:

```
if large_model_queue_depth > threshold:
    # Migrate some smaller-model capacity to the larger model
    reallocate_cluster_capacity(from="smaller-model", to="large-model", fraction=0.2)
```

The serving lesson is that LLM APIs are governed by variance control: continuous batching, prefix caching, and multi-tier rate limiting keep long prompts, repeated conversational context, and traffic spikes from turning one user's request into everyone else's latency. Search ranking faces a different shape of the same problem: many cheaper models must cooperate under one strict deadline.

10.10.3 Google search ranking

Google Search binds on cascade cost: each query coordinates many smaller models under one end-to-end latency budget rather than serving one large model. The ensemble combines specialized models for query understanding, document relevance, and result ranking.

The scale numbers explain why a cascade is mandatory:

- Request volume: Billions of searches per day
- Latency target: <200 ms end-to-end
- Model count: Dozens of models per query
- Result processing: Thousands of documents per query

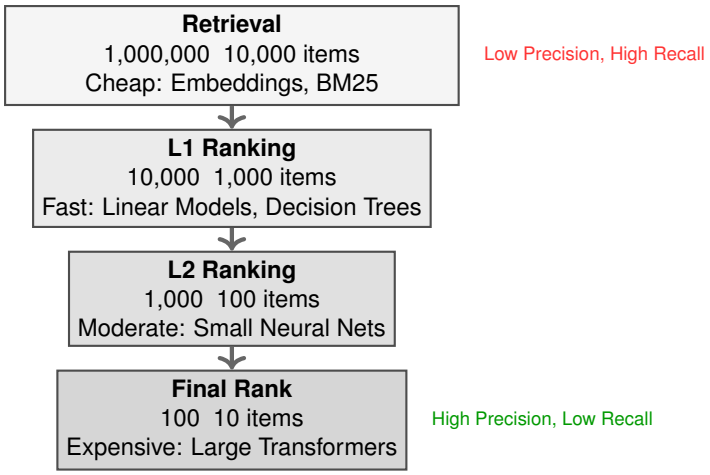


Figure 10.26: Ranking Cascade Architecture: To optimize latency and cost, search and recommendation systems use a cascade of increasingly complex models. Early stages (Retrieval) filter millions of candidates down to thousands using fast, cheap heuristics. Later stages use expensive, high-precision models only on the most promising candidates.

To meet this latency budget while evaluating thousands of documents, the system employs a ranking cascade (figure 10.26) that progressively narrows the candidate set through increasingly expensive models.

The central design decision is progressive refinement: rather than running one expensive model on all candidates, Google uses a ranking cascade. Table 10.62 shows the trade the cascade makes: each stage spends a larger latency budget on a smaller candidate set, so the expensive L3 ensemble is affordable only because L0 through L2 have already cut the candidates by four orders of magnitude.

Table 10.62: Google Ranking Cascade: Four-stage cascade that filters a million candidates down to ten while spending compute proportional to depth. Each stage uses a more expensive model on fewer candidates, so total cost is dominated by the cheap early stages rather than the expensive final ensemble.

Stage	Model Complexity	Candidates	Latency Budget
L0 (Retrieval)	Embedding lookup	1,000,000 → 10,000	10 ms
L1 (First pass)	Linear model	10,000 → 1,000	20 ms
L2 (Second pass)	Small transformer	1,000 → 100	50 ms
L3 (Final rank)	Large ensemble	100 → 10	100 ms

The cascade reduces final L3 evaluations by 10,000× compared to running L3 on all candidates. Under this simplified request-budget model, the L3 stage costs 1 ms per candidate. Applying L3 to the full million-candidate set would therefore consume about 1000 s of model work, while the cascade budget sums to 180 ms. The resulting model-work reduction is roughly 5,600×.

Given tight latency budgets, Google uses speculative execution for model ensembles. Here, speculation means launching candidate ranking work in parallel under a deadline, not draft-token verification for LLM decoding:

```
# Instead of sequential:
#   q1 = model1(query)
#   q2 = model2(query)
#   q3 = model3(query, q1, q2)

# Speculative parallel:
async_q1 = async model1(query)
async_q2 = async model2(query)
```

```
async_q3 = async model3(query, predicted_q1, predicted_q2)

# Use actual results if they arrive in time, otherwise use speculative
```

Search ranking deployments can run ranking models on Tensor Processing Units (TPUs)[24] optimized for transformer inference. TPU pods provide:

- 2D mesh topology for efficient AllReduce
- High memory bandwidth for attention operations
- Custom quantization for serving efficiency

Each sub-request carries a deadline, and workers prioritize by deadline proximity:

```
Worker queue: [Doc1: 50 ms left] [Doc2: 30 ms left] [Doc3: 80 ms left]
                                    ↑ Process first

If deadline will be missed:
  Return cached/default result rather than timing out
```

The serving lesson is that ranking cascades reduce cost by spending expensive models only on a shrinking candidate set, while deadline propagation and priority scheduling keep the ensemble inside a strict latency budget. Custom hardware can improve predictability when the workload matches the accelerator, but the next case adds a freshness constraint that hardware alone cannot solve.

10.10.4 TikTok multimodal recommendation

TikTok's recommendation system binds on freshness under a tight ranking SLO: video understanding is expensive, new content arrives continuously, and user modeling remains online. It combines video understanding (vision) with user modeling (recommendation) for personalized content ranking, creating a multimodal inference challenge where different model types must coordinate.

The scale numbers explain why online/offline separation matters:

- Request volume: Millions of video rankings per second
- Latency target: <50 ms P99
- Content volume: Millions of new videos daily
- Modalities: Video, audio, text, user signals

The serving stack separates content and user paths:

```
User request → User embedding → Candidate videos → Video understanding → Ranking
```

```
   User Tower          Video Cache          Vision Models
(Transformer)      (Pre-computed)          (On-demand)
```

TikTok's architecture separates user understanding (online) from content understanding (offline) through a two-tower architecture with caching[25]. Table 10.63 contrasts the online and offline tower update cadences:

Table 10.63: TikTok Two-Tower Update Cadence: User and video towers run on different cadences: user features change second-to-second and must run inference online; video features change much more slowly and are precomputed in hourly batches. This asymmetric scheduling is the design that takes vision inference out of the critical path for almost every request.

Tower	Update Frequency	Latency	Compute
User tower	Real-time	5 ms	GPU (online)
Video tower	Hourly	N/A	GPU (batch)

[24] **TPU for Inference**: TPU systolic arrays provide deterministic, low-variance matrix execution for ranking-style inference. GPU latency can vary with batch composition, while TPU-style execution can deliver consistent per-request timing for strict search SLOs where P99 latency directly affects revenue.

[25] **Two-Tower Architecture**: Encodes user features and item features through separate neural networks ("towers") into embeddings, then scores via dot product. The serving advantage: item embeddings can be precomputed offline and cached, reducing online inference to the user tower forward pass plus a sub-millisecond vector similarity lookup. The trade-off is reduced model expressiveness from late interaction, since the towers cannot attend to each other's features during encoding.

Video embeddings are precomputed and cached, eliminating vision inference from the critical path for most requests. Only new videos (uploaded within the hour) require online vision inference.

Like Meta, TikTok uses CPU for embedding operations and GPU for dense model computation:

```
User features → CPU preprocessing (1 ms)
             → Embedding lookup (2 ms, CPU+DRAM)
             → Dense ranking (10 ms, GPU)
             → Response formatting (1 ms)
```

New video content is processed with different priorities, as table 10.64 shows:

Table 10.64: TikTok Video Ingestion Priority Tiers: Three-tier SLA for video ingestion. High-follower-count creators get a five-minute SLA to maximize content freshness for their audience; standard uploads get 30 minutes; bulk imports get two hours of slack so the system can absorb them during quiet periods.

Priority	SLA	Use Case
Critical	5 min	Creator with large following
Standard	30 min	Normal uploads
Background	2 hours	Bulk/imported content

The priority scheme ensures popular creators' content reaches recommendations quickly while managing compute costs.

TikTok combines multiple understanding modalities through late fusion, which combines independently computed modality embeddings or scores near the end of ranking rather than running one joint model over all raw modalities:

```
Video embedding (512d)
Audio embedding (256d)    Concat → Fusion MLP → Final embedding (256d)
Text embedding (256d)
```

Late fusion allows independent updates to each modality's model without retraining the full system.

The serving lesson is that freshness requires separating online and offline work. Two-tower architectures make aggressive caching possible because item embeddings can be refreshed asynchronously, while priority-based processing spends the freshest compute where it has the largest user impact.

10.10.5 Cross-cutting observations

Across these cases, the common pattern is specialization instead of a universal serving stack. Systems separate embedding and retrieval from ranking and generation, use hybrid CPU-GPU architectures to match hardware to workload characteristics, cache at multiple levels while accepting staleness, progressively refine candidate sets to avoid expensive work on unlikely items, and propagate deadlines so quality/latency trade-offs stay explicit under pressure. Table 10.65 identifies the primary technique and key innovation from each system. The four differ precisely because each is specialized to its own workload; that specialization is the shared pattern, not a contradiction of it.

Table 10.65: Case Study Summary: Each system innovates on a core technique matched to its workload characteristics.

System	Primary Technique	Key Innovation
Meta	Embedding sharding	Feature-parallel batching
OpenAI	Continuous batching	Chunked prefill
Google	Ranking cascade	Speculative execution
TikTok	Two-tower caching	Multimodal fusion

The case studies reveal how deeply custom engineering is required to serve ML at scale. Engineers attempting to replicate these architectures often fall victim to conventional wisdom that does not translate to GPU-bound workloads, leading us to the field's most common fallacies and pitfalls.

10.11 Fallacies and Pitfalls

A DevOps team applies their standard web-server load balancing rules to a new fleet of GPU inference nodes, only to watch tail latency explode because they ignored the extreme variance of autoregressive token generation. Inference at scale breaks the rules of traditional microservices, creating dangerous fallacies for engineers transitioning from standard web development to machine learning systems.

Fallacy: *Inference at scale is synonymous with LLM serving.*

The misconception, reinforced by LLM-centric discourse, leads to over-focus on LLM-specific techniques while ignoring the broader inference landscape. In large consumer platforms, recommendation systems can constitute the majority of AI inference cycles or capacity pressure, with vision, language, fraud, ads, and other models comprising the remainder (Gupta et al. 2020; Hazelwood et al. 2018). A practitioner who only understands continuous batching and KV cache management is unprepared for the feature-parallel batching and embedding sharding that dominate high-volume recommendation inference. Technique selection must match the actual workload.

Pitfall: *Using training infrastructure for production serving.*

Training and serving have different requirements. Training optimizes for aggregate throughput over hours or days; serving optimizes for per-request latency under strict SLOs. Training tolerates batch sizes of thousands; serving often requires batch sizes in single digits. Training accepts checkpoint-based recovery; serving requires graceful failover without user impact. Teams that deploy training clusters for serving often discover unacceptable latency variance, poor resource utilization, and difficulty meeting SLOs. Purpose-built serving infrastructure with appropriate batching, load balancing, and autoscaling is essential.

Fallacy: *Continuous batching solves the whole LLM serving problem.*

Continuous batching dramatically improves GPU utilization for LLM serving, but it addresses only one dimension of the problem. Prefill remains a bottleneck for long contexts, as the quadratic attention computation cannot be avoided regardless of how subsequent decode iterations are batched. As discussed in section 10.4, KV cache memory, not compute, often limits batch size. Network bandwidth between sharded model components can dominate latency for large models. Continuous batching is necessary but not sufficient for efficient LLM serving.

Pitfall: *Sizing the fleet by average throughput.*

Queuing theory establishes that systems provisioned for average load violate SLOs during traffic peaks. At 80 percent average utilization, a modest 25 percent traffic spike pushes utilization above 100 percent, causing unbounded queue growth. As discussed in section 10.8, the cold start problem exacerbates this: by the time new capacity is available, which can take multiple minutes for GPU instances, SLO violations have already occurred. Capacity planning must account for peak load plus headroom, not average load.

Fallacy: *Load balancing does not matter much for inference.*

Simple load balancing strategies like round-robin seem adequate until examined quantitatively. As shown in section 10.6, random assignment produces maximum queue lengths of $\mathcal{O}(\log R / \log \log R)$ across R servers. Power-of-two-choices reduces this to $\mathcal{O}(\log \log R)$, an exponential improvement. For a 1,000-server cluster, this translates from ~4-5 requests maximum queue to ~2 requests. At the tail latencies that determine SLO compliance, this difference is substantial. The choice of load balancing algorithm has first-order impact on system performance.

Pitfall: *Ignoring the serving tax.*

Distributed inference introduces overhead absent from single-machine serving: network round-trips, serialization, load balancer decisions, and coordination for sharded models. This "serving tax" often consumes 10–30 percent of the latency budget. A team that achieves 70 ms model inference on a single GPU may be surprised when end-to-end latency reaches 100 ms in production due to these overheads. Latency budgets must explicitly account for distribution overhead, not just compute time.

Fallacy: *More GPU memory always means larger batch size and higher throughput.*

Larger GPU memory enables larger batches only for models whose throughput is capacity-bound. The binding constraint in LLM decode is *memory bandwidth*, not *capacity*: once HBM bandwidth saturates, adding memory does not increase tokens per second. The bottleneck hierarchy makes the mistake visible.

While larger GPU memory enables larger batches for models that fit in memory, the bottleneck often shifts before memory is exhausted. Memory bandwidth limits throughput for bandwidth-bound operations (LLM decode). Compute limits throughput for compute-bound operations (prefill, vision inference). A bandwidth-bound decode workload on an H100 with 80 GB memory eventually plateaus when HBM3 bandwidth (3.35 TB/s) saturates; adding memory alone does not raise token throughput once that point is reached. Understanding whether the workload is compute-bound, memory-bandwidth-bound, or capacity-bound guides appropriate resource allocation.

Pitfall: *Neglecting multi-tenancy isolation until production.*

In development and staging, single-tenant deployments work well. In production, noisy neighbors cause sudden, unpredictable performance degradation that is difficult to diagnose and resolve. A tenant bursting to 5× normal traffic can degrade latency for all other tenants on shared infrastructure. As emphasized in section 10.7, resource quotas, priority scheduling, and bulkhead isolation must be designed into the system from the start, not retrofitted after production incidents.

Avoiding these pitfalls keeps serving infrastructure resilient to traffic variance, cost-efficient under fixed SLOs, and consistently responsive across tenants.

10.12 Summary

Inference at scale is the service layer that turns trained models and distributed infrastructure into low-latency global predictions. It is where throughput, tail latency, memory residency, and tenant isolation become user-visible system behavior.

The serving hierarchy organizes the optimization problem from request-level batching and caching to replica-level GPU memory management, service-level load balancing, and platform-level multi-tenancy. The inversion from training to inference changes the dominant metric: aggregate throughput remains important, but P99 latency determines whether the service feels usable and whether downstream systems can depend on it.

Different model architectures require different batching strategies. Vision models benefit from static or dynamic batches because inputs have predictable shapes. Recommendation systems batch and shard sparse features because embedding lookup dominates. LLMs need continuous batching and KV-cache management because every active request carries private sequence state. Model sharding techniques such as tensor and expert parallelism reduce latency for very large models only when the networking fabric discussed in Chapter 6 can sustain the synchronization overhead.

Understanding inference at scale matters because serving cost, not training cost, can dominate the total economics of machine learning in production. A very large model may require tens of millions of dollars and several months to train, but that investment is amortized once. Serving that same model to millions of users over its operational lifetime can exceed the original training expenditure by large factors when request volume is high and the model stays in service for years. This asymmetry means that every optimization discussed in this chapter, from continuous batching to PagedAttention to power-of-two-choices load balancing, translates directly into sustained cost savings that compound over the model's deployment period.

The engineer who internalizes the prefill/decode split, who understands why wasted KV-cache memory destroys throughput, and who can reason about the interaction between batching strategy and tail latency can make serving economically viable under fixed SLOs. Choosing continuous batching over static batching can raise utilization when output lengths vary; implementing PagedAttention can recover memory otherwise lost to KV-cache fragmentation; and disaggregating prefill from decode can lower per-token cost when the workload mix justifies separate hardware pools. In aggregate, the techniques presented in this chapter represent the difference between inference infrastructure that scales economically and one whose serving bill grows faster than useful demand.

Key Takeaways: Serving inverts every assumption

- **Serving cost compounds forever**: Training is paid once, but inference OpEx accrues on every request; for high-traffic production systems it can dominate by 100× or more over a model's lifetime. Milliseconds, memory fragments, and utilization points are financial levers, not local optimizations.

- **Tail latency governs architecture**: Inference systems optimize under P99 SLOs, so batching, sharding, caching, and autoscaling must raise utilization without spending the user-visible deadline. Aggregate throughput is necessary but insufficient when a slow request can break downstream services.
- **Decode turns memory into capacity**: LLM prefill is compute-bound, while autoregressive decode rereads weights and expands private KV state token by token. Continuous batching, PagedAttention, prefix reuse, and KV compression matter because they convert scarce HBM bandwidth and memory into admitted requests.
- **Model class sets the primitive**: Vision models benefit from predictable static or dynamic batches, recommenders from feature-parallel embedding sharding, and LLMs from iteration-level scheduling. A fleet scheduler must match the workload shape instead of applying one universal batch-size rule.
- **Distribution always sends a bill**: Tensor parallelism, expert routing, global load balancing, multi-tenancy, and failover all buy capacity or isolation by adding communication and coordination. Production serving works only when that tax is explicitly budgeted inside latency, cost, and reliability envelopes.

The cost that decides an inference system is usually serving OpEx, not the one-time training bill. A frontier model's training run is paid once, while serving is paid on every request for as long as the model remains in production; a small utilization gain can therefore outweigh a large training expense. The difficulty is physical: autoregressive generation produces one token at a time, and each decode step rereads model weights from memory, so bandwidth rather than arithmetic often limits throughput. The techniques in this chapter improve utilization under that constraint, and at serving scale each saved millisecond, memory page, or idle accelerator slot is collected across the full request stream.



What's Next: From data center to edge

The architectural foundations for inference at scale developed here, continuous batching, PagedAttention, model sharding, and load balancing, all assume data center-grade hardware: H100 GPUs, NVLink interconnects, terabytes of HBM. Many applications, however, require inference closer to the user: on smartphones, IoT devices, or local gateways where power budgets are measured in milliwatts, memory in megabytes, and connectivity is intermittent. Chapter 11 examines how serving principles change when intelligence moves to the edge and every data center assumption is inverted.

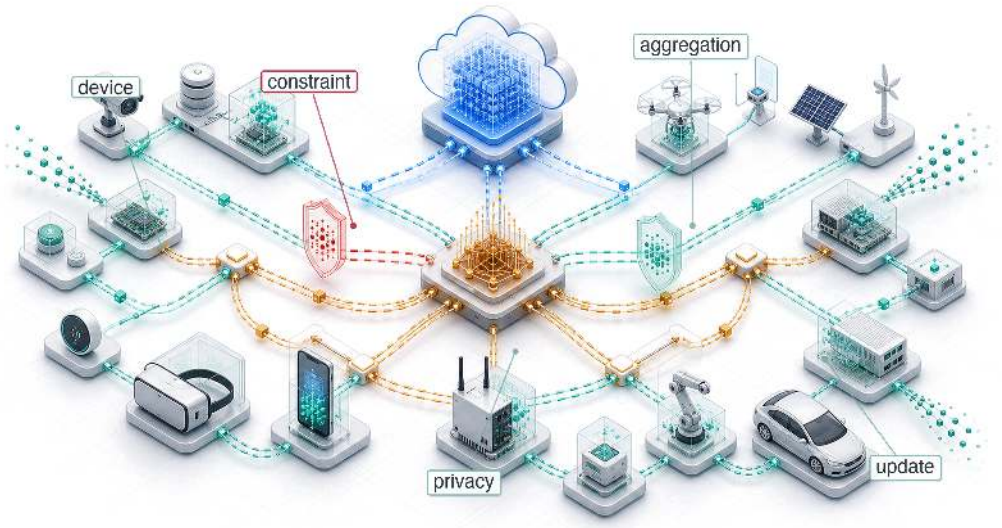


Research Questions: For further inquiry

- How should lifetime serving economics reshape architecture compared with one-time training efficiency?
- How should continuous batching and KV-cache policies use performance-engineering tools while preserving p99 latency?
- When should a service choose sharding, disaggregation, quantization, or a smaller model instead of adding replicas?
- How should routing account for request length variability, state affinity, and noisy-neighbor interference?

11

Edge Intelligence



11.1 The Edge Learning Paradigm
11.2 Design Constraints
11.3 Model Adaptation
11.4 Data Efficiency
11.5 Federated Learning: Algorithms
11.6 Federated Learning: Systems at Scale
11.7 Continual Adaptation Under Edge Budgets
11.8 Production Integration
11.9 Putting the Pillars Together
11.10 Engineering Challenges and Mitigations
11.11 Fallacies and Pitfalls
11.12 Summary

Purpose

Why must intelligence adapt where it operates rather than remain frozen from distant training?

A model trained in a data center encounters a world it has never seen when deployed to a user's device. The user's vocabulary differs from training data. Local conditions (lighting, acoustics, usage patterns) diverge from the distributions that shaped the model's parameters. Over time, the gap widens as user behavior evolves while the model remains static. On-device learning exists because centralized training cannot anticipate every context, personalize for every user, or adapt to continuous change. It also exists because data cannot always travel: privacy constraints, bandwidth limitations, and latency requirements may prohibit sending raw observations to distant servers. The alternative to on-device learning is accepting that deployed models grow stale, that personalization requires privacy sacrifice, and that disconnected operation means degraded intelligence. On-device learning rejects these compromises by bringing the learning process to where the data lives and where the adaptation matters. Edge intelligence turns adaptation into a C^3 problem: compute moves onto the device, communication shrinks from raw observations to selective updates, and coordination must operate under power and memory budgets the data center never sees.

Learning Objectives

- Contrast centralized cloud training, on-device learning, and federated learning by analyzing their data flow, privacy properties, and operational trade-offs
- Quantify edge-device training overhead relative to inference-only deployment across memory, compute, and energy
- Select model adaptation strategies by comparing memory footprints, expressivity, and convergence for specific device capabilities
- Apply data-efficiency techniques (few-shot learning, experience replay, contrastive learning) to learn from limited local data without catastrophic forgetting
- Design federated learning systems that aggregate privacy-preserving updates while managing heterogeneous data, communication efficiency, and stragglers
- Explain the operational controls needed for on-device learning: device-aware deployment, distributed validation, privacy-preserving monitoring, and rollback across heterogeneous populations

11.1 The Edge Learning Paradigm

Imagine a voice assistant trained on millions of generic audio samples in a cloud data center. When deployed, it struggles to understand a user's heavy regional accent or specific household vocabulary. If the model cannot adapt locally on the device itself, it remains permanently frozen and becomes less useful as local usage diverges from the training distribution. **Edge intelligence** places inference, adaptation, and coordination on or near the devices that observe the data, so models can respond to local context under power, memory, privacy, and connectivity constraints. The edge learning paradigm shifts ML from a *centralized factory model* to a *decentralized, continuously adapting network*.

Data center ML systems assume a controlled environment where computational resources are abundant, network connectivity is reliable, and system behavior is predictable. Centralized inference is the clearest case of that assumption, and the edge breaks it.

A smartphone learning to predict user text input, a smart home device adapting to household routines, or an autonomous vehicle updating its perception models based on local driving conditions all demonstrate scenarios where traditional centralized training approaches prove inadequate. The smartphone encounters linguistic patterns unique to individual users that were not present in global training data. The smart home device must adapt to seasonal changes and family dynamics that vary dramatically across households. The autonomous vehicle faces local road conditions, weather patterns, and traffic behaviors that differ from its original training environment.

These scenarios require on-device learning, where models train and adapt directly on the devices where they operate.[1] The consequence is a fundamental shift: machine learning moves from centralized training to distributed learning across millions of heterogeneous devices, each operating under unique constraints and local conditions. Within the fleet stack, distributed learning pushes the physical layer to its thermodynamic limits.

Local adaptation can only happen on hardware that already clears the inference floor, because a gradient update costs strictly more than the forward pass it builds on. Specialized accelerators set that floor: they determine whether an edge device has enough latency and energy margin to run intelligent workloads at all, and therefore whether any local training is even possible.

Systems Perspective 11.1: The silicon dividend of a dedicated NPU

The gap between running a model on a generic mobile CPU and on a dedicated Neural Processing Unit (NPU) is best understood as a "silicon dividend" rather than a marginal speedup. Published NPU profiles vary by silicon generation and workload, but for a MobileNet-class classifier the representative figures anchor the chapter's argument: a dedicated NPU runs the forward pass on the order of 20× faster than the same model on a mobile CPU, and at

[1] **A11 Bionic (2017)**: Apple's first SoC with a dedicated Neural Engine, rated up to 600 billion operations per second for machine-learning inference workloads. The systems consequence was establishing the mobile neural processing unit (NPU) as a distinct power domain: on-device adaptation becomes more practical when inference-optimized silicon frees thermal headroom for short local training or personalization bursts.

roughly 60× better energy efficiency. The CPU baseline here is computed for MobileNetV2 at low utilization (1.711 ms per inference); the NPU figures are illustrative targets for this class of accelerator, not a measured profile.

What matters for system design is the *kind* of difference these ratios represent. Specialized hardware makes the model feasible, not merely faster: an energy gain of this magnitude is the difference between a workload that drains the battery in minutes and one that can run continuously in the background. On the edge, the CPU is for coordination and the NPU is for survival. Falling back to the CPU when the NPU target is missed does more than cost performance; it typically renders the application unusable through rapid battery drain and thermal throttling.

The transition to on-device learning introduces fundamental tension in machine learning systems design. While cloud-based architectures use abundant computational resources and controlled operational environments, edge devices must function within severely constrained resource envelopes. Memory is measured in megabytes rather than gigabytes; power budgets are measured in milliwatts rather than watts; and network connectivity may be intermittent or entirely absent. When a local model operates in a safety-critical loop, these constraints become life-safety engineering problems.

These constraints run through Archetype C (Federated MobileNet) (section 1.6.1.3), where autonomy demands that learning happen on-device under severe resource limits. Archetype C represents the extreme end of this spectrum. Operating on milliwatt power budgets, it cannot afford the energy cost of transmitting raw data to the cloud. Data locality is therefore a physical necessity imposed by the power wall, not merely a privacy feature. Quantized training, sparse updates, and federated coordination are the survival strategies that allow Archetype C to exist.

Navigating this architectural tension requires the compression techniques carried over from inference, combined with algorithmic techniques, design patterns, and system principles that enable effective learning under extreme resource constraints. The challenge extends beyond conventional optimization of training algorithms: it requires rethinking the entire machine learning pipeline for deployment environments where traditional computational assumptions fail.

Definition 11.1: On-device learning

On-Device Learning is the local training or adaptation of machine learning models directly on deployed hardware without requiring server connectivity.

1. **Significance**: It enables Hyper-Personalization and autonomous operation under severe resource constraints. Within the iron law, on-device learning must minimize the total energy consumed per update, because every gradient step draws on limited battery power and competes with other system tasks for the available compute throughput (R_{peak}).
2. **Distinction**: Unlike Edge Inference, which only executes a fixed model, on-device learning involves a Local Optimization Loop (forward and backward passes) that requires significantly more memory and compute.
3. **Common pitfall**: A frequent misconception is that on-device learning requires training from scratch. In reality, it is almost always On-Device Adaptation: fine-tuning a small subset of a pretrained base model's parameters to fit the specific data distribution of a local environment.

The consequences reach beyond technical optimization into every phase of the machine learning lifecycle. Models transition from predictable versioning patterns to continuous divergence and adaptation trajectories. Performance evaluation shifts from centralized monitoring dashboards to distributed assessment across heterogeneous user populations. Privacy preservation becomes a core architectural requirement that shapes system design decisions rather than a downstream compliance concern.

11.1.1 Motivations and benefits

Machine learning systems have traditionally relied on centralized training pipelines. Models are developed and refined using large, curated datasets and cloud-based infrastructure (Dean et al. 2012). Once trained, these models are deployed to client devices for inference, creating a clear separation between the training and deployment phases. While this architectural separation has served many use cases well, it imposes significant limitations in applications where local data is dynamic, private, or highly personalized.

On-device learning challenges this established model by enabling systems to train or adapt directly on the device, without relying on constant connectivity to the cloud. The shift reflects changing application requirements and user expectations that demand responsive, personalized, and privacy-preserving[2] machine learning systems.

Consider a smartphone keyboard adapting to a user's unique vocabulary and typing patterns. To personalize predictions, the system must perform gradient updates on a compact language model using locally observed text input. A single gradient update for even a minimal language model requires 50 MB–100 MB of memory for activations and optimizer state. Smartphone operating systems often leave only 200 MB–300 MB for background applications like keyboards, with the exact budget varying by OS, device generation, and foreground workload. This razor-thin margin demonstrates the central engineering challenge of on-device learning: a single training step can consume 25 percent of available memory. The system must achieve meaningful personalization while operating within constraints so severe that traditional training approaches become architecturally infeasible. This quantitative reality drives the need for specialized techniques that make adaptation possible within extreme resource limitations.

Four key considerations motivate the shift from centralized to decentralized learning (T. Li et al. 2020): personalization, latency and availability, privacy, and infrastructure efficiency. Personalization represents the most compelling motivation, as deployed models often encounter usage patterns and data distributions that differ from their training environments. Local adaptation allows models to refine behavior in response to user-specific data, capturing linguistic preferences, physiological baselines, sensor characteristics, or environmental conditions. This capability is essential in applications with high inter-user variability, where a single global model cannot serve all users effectively.

Latency and availability constraints provide additional justification for local learning. In edge computing scenarios, connectivity to centralized infrastructure may be unreliable, delayed, or intentionally limited to preserve bandwidth or reduce energy consumption. On-device learning enables autonomous improvement of models even in fully offline or delay-sensitive contexts, where round-trip updates to the cloud are architecturally infeasible.

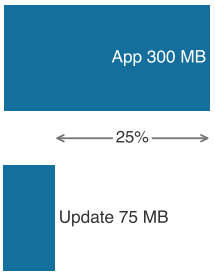
Privacy considerations provide a third compelling driver. Many applications involve sensitive or regulated data including biometric measurements, typed input, location traces, or health information. Local learning can reduce privacy exposure by keeping raw data on the device and limiting transmission to centralized systems. This can simplify compliance engineering, but it does not by itself establish adherence to regulations such as the General Data Protection Regulation (GDPR),[3] the Health Insurance Portability and Accountability Act (HIPAA)[4] (U.S. Department of Health and Human Services 2026), or region-specific data sovereignty laws.

Infrastructure efficiency provides economic motivation for distributed learning approaches. Centralized training pipelines require substantial backend infrastructure to collect, store, and process user data from potentially millions of devices. By shifting learning to the edge, systems reduce communication costs and distribute training workloads across the deployment fleet, relieving pressure on centralized resources while improving scalability.

11.1.2 Alternative approaches and decision criteria

On-device learning demands significant engineering investment that may not always be justified. Simpler alternatives can often achieve comparable results with lower operational overhead, and premature adoption introduces complexity without proportional value.

Table 11.1 compares three alternatives that often satisfy personalization and adaptation requirements without local training complexity.



On-device learning hits the memory wall before the compute wall.

2 **Privacy-Preserving (Differential Privacy)**: While federated learning avoids moving raw data, research has shown that raw gradients can be "inverted" to reconstruct input samples. Differential privacy adds calibrated noise to bound the privacy loss contributed by any one record or user under a stated adjacency definition; it reduces inference risk, but the strength of the guarantee depends on ε, δ, composition, and implementation quality.

3 **GDPR (General Data Protection Regulation)**: Effective May 2018, GDPR requires a lawful basis, data minimization, purpose limitation, and rights-handling processes for personal-data processing. Explicit consent may be required in sensitive contexts, but it is not the only lawful basis. On-device learning can reduce the amount of personal data transferred to centralized systems and can simplify some cross-border and retention questions, but deployments still need consent, rights, and governance mechanisms appropriate to the data and use case.

4 **HIPAA (Health Insurance Portability and Accountability Act)**: Requires covered entities and business associates to apply administrative, physical, and technical safeguards for protected health information (PHI). Cloud services that create, receive, maintain, or transmit electronic PHI on behalf of a covered entity or business associate generally require an appropriate Business Associate Agreement and Security Rule controls. On-device learning can reduce PHI transmission and vendor-management risk, but it does not remove HIPAA obligations for in-scope healthcare deployments.

Table 11.1: Alternatives to On-Device Learning: Simpler personalization mechanisms can satisfy many adaptation requirements without local gradient updates, so on-device learning should be reserved for cases where privacy, latency, connectivity, or measured quality requirements rule out these options.

Alternative	Mechanism	When it fits
Feature-based personalization	Store user preferences, interaction history, and behavioral features locally, then feed those features into a static model rather than adapting model weights.	News recommendation systems can store topic preferences and reading patterns locally, then combine those features with a centralized content model.
Cloud-based fine-tuning with privacy controls	Centralize adaptation while processing user data in batches during off-peak hours with privacy-preserving techniques such as differential privacy, federated analytics, secure aggregation (Bonawitz et al. 2017), or combinations of these controls.	This approach often achieves higher accuracy than resource-constrained on-device updates while maintaining acceptable privacy properties for many applications.
User-specific lookup tables	Combine global models with personalized retrieval mechanisms by maintaining a lightweight table for frequently accessed local patterns.	Personalization benefits are needed with minimal computational and storage overhead.

In the cloud-fine-tuning row, differential privacy[5] is the mechanism that bounds information leakage while adding noise that can require more rounds or more data.

The decision to implement on-device learning should be driven by quantifiable requirements that preclude these simpler alternatives. True data privacy constraints that legally prohibit cloud processing, genuine network limitations that prevent reliable connectivity, quantitative latency budgets that preclude cloud round-trips, or demonstrable performance improvements that justify the operational complexity represent legitimate drivers for on-device learning adoption.

For applications with critical timing requirements, network round-trip times make cloud-based alternatives architecturally infeasible. Camera processing under 33 ms, voice response under 500 ms, AR/VR motion-to-photon latency under 20 ms, or safety-critical control under 10 ms all face network round-trip times typically ranging from 50 to 200 ms. In such scenarios, on-device learning becomes necessary regardless of complexity considerations. Teams should thoroughly evaluate simpler solutions before committing to the significant engineering investment that on-device learning requires.

5 **Differential Privacy (DP)**: A mathematical framework that bounds information leakage by adding calibrated noise to computations. In edge learning, that noise becomes a systems cost because noisier updates usually require more rounds or more data to reach the same utility. Chapter 13 develops the formal privacy-budget machinery.

11.1.3 Knowledge transfer as the adaptation foundation

These motivations are grounded in the broader concept of knowledge transfer, where a pretrained model transfers useful representations to a new task or domain. This foundational principle makes on-device learning both feasible and effective, enabling sophisticated adaptation with minimal local resources. Figure 11.1 illustrates how knowledge transfer occurs between closely related tasks such as playing different board games or musical instruments, or across domains that share structure such as from riding a bicycle to driving a scooter. In on-device learning, a model pretrained in the cloud adapts efficiently to a new context using only local data and limited updates, allowing fast adaptation without relearning from scratch even when the new task diverges in input modality or goal.

This conceptual shift, enabled by transfer learning and adaptation, enables real-world on-device applications. Whether adapting a language model for personal typing preferences, adjusting gesture recognition to individual movement patterns, or recalibrating a sensor model in changing environments, on-device learning allows systems to remain responsive, efficient, and user-aligned over time.

11.1.4 Real-world application domains

The motivations for on-device learning manifest concretely across consumer technologies, healthcare, industrial systems, and embedded applications, each domain presenting scenarios where personalization, latency, privacy, and infrastructure efficiency become essential. Mobile input prediction is a well-documented example of on-device learning. In systems such as smartphone keyboards, predictive text and autocorrect features benefit substantially from continuous local adaptation.

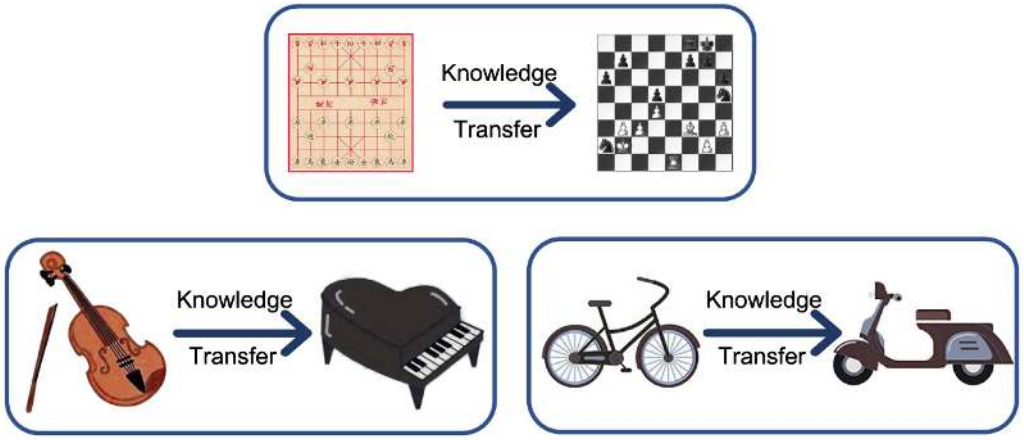


Figure 11.1: Knowledge Transfer: Pretrained models accelerate learning on new tasks by using existing representations, as seen by adapting skills between related board games or musical instruments. This transfer extends across domains like bicycle riding and scooter operation, where shared underlying structures allow efficient adaptation with limited new data.

User typing patterns are highly personalized and evolve dynamically, making centralized static models insufficient for high-quality user experience. On-device learning allows language models to fine-tune their predictions directly on the device, achieving personalization while maintaining data locality. Google's Gboard[6], for instance, employs federated learning[7] (each device shares model updates, never the underlying data) to improve shared models across a large population of users while keeping raw data local to each device (Hard et al. 2018).

Figure 11.2 demonstrates how different prediction strategies enable local adaptation in real time. Next-word prediction suggests likely continuations based on prior text, while Smart Compose uses on-the-fly rescoring to offer dynamic completions. These techniques demonstrate the sophistication of local inference mechanisms.

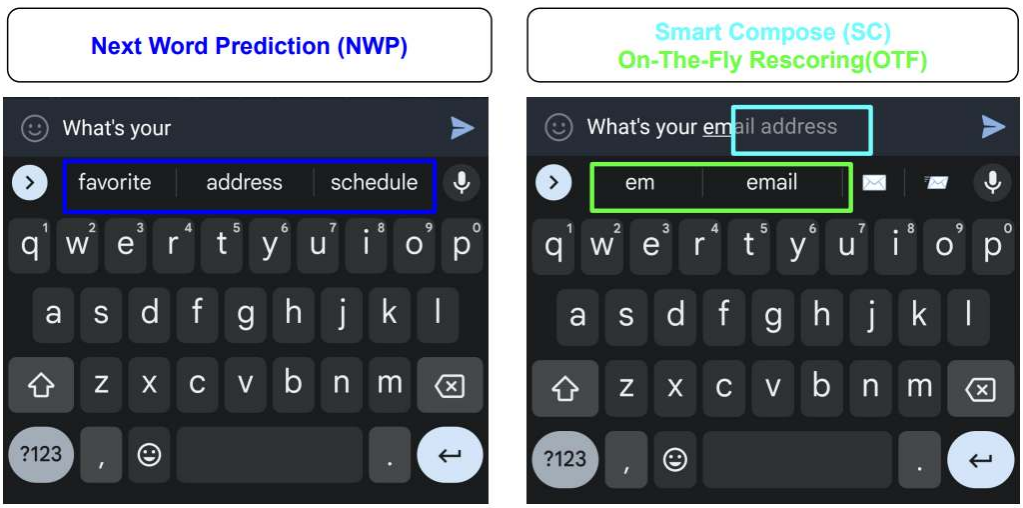


Figure 11.2: On-Device Prediction Strategies: Gboard employs both next-word prediction and smart compose with on-the-fly rescoring to adapt to user typing patterns locally, enhancing personalization and preserving privacy. These techniques demonstrate how machine learning models can refine predictions in real time without transmitting data to a central server, enabling efficient and private mobile input experiences.

6 | **Gboard (2017)**: One of the first widely reported commercial federated learning deployments at mobile scale. Published reports showed that keyboard models could improve from aggregated on-device updates while raw typed text remained local. The systems consequence is durable: compressed updates and privacy mechanisms make cross-fleet learning possible over constrained mobile networks without centralizing raw keystrokes.

7 | **Federated Learning**: Named by direct analogy to a political federation (for example, the United States), where individual states (devices) maintain *local autonomy* over their data while participating in a *global union* to improve a shared model. This decentralization is one mechanism for reducing raw data movement and supporting data residency or regulated-data requirements, but legal compliance still depends on the deployment context, governance controls, and privacy guarantees around model updates.

Wearable and health monitoring devices present equally compelling use cases with additional regulatory constraints. These systems rely on real-time data from accelerometers, heart rate sensors, and electrodermal activity monitors to track user health and fitness. Physiological baselines vary dramatically between individuals, creating a personalization challenge that static models cannot address effectively. On-device learning allows models to adapt to these individual baselines over time, substantially improving the accuracy of activity recognition, stress detection, and sleep staging while meeting regulatory requirements for data localization.

Voice interaction technologies present another important application domain with unique acoustic challenges. Wake-word detection[8] and voice interfaces in devices such as smart speakers and earbuds must recognize voice commands quickly and accurately even in noisy or dynamic acoustic environments.

These systems face strict latency requirements. Voice interfaces must maintain end-to-end response times under 500 ms to preserve natural conversation flow, with wake-word detection requiring sub-100 ms response times to avoid user frustration. Local training allows models to adapt to the user's unique voice profile and changing ambient context, reducing false positives and missed detections while meeting these demanding performance constraints. This adaptation is particularly valuable in far-field audio settings, where microphone configurations and room acoustics vary dramatically across deployments.

Beyond consumer applications, industrial IoT and remote monitoring systems demonstrate the value of on-device learning in resource-constrained environments. In applications such as agricultural sensing, pipeline monitoring, or environmental surveillance, connectivity to centralized infrastructure may be limited, expensive, or entirely unavailable. On-device learning allows these systems to detect anomalies, adjust thresholds, or adapt to seasonal trends without continuous communication with the cloud. This capability is necessary for maintaining autonomy and reliability in edge-deployed sensor networks, where system downtime or missed detections can have significant economic or safety consequences.

The most demanding applications emerge in embedded computer vision systems including robotics, AR/VR, and smart cameras. These combine complex visual processing with the tightest end of the latency budgets established in section 11.1.2: the 33 ms frame budget for 30 FPS cameras, the sub-20 ms motion-to-photon budget that prevents AR/VR nausea, and the sub-10 ms bound on safety-critical control. These systems also operate in novel or rapidly changing environments that differ substantially from their original training conditions. **On-device adaptation** allows models to recalibrate to new lighting conditions, object appearances, or motion patterns while meeting these critical latency budgets that fundamentally drive the architectural decision between on-device vs. cloud-based processing.

Each domain reveals a common pattern. Deployment environments introduce variation and context-specific requirements that cannot be anticipated during centralized training. These applications demonstrate how the motivational drivers manifest as concrete engineering constraints. Mobile keyboards face memory limitations for storing user-specific patterns. Wearable devices encounter energy budgets that restrict training frequency. Voice interfaces must meet sub-100 ms latency requirements that preclude cloud coordination. Industrial IoT systems operate in network-constrained environments that demand autonomous adaptation. This pattern illuminates the fundamental design requirement shaping all subsequent technical decisions. Learning must be performed efficiently, privately, and reliably under significant resource constraints. Section 11.2 analyzes these constraints systematically, section 11.3 presents techniques for adapting models within tight resource envelopes, and section 11.5 establishes protocols for privacy-preserving coordination across device populations.

11.1.5 Architectural trade-offs: Centralized vs. decentralized training

The diversity of these applications reveals how fundamentally on-device learning differs from traditional ML architectures, extending beyond deployment choices into a complete reimagining of the training lifecycle. Many machine learning systems follow a centralized learning paradigm: models train in data centers using large-scale, curated datasets aggregated from many sources, deploy to client devices in static form for inference without further modification, and receive updates periodically through offline retraining using newly collected or labeled data sent back from the field. This centralized model offers proven advantages, including high-performance computing

8 **Wake-Word Detection**: Always-on keyword spotting commonly runs in a sub-milliwatt to low-milliwatt envelope, far below full speech recognition, using compact neural networks optimized for sub-100 ms latency and very low false-activation rates. This extreme power constraint defines the design space: the model must be small enough to run on a dedicated always-on processor, making on-device personalization critical for reducing false activations without increasing model size.

infrastructure, access to diverse data distributions, and robust debugging and validation pipelines, but it also depends on assumptions that edge deployments often violate: reliable data transfer, trust in data custodianship, and infrastructure capable of managing global updates across device fleets.

On-device learning inverts these assumptions. Each device maintains its own model copy and adapts it locally using data unavailable to centralized infrastructure. Training occurs asynchronously under varying resource conditions driven by device usage patterns, battery levels, and thermal states. Raw data remains on the device, reducing privacy exposure but complicating coordination. Hardware capabilities, runtime environments, and usage patterns vary dramatically across devices, making the learning process heterogeneous and difficult to standardize.

Decentralization introduces a new class of systems challenges. Devices may operate with different model versions, leading to behavioral inconsistencies across the deployment fleet. Evaluation and validation grow more complex without a central point from which to measure performance (McMahan et al. 2017). Model updates must be carefully managed to prevent degradation, and safety guarantees become harder to enforce without centralized testing infrastructure.

Managing thousands of heterogeneous edge devices exceeds typical distributed systems complexity. Device heterogeneity extends beyond hardware differences to include varying operating system versions, security patches, network configurations, and power management policies. Large-scale federated systems must tolerate clients that fail eligibility checks, arrive late, or drop out during training rounds (Bonawitz et al. 2019). Others have been disconnected for weeks or months, creating persistent coordination challenges.

When disconnected devices reconnect, they require state reconciliation to avoid version conflicts. Update verification becomes critical as devices can silently fail to apply updates or report success while running outdated models. Robust systems implement multi-stage verification. Cryptographic signatures confirm update integrity, functional tests validate model behavior, and telemetry confirms deployment success. Rollback strategies must handle partial deployments where some devices received updates while others remain on previous versions. Maintaining system consistency during failure recovery requires sophisticated orchestration that draws on distributed systems principles while introducing edge-specific complexities.

Despite these challenges, decentralization enables deep personalization without centralized oversight, supports learning in disconnected or bandwidth-limited environments, and reduces the operational cost of model updates. The central question is how to coordinate learning across devices, and that question unfolds across three operational phases: centralized training, local adaptation, and federated coordination.

The traditional centralized paradigm begins with cloud-based training on aggregated data, followed by static model deployment to client devices. This approach works well when data collection is feasible, network connectivity is reliable, and a single global model can serve all users effectively. However, it breaks down when data becomes personalized, privacy-sensitive, or collected in environments with limited connectivity.

Once deployed, local differences begin to emerge as each device encounters its own unique data distribution. Devices collect data that reflects individual user patterns, environmental conditions, and usage contexts. This data is often non-i.i.d.[9] and noisy, requiring local model adaptation to maintain performance. This transition marks the shift from *global generalization* to *local specialization*.

The final phase introduces federated coordination, where devices periodically synchronize their local adaptations through aggregated model updates rather than raw data sharing. This enables privacy-preserving global refinement while maintaining the benefits of local personalization.

Figure 11.3 traces this evolution from centralized training through local adaptation to federated coordination. Each phase increases coordination complexity while enabling capabilities impossible in purely centralized deployments.

Decentralized, continuous learning rewrites the rules of system design. We are no longer operating in climate-controlled data centers with effectively infinite power; we must confront the severe physical and algorithmic limits of edge devices.

11.2 Design Constraints

Attempting to fine-tune a neural network on a smartphone is a thermodynamic battle. A typical GPU training cluster consumes megawatts of power, while a smartphone must execute backpropagation

[9] **Non-IID (Not Independent and Identically Distributed)**: Data where samples are not drawn from a single common distribution, the default condition in federated learning where each device reflects a single user's patterns. Non-IID label and feature distributions can sharply degrade federated averaging compared with IID baselines (Zhao et al. 2018), forcing systems to adopt techniques like local adaptation layers or clustered aggregation to maintain convergence.

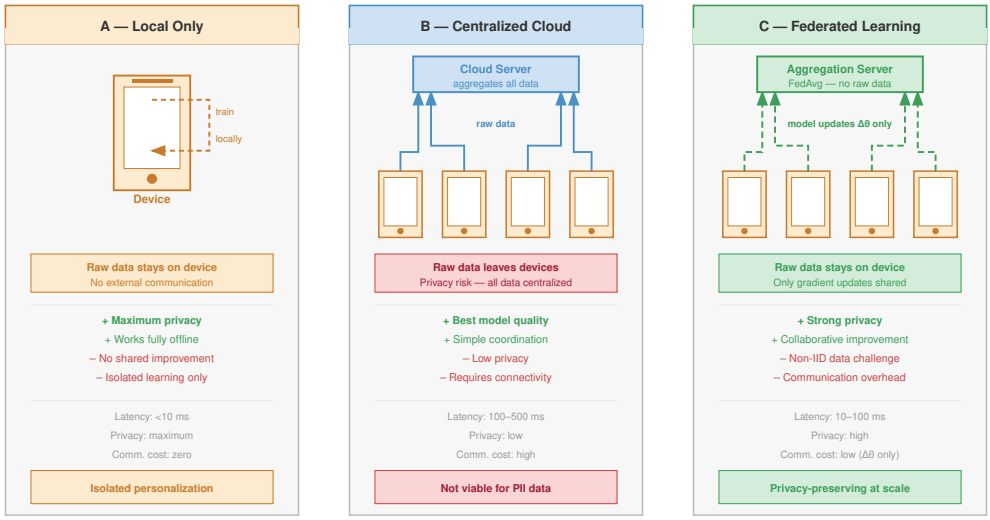


Figure 11.3: Local Only to Centralized Cloud to Federated Learning: The evolution of distributed learning unfolds in three phases: (A) Local Only, where a single device trains on its own data with no external communication; (B) Centralized Cloud, where multiple devices send raw data to a cloud server for aggregation (simple but privacy-risky); and (C) federated learning, where devices share only model updates ($\Delta\theta$) with a Federated Averaging (FedAvg) aggregation server, preserving privacy at scale. Each phase trades coordination cost for privacy and collaboration benefits.

on a 10 W thermal budget without burning the user's hand or draining the battery in five minutes. These severe constraints force us to radically redesign our learning algorithms to be hyper-efficient in compute, memory, and data.

The constraints on parameters, operations, and data interact *multiplicatively* rather than *additively*, creating a constrained optimization problem far more challenging than inference-only deployment. The most important shift is in what compression is for: quantization shrinks bytes per weight, pruning removes unnecessary parameters, and knowledge distillation transfers behavior into a smaller architecture, and on-device learning turns these tools from optional optimizations into baseline feasibility requirements. This compression baseline, established here, is what later sections build on rather than re-derive.

On-device learning operates under the same efficiency constraints as inference but with training-specific amplifications that make optimization far more demanding. Where inference requires a single forward pass through the network, training demands forward propagation, gradient computation through backpropagation, and weight updates, increasing memory requirements by 4–12$\times$ and computational costs by 2–3$\times$ when activations, gradients, and optimizer state are included. These amplifications are why the compression baseline is non-negotiable at the edge: training within these device constraints would be impossible without it.

The fundamental engineering challenges that shape on-device learning implementation follow directly from these motivations. Enabling learning on the device requires completely rethinking conventional assumptions about where and how machine learning systems operate. In centralized environments, models are trained with access to extensive compute infrastructure, large and curated datasets, and generous memory and energy budgets. At the edge, none of these assumptions hold, creating a fundamentally different design space.

These constraints define a feasible region rather than a checklist. Model compression determines how little of the algorithm can change. Sparse, nonuniform data determines how much signal each local update can extract. Limited compute determines when training can run at all. The interaction among these dimensions sets the later choice among adaptation, data-efficiency, and federated-coordination techniques.

11.2.1 Quantifying training overhead on edge devices

Beyond the compression baseline, training amplifies memory footprint, memory bandwidth, and hardware utilization pressures in ways that are specific to local adaptation, and these pressures compound rather than add. Table 11.2 quantifies how compute operations grow 2–3× and energy consumption can balloon 10–50×, while figure 11.4 shows a representative 9× Adam-style memory budget within that broader range.

Table 11.2: Training Amplifies Inference Constraints: On-device learning operates under the same efficiency constraints as inference but with training-specific amplifications that make optimization dramatically more challenging. This table quantifies how each constraint dimension intensifies when transitioning from running pretrained models to adapting them locally. Amplification factors assume standard backpropagation without optimizations like gradient checkpointing.

Constraint Dimension	Inference	Training Amplification	Impact on Design
Memory Footprint	Model weights + single activation map	Weights + full activation cache + gradients + optimizer state	4–12× increase; forces aggressive compression
Compute Operations	Forward pass only	Forward + backward + weight update	2–3× increase; limits model complexity
Memory Bandwidth	Sequential weight reads	Bidirectional data flow for gradients	5–10× increase; creates bottlenecks
Energy per Sample	Single inference operation	Multiple gradient steps with convergence	10–50× increase; requires opportunistic scheduling
Data Requirements	Precollected, curated datasets	Sparse, noisy, streaming local data	Necessitates sample-efficient methods
Hardware Utilization	Optimized for forward passes	Different access patterns for backprop	Inference accelerators may not help training

As figure 11.4 shows, a representative Adam-style training budget can reach a 9× memory footprint increase across model scales; the broader 4–12× range depends on optimizer choice, batch size, activation checkpointing, and how many layers are updated.

11.2.1.1 Peak memory usage

Memory consumption during training is not static. It fluctuates dynamically, reaching a maximum during the backward pass when activations, gradients, and optimizer states must coexist. This peak memory usage determines whether a model can be trained on a device. For a 10M parameter model on a smartphone with 8 GB RAM, the 40 MB of FP32 weights might spike to over 200 MB during backpropagation as activations, gradients, and optimizer states accumulate—competing directly with the operating system and foreground applications for the device's limited memory. Techniques like gradient checkpointing mitigate this by discarding intermediate activations during the forward pass and recomputing them on-demand during the backward pass (Chen et al. 2016). This approach trades extra computation for lower peak memory; the exact savings depend on which activations are checkpointed and how much recomputation the device can tolerate.

These amplifications explain why standard optimization techniques fail when applied to training workloads without modification. Each constraint category shapes on-device learning system design, requiring approaches that build on but extend beyond inference-focused methods.

Checkpoint 11.1: On-device training constraints

Verify your understanding of how training amplifies edge constraints:

- ☐ **Training memory blowup**: Explain why on-device training typically requires 4× to 12× more memory than on-device inference for the same model.
- ☐ **Starved NPU**: Explain how a fast NPU can still sit idle when the training pipeline cannot keep data and activations moving within the device's memory budget.
- ☐ **Backward-pass OOM**: Explain why a 10M parameter model can still trigger out-of-memory errors during the backward pass on a smartphone with 8 GB of RAM.

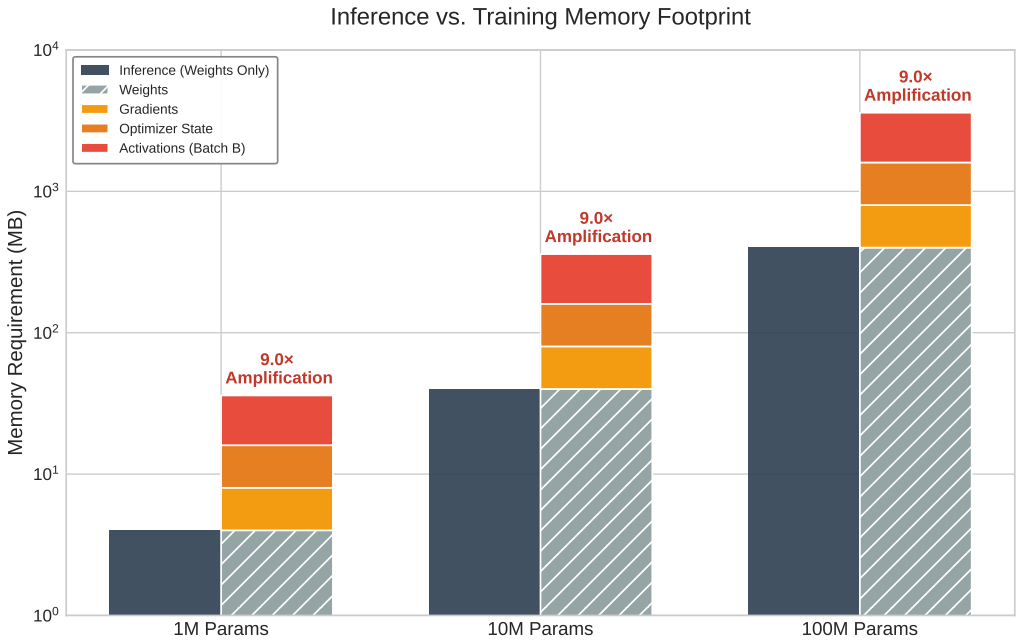


Figure 11.4: The Training Memory Amplifier: Comparison of memory requirements for Inference vs. Training across model scales (1M to 100M parameters). This illustrative configuration stores weights, gradients, optimizer state, and activations, producing a representative 9× memory footprint increase.

☐ **Checkpointing energy**: Evaluate the claim that gradient checkpointing reduces the total energy consumed by on-device training.

Figure 11.5 illustrates how the complete training pipeline combines offline pretraining with online adaptive learning on resource-constrained IoT devices. The system first undergoes meta-training with generic data. During deployment, device-specific constraints such as data availability, compute, and memory shape the adaptation strategy by ranking and selecting layers and channels to update. This selective fine-tuning allows efficient on-device learning within limited resource envelopes.

11.2.2 Model constraints

The structure and size of the machine learning model directly determine whether on-device training is possible. Cloud-deployed models can span billions of parameters and rely on multi-gigabyte memory budgets; models intended for on-device learning must conform to tight constraints on memory, storage, and computational complexity. These constraints tighten further during training, where gradient computation, parameter updates, and optimizer state management all demand additional resources beyond inference.

The scale of these constraints becomes concrete across the device spectrum. MobileNetV2, commonly used in mobile vision tasks, requires approximately 14 MB of storage in its standard configuration. While feasible for smartphones with gigabytes of available RAM, this far exceeds the 256 KB of SRAM and 1 MB of flash storage on microcontrollers such as the Arduino Nano 33 BLE Sense[10]. On such severely constrained platforms, even a single convolutional layer may exceed available RAM during training due to intermediate feature maps and gradient storage.

The training process itself dramatically expands the effective memory footprint. Standard backpropagation caches activations for each layer during the forward pass, then reuses them during gradient computation in the backward pass. The constraint-amplification analysis established that this activation caching multiplies memory requirements compared to inference-only deployment. A

10 **Arduino Nano 33 BLE Sense**: With 256 KB SRAM, roughly 65,536× smaller than a flagship smartphone's 16 GB, a single 224×224×3 RGB image occupies ~151 KB and consumes roughly 60 percent of available memory. Activation and gradient storage add several more live tensors, and optimizer state can push the training footprint far beyond the inference footprint. This means even a tiny convolutional neural network (CNN) layer can exceed total SRAM during backpropagation. This memory wall forces 8-bit or 4-bit quantization as a prerequisite, not an optimization.

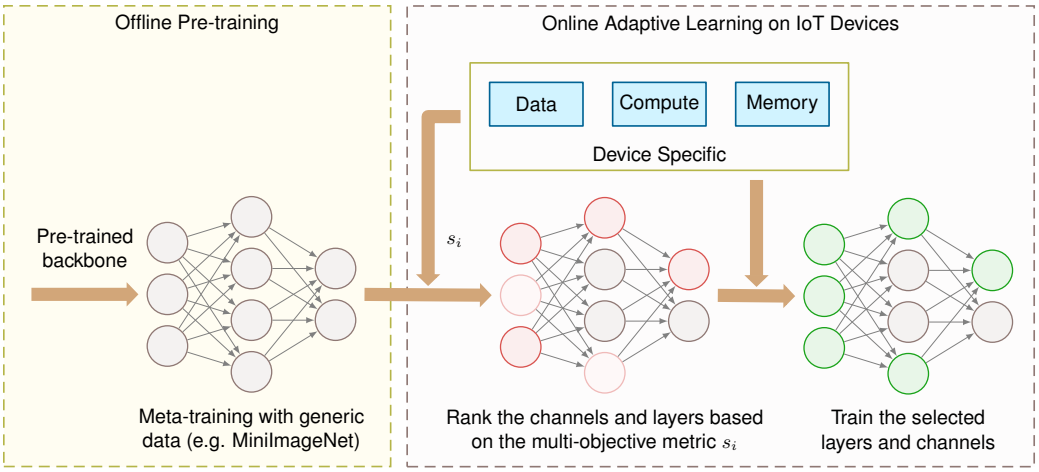


Figure 11.5: Resource-constrained devices use a two-stage learning process. Offline pretraining establishes initial model weights. Online adaptation then selectively updates layers based on available data, compute, and memory. This approach balances model performance with the practical limitations of edge deployment, enabling continuous learning in real-world environments.

seemingly modest 10-layer convolutional model processing 64×64 images may require 1 to 2 MB, well beyond the SRAM capacity of most embedded systems.

Model complexity also directly affects runtime energy consumption and thermal limits. In smartwatches or battery-powered wearables, sustained model training can rapidly deplete energy reserves or trigger thermal throttling that degrades performance. Training a full model using floating-point operations on these devices is often infeasible from an energy perspective, even when memory constraints are satisfied. Ultra-lightweight benchmarks such as MLPerf Tiny provide small quantized inference targets for severely constrained devices (Banbury et al. 2021). On-device adaptation techniques then reduce training cost by freezing most parameters, reducing activation storage, or selecting sparse update sets (Cai, Gan, Zhu, et al. 2020; Kwon et al. 2024).

The practical implications of battery and thermal constraints extend beyond just limiting training duration. Mobile devices must carefully balance training opportunities with user experience. Aggressive on-device training can cause noticeable device heating and rapid battery drain, leading to user dissatisfaction and potential app uninstalls. Smartphone ML workloads commonly operate within a sustained processing envelope of 2–3 W to prevent thermal discomfort, though they can burst to 5–10 W for brief periods before thermal throttling kicks in. Training even modest models can easily exceed these sustainable power limits. This reality necessitates intelligent scheduling strategies: training during charging periods when thermal dissipation is improved, using low-power cores for gradient computation when possible, and implementing thermal-aware duty cycling that pauses training when temperature thresholds are exceeded. Some systems even use device usage patterns, scheduling intensive adaptation only during overnight charging when the device is idle and connected to power.

These constraints demand that model architectures be designed for on-device learning from the outset. Large transformers and deep convolutional networks are simply not viable for on-device adaptation without partitioning, quantization, or offloading. Specialized lightweight architectures such as MobileNets[11], SqueezeNet (Iandola et al. 2016), and EfficientNet (Tan and Le 2019) address resource-constrained inference through mechanisms such as depthwise separable convolutions[12], bottleneck/fire modules, and compound model scaling. Quantization and selective updates are then separate deployment and adaptation choices layered on top of the architecture.

Modularity is a key design property. MobileNets (Howard et al. 2017) and MobileNetV2 (Sandler et al. 2018) can be configured with different width multipliers and resolution settings to balance performance and resource usage. A complete MobileNetV2 with width multiplier $\alpha_{\text{width}} = 1.0$ has approximately 3.5M parameters (14 MB in FP32). With $\alpha_{\text{width}} = 0.5$, the complete 1000-class model has approximately 2.0M parameters (7.9 MB), while a feature-extractor-only deployment with a

11 | **MobileNet (2017)**: Google's architecture achieved 8–9× FLOPs reduction over standard CNNs through depthwise separable convolutions. The systems consequence: MobileNetV2 runs ImageNet classification in approximately 75 ms on a Pixel phone vs. 1.8 seconds for ResNet-50, crossing the threshold where real-time on-device inference and adaptation become thermally sustainable within a smartphone's 2–3 W power envelope.

12 | **Depthwise Separable Convolutions**: Decomposes a standard convolution into a per-channel depthwise filter followed by a 1×1 pointwise combination. For a 3×3 convolution with 512 input/output channels, this reduces parameters from 2.4 M to about 267 K, an 8.8× reduction. The savings are not just theoretical: this decomposition is what makes real-time CNN inference possible on mobile CPUs, and it equally reduces the memory needed for activation caching during on-device training.

small task-specific head can be as small as 0.69M parameters (2.8 MB), fitting within a 4 MB model budget.

11.2.3 Data constraints

Model architecture determines the memory and computational baseline for on-device learning, but data availability and quality introduce equally fundamental limitations that shape every aspect of the learning process. Data available to on-device ML systems differs dramatically from the large, curated datasets used in cloud-based training. At the edge, data is locally collected, temporally sparse, and often unstructured or unlabeled. The resulting challenges in volume, quality, and statistical distribution directly affect the reliability and generalizability of on-device learning.

Data volume is severely limited by both storage constraints and the sporadic nature of user interaction. A smart fitness tracker may collect motion data only during physical activity, generating relatively few labeled samples per day. If a user exercises for 30 minutes, only a few hundred data points might be available for training, compared to the thousands or millions required for effective supervised learning. The scarcity forces a shift from *data-rich* to *data-efficient* algorithms.

On-device data is also frequently non-IID (Zhao et al. 2018), creating statistical challenges that cloud-based systems rarely encounter. A voice assistant deployed across households encounters wide variation in accents, languages, speaking styles, and command patterns. Smartphone keyboards adapt to individual typing patterns, autocorrect preferences, and multilingual usage that varies widely between users. The heterogeneity complicates both model convergence and the design of update mechanisms that must generalize across devices while maintaining personalization.

Label scarcity compounds the distribution problem. Most edge-collected data is unlabeled by default, requiring systems to learn from weak or implicit supervision signals. A smartphone camera may capture thousands of images throughout the day, but only a few are associated with meaningful user actions (tagging, favoriting, or sharing) that could serve as implicit labels. In many applications, including anomaly detection in sensor data and gesture recognition adaptation, explicit labels may be entirely unavailable, making traditional supervised learning infeasible without alternative methods for weak supervision or unsupervised adaptation.

Data quality introduces further challenges. Embedded systems such as environmental sensors or automotive ECUs experience fluctuations in sensor calibration, environmental interference, or mechanical wear, leading to corrupted or drifting input signals over time. Without centralized validation systems to detect and filter these errors, they silently degrade learning performance.

Privacy and security concerns impose the most restrictive constraints, often making data sharing architecturally *impossible* rather than merely *undesirable*. Sensitive information such as health data, personal communications, or behavioral patterns must be protected from unauthorized access under legal and ethical requirements. On-device learning must therefore rely on techniques that enable local adaptation without ever exposing sensitive information, fundamentally reshaping how learning systems are designed and validated.

11.2.4 Compute constraints

The edge hardware landscape provides the computational substrate for machine learning, spanning from microcontrollers like STM32F4 and ESP32 at the most constrained end, to mobile-class processors with dedicated AI accelerators (Apple Neural Engine, Qualcomm Hexagon, Google Tensor) in the middle, and high-capability edge devices at the upper end. While these devices offer varying levels of inference capabilities (computational throughput, memory bandwidth, and energy efficiency when executing pretrained models), training workloads exhibit fundamentally different computational characteristics that reshape hardware utilization patterns.

On-device learning must operate within computational envelopes that differ from cloud-based training infrastructure by factors of hundreds or thousands in raw capacity. The key difference: backpropagation requires significantly higher memory bandwidth than inference due to gradient computation and activation caching, weight updates create write-heavy access patterns unlike inference's read-only operations, and optimizer state management demands additional memory allocation. Hardware perfectly adequate for *inference* may prove entirely inadequate for *adaptation*, even when updating only a small parameter subset.

At the most constrained end, devices such as the STM32F4[13] or ESP32[14] microcontrollers offer

[13] **STM32F4 Microcontroller**: With 192 KB SRAM, a 168 MHz clock speed, and a single-precision FPU, the main constraint is not the absence of floating point but the tiny memory and energy envelope. A dense layer with 1000 input features and 10 hidden units requires approximately 40.040 KB for weights and biases in FP32, consuming about 20.854 percent of total SRAM before accounting for activations or gradients. At approximately 100 mW active power, even simple gradient updates must be duty-cycled, and INT8 or fixed-point kernels remain preferable when accuracy allows.

[14] **ESP32**: Provides 520 KB SRAM and dual-core 240 MHz processing with built-in Wi-Fi and Bluetooth, making it an inexpensive platform for experiments that combine local adaptation with federated coordination. Its CPU includes floating-point support, but fixed-point and 8-bit kernels are usually preferred for throughput, memory footprint, and energy efficiency; compact 8-bit models can fit in approximately 50 KB, enabling basic on-device adaptation for sensor anomaly detection.

only a few hundred kilobytes of SRAM and limited compute and power budgets (Warden and Situnayake 2020). Although these families include single-precision floating-point support, practical ML deployments commonly rely on quantized or fixed-point kernels for efficiency. Libraries like CMSIS-NN (Lai et al. 2018) provide optimized neural network kernels for Arm Cortex-M processors, achieving 4.6× runtime improvement through fixed-point arithmetic and single instruction, multiple data (SIMD) optimizations. These severe limitations preclude conventional deep learning libraries and require models designed for quantized arithmetic and minimal runtime memory allocation. Even simple models require quantization-aware training[15] and selective parameter updates to execute training loops without exceeding memory or power budgets.

The practical implications are stark: while the STM32F4 microcontroller can run a simple linear regression model with a few hundred parameters, training even a small convolutional neural network would immediately exceed its memory capacity. In these severely constrained environments, neural updates are often limited to simple algorithms such as stochastic gradient descent (SGD)[16], while non-neural adaptation may use k-means clustering to update centroids or thresholds for sensor anomaly detection. Both can be implemented using integer arithmetic and minimal memory overhead, representing a fundamental departure from data-center machine learning practice.

Moving up the computational hierarchy, mobile-class hardware represents improvement but still operates under severe constraints. Platforms including the Qualcomm Snapdragon, Apple Neural Engine[17], and Google Tensor SoC[18] provide significantly more compute power than microcontrollers, often featuring dedicated AI accelerators and optimized support for 8-bit or mixed-precision[19] matrix operations. These accelerators offer dedicated matrix multiplication units, on-chip memory hierarchies, and power management features specifically designed for neural network inference, with varying support for local training workloads. While these platforms can support more sophisticated training routines, including full backpropagation over compact models, they still fall far short of the computational throughput and memory bandwidth available in centralized data centers. For instance, training a lightweight transformer[20] on a smartphone is technically feasible but must be tightly bounded in both time and energy consumption to avoid degrading the user experience, highlighting the persistent tension between learning capabilities and practical deployment constraints.

Published benchmark results from MLPerf Tiny and official vendor data make these hardware tiers concrete. Figure 11.6 plots inference latency against energy per inference for representative devices, revealing three distinct clusters separated by approximately 100× in energy consumption. Dedicated neural processors such as the Syntiant Core 2 and STM32N6 NPU achieve keyword spotting in under 5 ms at 30 to 160 microjoules, while edge GPUs like the Jetson AGX Orin deliver sub-millisecond latency at 15 millijoules. The 100× energy gap between tiers determines which devices can operate on battery power for months vs. hours, fundamentally shaping the feasible design space for on-device learning.

These computational limitations become especially acute in real-time or battery-operated systems, where the latency budgets established in section 11.1.2 turn into hard architectural constraints: they determine whether on-device adaptation is feasible at all or whether cloud-based alternatives become architecturally necessary. The harder constraint is that adaptation cannot have the device to itself. In a smartphone-based speech recognizer, on-device adaptation must seamlessly coexist with primary inference workloads without interfering with response latency or system responsiveness. Similarly, in wearable medical monitors, training must occur opportunistically during carefully managed windows (typically during periods of low activity or charging) to preserve battery life and avoid thermal management issues.

Beyond raw computational capacity, the architectural implications of these hardware constraints extend into fundamental system design choices. Training operations exhibit fundamentally different memory access patterns than inference workloads: backpropagation requires 3–5× higher memory bandwidth due to gradient computation and activation caching, creating bottlenecks that pure computational metrics do not capture. Edge accelerator designs address these challenges through specialized hardware features. Adaptive precision datapaths allow dynamic switching between INT4 for forward passes and FP16 for gradient computation, optimizing both accuracy and efficiency within power budgets. Sparse computation units accelerate selective parameter updates by skipping zero gradients, a capability critical for efficient bias-only and structured low-rank updates

15 | **Quantization-Aware Training (QAT)**: Simulates low-precision arithmetic during training so the model learns robust representations despite reduced precision, unlike post-training quantization which converts a trained FP32 model after the fact (Jacob et al. 2018; Krishnamoorthi 2018). The systems payoff is lower memory traffic and cheaper integer arithmetic: hardware energy models show that lower-precision arithmetic and memory movement can be much cheaper than FP32 computation (Horowitz 2014). Exact speed, energy, and accuracy outcomes depend on the processor, kernel implementation, model, and calibration data. On edge devices, QAT is often a prerequisite for fitting adaptation within thermal and memory budgets.

16 | **SGD (Stochastic Gradient Descent)**: Updates parameters from single-sample or small-batch gradients, storing only current parameters and their gradients. This minimal memory footprint is what makes SGD one of the few plausible optimizers on microcontrollers: Adaptive Moment Estimation (Adam) requires 3× the memory of SGD for its per-parameter moment estimates, exceeding the SRAM budget on devices with $<$512 KB. The trade-off is slower convergence, requiring more gradient steps to reach comparable accuracy.

17 | **Apple Neural Engine**: From the A11 (0.6 TOPS) to flagship-class mobile NPUs around 35 TOPS, mobile neural acceleration improved by roughly two orders of magnitude across that device lineage. The systems consequence: fine-tuning a MobileNet classifier takes approximately 2 seconds on the neural accelerator vs. 45 seconds on CPU in this representative calculation, while consuming only approximately 500 mW additional power. This roughly 23× speedup at minimal power cost is what makes on-device adaptation thermally feasible during normal phone usage, allowing compact personalization jobs to run as short accelerator-assisted bursts within strict thermal and battery policies.

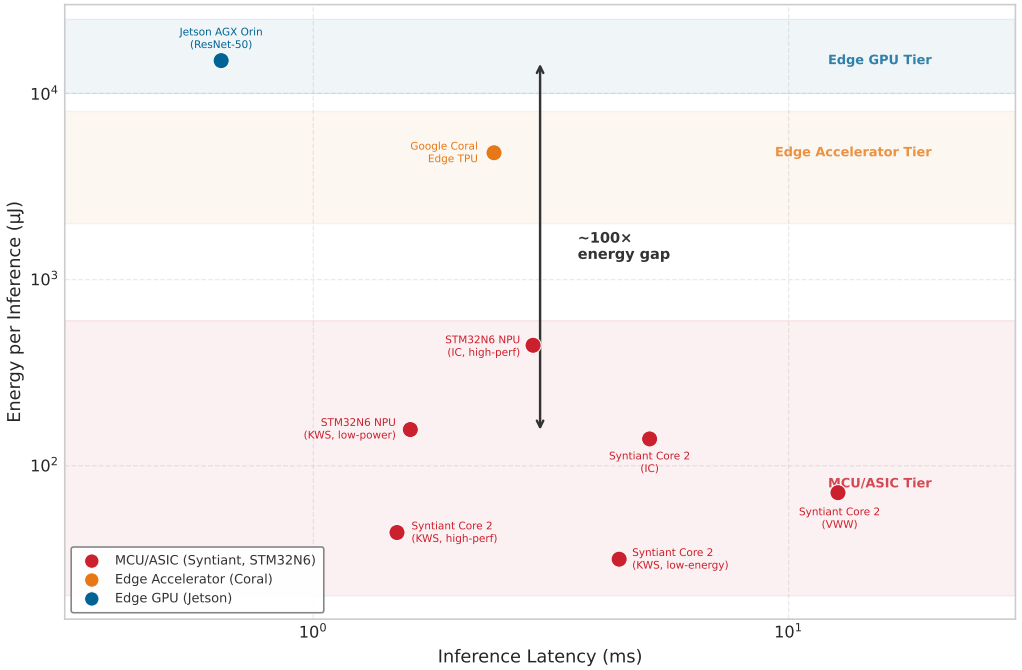


Figure 11.6: The Edge Inference Landscape: Published inference latency and energy measurements from MLPerf Tiny and official benchmarks reveal three distinct hardware tiers separated by approximately 100× in energy consumption. Dedicated neural processors (Syntiant, STM32N6) achieve keyword spotting in under 5 ms at 30-160 microjoules, while edge GPUs (Jetson Orin) deliver sub-millisecond latency at 15 mJ.

(section 11.3.2.2). Near-memory compute architectures[21] reduce data movement costs by performing gradient updates directly adjacent to weight storage, addressing the memory bandwidth bottleneck. However, many edge accelerators remain fundamentally optimized for inference workloads, creating hardware-software co-design opportunities for on-device training accelerators designed to handle the unique demands of local adaptation.

11.2.4.1 The mobile memory wall

While mobile NPUs deliver impressive TOPS, the "memory wall" that section 2.3 examines becomes an impassable barrier for large-scale models on edge devices. The same bandwidth wall that limits training updates is easiest to see in mobile language-model decode: if weights and KV state cannot stream through memory fast enough, no advertised TOPS figure can rescue the workload. Recent analysis highlights how autoregressive decode shifts LLM inference pressure toward memory and interconnect rather than peak arithmetic throughput (Ma and Patterson 2026). The quantitative disparity is stark: data-center HBM3 on an H100 provides 3,350 GB/s, while flagship mobile systems such as A17 Pro or Snapdragon 8 Gen 3 devices sit near 64–100 GB/s of LPDDR5X bandwidth.

A 30–50× bandwidth gap means that even if a model fits in mobile RAM, it will generate tokens 30–50× slower than a data center GPU. On-device large language model (LLM) serving therefore requires aggressive quantization (INT4 or even INT2) as a **bandwidth survival strategy**, not just a capacity optimization. Reducing model size by 8× effectively increases the relative bandwidth, making interactive generation speeds possible on mobile hardware.

> Systems Perspective 11.2: Why bandwidth, not TOPS, is the binding constraint
>
> The bandwidth gap binds because each generated token streams the model's weights from memory to the compute units, so the iron law's data term $(D_{\text{vol}}/\text{BW})$ dominates the compute term whenever weights exceed on-chip cache, which is the case for billion-parameter models

18 **Google Tensor SoC (2021)**: Integrates custom ML acceleration with Android and TensorFlow Lite tooling, making it a useful example of hardware-software co-design for on-device inference and federated workflows. The systems consequence is that accelerator capability alone is insufficient: local adaptation also depends on compiler support, runtime operators, power management, and update orchestration.

19 **Mixed-Precision Training**: Assigns different numerical precisions to different operations: FP16 for forward and backward passes, FP32 for parameter accumulation. This halves memory usage and doubles throughput on hardware with Tensor Cores. Mobile implementations go further, using INT8 for inference and FP16 for gradients, which reduces training memory by 4× compared to full FP32 while keeping accumulation errors bounded through loss scaling.

on mobile-class silicon. That is also why quantization shrinks D_{vol} rather than arithmetic per token.

The practical corollary follows: when ranking edge accelerator options for transformer decoding, the figure of merit is the GB/s a chip can sustain to its on-package memory, not the peak TOPS on its spec sheet. The two can diverge by orders of magnitude on mobile-class silicon.

11.2.5 Edge hardware integration challenges

Beyond the individual constraints of models, data, and computation, on-device learning systems must navigate the underlying physics of mobile computing: power dissipation, thermal limits, and energy budgets. These physical constraints are fundamental design drivers that determine the entire feasible space of on-device learning algorithms.

11.2.5.1 Energy and thermal constraint analysis

Energy and thermal management represent the most challenging aspects of on-device learning system design, as they directly impact user experience and device longevity. Mobile devices operate under strict power budgets that fundamentally determine feasible model complexity and training schedules.

Napkin Math 11.1: Battery drain: The cost of edge learning

Problem: A team is designing a background fine-tuning job for a personalized voice assistant on a smartphone. The training job consumes 4.5 W and takes 30 minutes to complete. If the phone has a 15 Wh battery, how much of the user's battery will this "invisible" update consume?

Math:

1. **Total energy**: 4.5 W $\times$ 0.5 h = 2.25 Wh.
2. **Battery Impact**: 2.25 Wh/15 Wh = 15 percent.

Systems insight: Consuming 15 percent of a user's battery for a background task is a severe violation of mobile UX principles—it is equivalent to losing an hour of screen-on time. This is why on-device training is typically restricted to **Opportunistic Scheduling**: it only runs when the device is plugged in, connected to Wi-Fi, and thermally stable. Designing for the edge means respecting the user's energy budget as strictly as the model's accuracy budget.

This battery calculation is the local edge-learning version of a broader energy constraint: training must fit the user's energy budget, not only the model's accuracy target. A production design therefore has to account for energy at scheduling time, compare local computation against communication cost, and defer learning when the user's device cannot absorb the update.

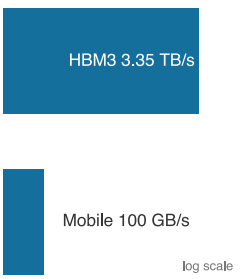
11.2.5.2 Memory hierarchy optimization

Complementing the thermal and power challenges, memory hierarchy constraints create another fundamental bottleneck that shapes on-device learning system design. The constraint-amplification analysis shows that these limitations affect both static model storage and the dynamic memory requirements during training, often pushing systems beyond their practical limits.

The device memory hierarchy spans several orders of magnitude across different device classes, each presenting distinct constraints for on-device learning. A flagship phone provides about 8 GB total system memory, but only part of that remains available for application workloads after accounting for operating system requirements and background processes. Budget Android devices operate with about 4 GB total system memory, leaving roughly 1 GB–2 GB available for ML workloads after OS overhead consumes significant resources. IoT embedded systems provide 64 MB–1 GB total memory that must be shared between system tasks and application data, creating severe constraints for any learning algorithms. Microcontrollers offer only 256 KB–2 MB SRAM, requiring extreme optimization and careful memory management that fundamentally limits the complexity of models that can adapt on such platforms.

[20] **Lightweight Transformers**: Mobile-optimized architectures like MobileBERT achieve 4–6× speedup over full models through knowledge distillation and attention head pruning, retaining 97 percent of BERT-base accuracy at approximately 40 ms inference on mobile CPUs vs. 160 ms for full BERT. The constraint that matters for on-device learning: even these compressed transformers require 50–200 MB for training activations, pushing against the 2–4 GB available on mid-range phones.

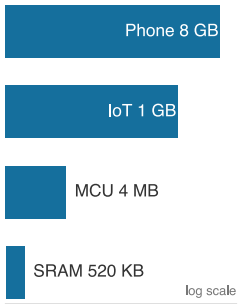
[21] **Near-Memory Computing**: Places processing elements adjacent to or within memory arrays, reducing the data movement that often dominates arithmetic energy in conventional architectures (Horowitz 2014). Near-memory accelerator designs such as TensorDIMM illustrate how moving tensor operations closer to memory can improve throughput and energy behavior for memory-bound deep learning workloads (Kwon and Rhu 2018). For edge training, where gradient computation generates write-heavy access patterns that overwhelm cache hierarchies, this architectural shift could make some forms of on-device backpropagation more practical.



Datacenter HBM outruns mobile memory bandwidth by about 34 times.

The memory expansion during training creates particularly acute challenges that often determine system feasibility. Standard backpropagation requires caching intermediate activations for each layer during the forward pass, which are then reused during gradient computation in the backward pass, creating substantial memory overhead. A MobileNetV2-scale model that fits comfortably for inference can exceed the available memory budget once activations, gradients, and optimizer state are included, making training impossible without aggressive optimization. Quantization, pruning, and distillation can each reduce model footprint or update cost (Jacob et al. 2018; Han et al. 2015; Hinton et al. 2015), but their gains are workload-specific and do not combine automatically. These techniques must be validated together to achieve the compression required for practical deployment.

Cache optimization therefore becomes critical for achieving acceptable performance with constrained memory pools. Modern mobile SoCs feature complex memory hierarchies with L1 cache (32–64 KB), L2 cache (1–8 MB), and system memory (4–16 GB) that exhibit 10–100× latency differences between levels, creating severe performance cliffs when working sets exceed cache capacity. Training workloads that exceed cache capacity face dramatic performance degradation due to memory bandwidth bottlenecks that can slow training by orders of magnitude. Successful on-device learning systems must carefully design data access patterns to maximize cache hit rates, often requiring specialized memory layouts that group related parameters for spatial locality, carefully sized mini-batches that fit entirely within cache constraints, and sophisticated gradient accumulation strategies that minimize expensive memory bus traffic.



Device memory spans about 15,000 times, phone to microcontroller.

The memory bandwidth limitations become particularly acute during training. While inference workloads primarily read model weights sequentially, training requires bidirectional data flow for gradient computation and weight updates. This increased memory traffic can saturate the memory subsystem, creating bottlenecks that limit training throughput regardless of computational capacity. Advanced implementations employ techniques such as gradient checkpointing[22] to trade computation for memory (Chen et al. 2016), and mixed-precision training to reduce bandwidth requirements while maintaining numerical stability.

[22] **Gradient Checkpointing**: Trades computation for memory by recomputing selected intermediate activations during the backward pass instead of storing all of them. The general checkpointing result is sublinear activation memory at the cost of extra forward computation (Chen et al. 2016). On edge devices where memory is the binding constraint, this trade-off can make the difference between a model that fits in 2 GB RAM and one that requires 8 GB, but the exact memory and compute factors are architecture- and schedule-dependent.

11.2.5.3 Mobile AI accelerator optimization

Accelerator choice matters less as a peak-TOPS comparison than as a question of which training primitive the hardware can sustain. Fixed-precision inference engines favor inference-heavy adaptation, programmable vector units make compact backpropagation more plausible, and tight framework integration lowers federated coordination overhead. The architectural differences between these accelerators shape the design space for on-device training algorithms.

Flagship-class mobile accelerators should be read as different adaptation envelopes, not as a peak-TOPS ranking. Flagship-class mobile NPUs provide tens of TOPS of peak performance, with 35 TOPS serving here as a generic high-end reference point rather than a single named chip. Apple's Neural Engine is optimized primarily for CoreML inference patterns with limited training support, making it well suited for inference-heavy adaptation techniques. Qualcomm's Hexagon DSP in the Snapdragon 8 Gen 3 achieves 45 TOPS with flexible precision support and programmable vector units, enabling mixed-precision training workflows that can adapt precision dynamically based on training phase and memory constraints. Google's Tensor TPU in the Pixel 8 is optimized specifically for TensorFlow Lite operations with strong INT8 performance and tight integration with federated learning frameworks, reflecting a software-stack focus on distributed learning scenarios. The energy-efficiency gap explains why dedicated neural processing units are essential: NPUs achieve 1–5 TOPS per watt vs. general-purpose CPUs at just 0.1-0.2 TOPS per watt, representing a 5–50× efficiency advantage that makes the difference between feasible and infeasible on-device training.

Each accelerator implies different learning paradigms. Apple's Neural Engine excels at fixed-precision inference but provides limited support for dynamic precision gradient computation, making it better suited for inference-heavy adaptation techniques like few-shot learning. Programmable DSPs offer greater training flexibility through vector units and mixed-precision arithmetic, enabling backpropagation on compact models when the software stack exposes the needed operators. Mobile TPU-class accelerators integrate tightly with framework runtimes and federated learning tooling, reducing the systems overhead of local training and update coordination.

These per-vendor envelopes inherit the same training-specific access patterns and hardware features discussed under section 11.2.4: the write-heavy gradient flow, the adaptive-precision datapaths, sparse-update units, and near-memory compute that distinguish a training-capable

accelerator from an inference-only one. The consequence for design is that architecture selection here is not a peak-throughput choice; it influences everything from model quantization strategies and gradient computation approaches to federated communication protocols and thermal management policies.

11.2.6 Holistic resource management strategies

The constraint analysis above reveals three challenge categories that define the on-device learning design space, and each category drives a corresponding solution pillar. Resource amplification, where training increases memory requirements by 4–12×, computational costs by 2–3×, and energy consumption proportionally, necessitates Model Adaptation approaches that reduce the scope of parameter updates while preserving learning capability. Information scarcity, including limited local datasets, non-IID distributions, privacy restrictions on data sharing, and minimal supervision, drives data efficiency solutions that extract maximum learning signal from minimal examples. Coordination challenges, such as device heterogeneity, intermittent connectivity, distributed validation complexity, and scalability requirements, motivate federated coordination mechanisms that enable privacy-preserving collaboration across device populations.

Table 11.3 reveals how this on-device learning constraint-solution mapping creates a systematic engineering framework: Model Adaptation addresses memory and compute limits through selective parameter updates, Data Efficiency maximizes learning from scarce private samples, and Federated Coordination enables privacy-preserving collaboration. Rather than viewing these as independent techniques, robust systems orchestrate all three approaches to create coherent adaptive systems that operate effectively within edge constraints.

Table 11.3: Constraint-Solution Mapping: The three fundamental constraint categories in on-device learning each drive corresponding solution approaches through direct necessity.

Constraint Category	Key Challenges	Solution Approach
Resource Amplification	• Training workloads (4–12× memory) • Memory limitations • Power constraints	**Model Adaptation** • Parameter-efficient updates • Selective layer fine-tuning • Low-rank adaptations
Information Scarcity	• Limited local datasets • Non-IID distributions	**Data Efficiency**
	• Privacy restrictions	• Few-shot learning • Meta-learning • Transfer learning
Coordination Challenges	• Device heterogeneity • Intermittent connectivity • Distributed validation complexity • Scalability requirements	**Federated Coordination** • Federated averaging and asynchronous aggregation • Secure aggregation and differential privacy • Client selection and stragglers handling

Each solution pillar extends compression and distributed-systems tools to address a specific constraint category. No single pillar suffices on its own, but their integration creates systems capable of meaningful adaptation within the severe constraints of edge deployment environments.

11.3 Model Adaptation

Personalizing a billion-parameter model on a smartwatch with 500 MB of total system memory is impossible without abandoning the goal of updating the complete model. The answer is that we do not update the whole model. We freeze the vast majority of the network and only train a tiny, strategically placed set of new parameters. This is the essence of resource-efficient model adaptation.

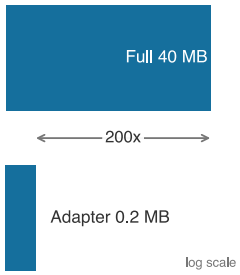
Napkin Math 11.2: The hidden cost of personalization

Problem: A team is deploying a 10M parameter vision model to a smartphone with support for 10 different "User Contexts" (Home, Office, Car, etc.). If a full fine-tuned model requires 40 MB, how much storage does using residual adapters save instead?

Math:

1. **Full Fine-Tuning**: 10 contexts $\times$ 40 MB = 400 MB.
2. **Adapter Approach**: (1 $\times$ 40 MB backbone) + (10 $\times$ 200 KB adapters) $\approx$ 42 MB.
3. **Storage Savings**: 400 MB/42 MB $\approx$ 9.5$\times$ total device storage reduction.
4. **Per-Context Efficiency**: 40 MB/0.2 MB = 200$\times$.

Systems insight: Personalization is a *storage density* problem. On a device with limited flash memory, storing 10 versions of the same 40 MB model quickly consumes 400 MB. Sharding the model into a frozen backbone and dynamic adapters reduces the marginal cost of a new user context by 200$\times$. In the edge fleet, modularity is the only way to scale intelligence without exhausting the physical hardware.



Adapters make per-user personalization cheap versus full model copies.

The engineering challenge centers on navigating a fundamental trade-off space: adaptation expressivity vs. resource consumption. At one extreme, updating all parameters provides maximum flexibility but exceeds edge device capabilities. At the other extreme, no adaptation preserves resources but fails to capture user-specific patterns. Effective on-device learning systems must operate in the middle ground, selecting adaptation strategies based on three key engineering criteria:

- Resource envelope determines which adaptation approaches are feasible. A tiny wearable-class device with 1 MB RAM requires fundamentally different strategies than a smartphone with 8 GB.
- User-specific variation determines how much adaptation complexity the system needs. Simple preference learning may require only bias updates, while complex domain shifts demand more sophisticated approaches.
- Systems integration determines whether the adaptation technique can fit existing inference pipelines, federated coordination protocols, and operational monitoring systems for model deployment and lifecycle management.

The selection of adaptation techniques follows the same logic, starting with lightweight approaches (section 11.3.1) and progressing to more expressive but resource-intensive methods (section 11.3.3). Each technique represents a different point in the engineering trade-off space.

On top of the compression baseline established in section 11.2, model adaptation adds a second move: complete model retraining is neither necessary nor feasible at the edge, so systems use pretrained representations strategically and adapt only the minimal parameter subset required to capture local variations: *preserve what works globally, adapt what matters locally*. Those three criteria turn adaptation into a device-matching decision. Weight freezing fits severe memory limits by updating only bias terms or final layers. Structured updates use low-rank and residual adaptations when the device can afford more expressiveness but not full fine-tuning. Sparse updates reserve scarce training capacity for the parameters with the highest adaptation value.

11.3.1 Weight freezing

The most direct approach to on-device learning is to freeze the majority of a model's parameters and adapt only a minimal subset. **Bias-only adaptation** holds all weights fixed and updates only the bias terms (scalar offsets applied after linear or convolutional layers). The constraint reduces trainable parameters by 100–1000$\times$, simplifies memory management during backpropagation, and helps mitigate overfitting when training data is sparse or noisy.

Consider a standard neural network layer:

$$y = Wx + b$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix, $b \in \mathbb{R}^m$ is the bias vector, and $x \in \mathbb{R}^n$ is the input. In full training, gradients are computed for both W and b. In bias-only adaptation, we constrain:

$$\frac{\partial \mathcal{L}}{\partial W} = 0, \quad \frac{\partial \mathcal{L}}{\partial b} \neq 0$$

so that only the bias is updated via gradient descent:

$$b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$$

This reduces the number of stored gradients and optimizer states, enabling training to proceed under memory-constrained conditions. On embedded devices that lack floating-point units, this reduction enables on-device learning. Listing 11.1 makes the constraint explicit: all convolutional and fully connected weights remain frozen, while only bias terms adapt to local data.

Listing 11.1: Bias-Only Adaptation: Freezes model parameters except for biases to reduce memory usage and allow on-device learning.

```
# Freeze all parameters
for name, param in model.named_parameters():
    param.requires_grad = False

# Enable gradients for bias parameters only
for name, param in model.named_parameters():
    if "bias" in name:
        param.requires_grad = True
```

This pattern ensures that only bias terms participate in the backward pass and optimizer update, simplifying the training process while maintaining adaptation capability. This is valuable when adapting pretrained models to user-specific or device-local data where the core representations remain relevant but require calibration.

The practical effectiveness of this approach is demonstrated by TinyTL (Cai, Gan, Zhu, et al. 2020), a framework explicitly designed to enable efficient adaptation of deep neural networks on microcontrollers and other severely memory-limited platforms. Rather than updating all network parameters during training (impossible on such constrained devices), TinyTL strategically freezes both the convolutional weights and the batch normalization statistics, training only the bias terms and, in some cases, lightweight residual components. This architectural constraint creates a profound shift in memory requirements during backpropagation, since the largest memory consumers (intermediate activations) no longer need to be stored for gradient computation across frozen layers.

Figure 11.7 visualizes this architectural impact by contrasting standard training with the TinyTL approach. Where conventional backpropagation requires storing activations across all layers, TinyTL freezes backbone weights and batch normalization statistics, training only the final classifier and lightweight bias modules. This eliminates the need to store activations for frozen layers, making adaptation possible within the severe memory constraints established earlier.

The memory reduction is useful only when the frozen backbone remains expressive enough for the downstream task. The bias terms allow for minor but meaningful shifts in model behavior, particularly for personalization tasks. When domain shift is more significant, TinyTL can optionally incorporate small residual adapters to improve expressivity, all while preserving the system's tight memory and energy profile.

These design choices allow TinyTL to reduce training memory usage by 10×. For instance, adapting a MobileNetV2 model using TinyTL can reduce the number of updated parameters from over 3 million to fewer than 50,000[23]. Combined with quantization, this allows local adaptation on devices with only a few hundred kilobytes of memory, making on-device learning truly feasible in constrained environments.

23 **TinyTL Memory Savings**: The approximately 68× trainable-parameter reduction (3.4 M to 50 K) collapses MobileNetV2 training memory from approximately 20 MB (weights + activations in FP32) to approximately 600 KB, fitting within a 1 MB microcontroller budget. Real deployments on STM32H7 achieve 85 percent of full fine-tuning accuracy while completing updates in approximately 30 seconds vs. 8 minutes, demonstrating that bias-only adaptation trades expressivity for feasibility on the most constrained hardware.

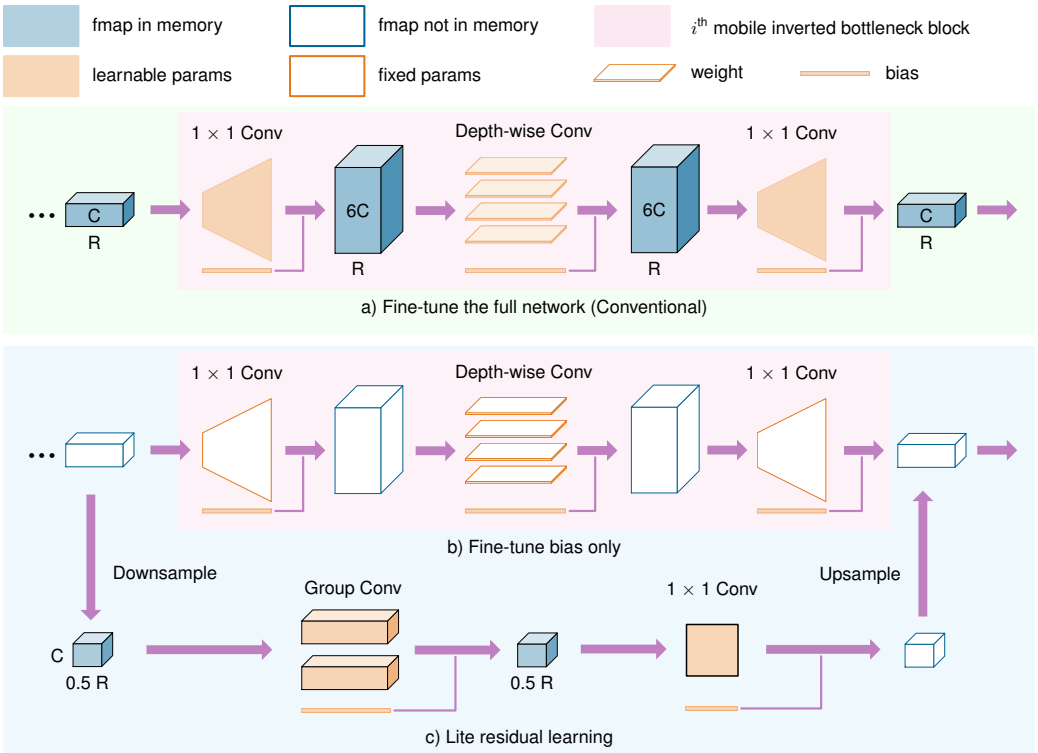


Figure 11.7: TinyTL Memory Optimization: Comparing conventional fine-tuning (top) vs. TinyTL approach (bottom). Conventional methods require storing activations for all layers during backpropagation. TinyTL freezes backbone weights and only trains bias terms, dramatically reducing memory requirements for on-device adaptation.

11.3.2 Structured parameter updates

While weight freezing provides computational efficiency and clear memory bounds, it severely limits model expressivity by constraining adaptation to a small parameter subset. When bias-only updates prove insufficient for capturing complex domain shifts or user-specific patterns, residual and low-rank techniques provide a middle ground between the extreme efficiency of weight freezing and the full expressivity of unrestricted fine-tuning. Rather than modifying existing parameters, these methods extend frozen models by adding small trainable components, such as residual adaptation modules (Houlsby et al. 2019) or low-rank parameterizations (Hu et al. 2021). The main body of the network remains fixed, while only the added components are optimized, so the model can respond to new data without giving up the memory and compute bounds that make on-device adaptation feasible.

11.3.2.1 Adapter-based adaptation

A common implementation involves inserting residual adapters, which are small residual bottleneck layers, between existing layers in a pretrained model. Consider a hidden representation h passed between layers. A residual adapter introduces a transformation:

$$h' = h + A(h)$$

where $A(\cdot)$ is a trainable function, typically composed of two linear layers with a nonlinearity:

$$A(h) = W_2\,\sigma(W_1 h)$$

with $W_1 \in \mathbb{R}^{r\times d}$ and $W_2 \in \mathbb{R}^{d\times r}$, where $r \ll d$. This bottleneck design ensures that only a small number of parameters are introduced per layer.

The adapters act as learnable perturbations on top of a frozen backbone. Because they are small and sparsely applied, they add negligible memory overhead, yet they allow the model to shift its predictions in response to new inputs.

11.3.2.2 Low-rank techniques

Another efficient strategy is to constrain weight updates themselves to a low-rank structure. Rather than updating a full matrix W, the update is approximated as:

$$\Delta W \approx UV^\top$$

where $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{n\times r}$, with $r \ll \min(m,n)$. This reduces the number of trainable parameters from mn to $r(m+n)$.

The mathematical intuition behind this decomposition connects to fundamental linear algebra principles: any matrix can be expressed as a sum of rank-one matrices through singular value decomposition. Constraining updates to low rank (typically $r = 4$ to 16) captures a restricted set of variation modes while reducing parameters. For a typical transformer layer with dimensions 768×768, full fine-tuning requires updating 589,824 parameters. With rank-4 decomposition, only \$768{×}4{×}2 = \$ 6,144 parameters are updated, a 99 percent reduction. Whether this retains most of the adaptation quality depends on the task, rank, and base model; Low-Rank Adaptation (LoRA) reports that low-rank adapters can match or approach full fine-tuning on several transformer adaptation benchmarks (Hu et al. 2021).

During adaptation, the new weight is computed as:

$$W_{\text{adapted}} = W_{\text{frozen}} + UV^\top$$

This formulation is commonly used in LoRA[24], originally developed for transformer models (Hu et al. 2021) but broadly applicable across architectures. From a systems engineering perspective, LoRA makes the trainable adapter the artifact that can be stored, transmitted, scheduled, and rolled back separately from the frozen base model.

Consider a mobile deployment where a language model with 7B parameters has a 14 GB FP16 weight footprint, before accounting for the additional gradients, optimizer states, and activations required by full fine-tuning. That footprint is impossible on smartphones with about 8 GB total memory unless the base model is aggressively quantized, partitioned, or offloaded. LoRA with rank 16 can reduce the trainable adapter parameters to tens or hundreds of megabytes, making local adaptation plausible only when the frozen base model is handled separately.

LoRA's efficiency becomes critical in intermittent connectivity scenarios. A full model update over cellular networks would require a 14 GB download and corresponding data-plan cost, while the tier-profile adapter update is 50 MB. Small adapter updates can synchronize in under a minute on good mobile links; larger adapter updates typically require LTE or Wi-Fi for sub-minute coordination and may take minutes on 3G. This is still dramatically more practical than hours-long full model transfers.

A device-tiered deployment can use different adapter configurations based on device capabilities. Flagship phones use wider bottlenecks for higher expressivity, mid-range devices use moderate bottlenecks for balanced performance, and budget devices use narrow bottlenecks to stay within about 2 GB memory limits. These residual adapter modules can be implemented efficiently on edge devices, particularly when the down-projection and up-projection matrices are small and fixed-point representations are supported. Listing 11.2 implements this low-rank adapter pattern as a residual bottleneck, showing how a compact down/up projection adds trainable capacity without updating the frozen backbone.

Example 11.1: On-device keyboard learning

Scenario: Large-scale mobile keyboard systems are a canonical deployment pattern for federated and on-device learning.

Mechanism: A keyboard can adapt next-word prediction without uploading raw keystrokes, but only if training is strictly constrained: it runs during idle, charging, Wi-Fi-connected windows and transmits compressed updates rather than text. Large deployments can combine secure aggregation with privacy accounting or noise addition so that the server learns population-level update statistics, not individual typing histories.

[24] **LoRA**: Hu et al. (2021) learns rank-r decomposition matrices A and B instead of updating full weight matrices, reducing trainable parameters by 100–10,000× while maintaining much of the adaptation quality. For on-device learning, this compression is decisive, but it affects adapter state rather than the full model footprint: a model with 7B parameters has about a 14 GB FP16 weight footprint before gradients, optimizer state, and activations, while rank 16 LoRA can reduce the trainable adapter parameters to tens or hundreds of megabytes. Practical phone deployment still requires storing, quantizing, or offloading the frozen base model.

Systems lesson: On-device learning is viable only when the training schedule, communication payload, and privacy boundary are engineered together. The model update is a systems protocol, not just a local optimization step.

Returning to the adapter mechanism, listing 11.2 shows the residual bottleneck that makes the memory savings concrete: the frozen backbone remains untouched, while only the down-projection and up-projection weights are trained.

Listing 11.2: Residual Bottleneck Adapter: The code implements a compact adapter module with down-projection and up-projection layers, reducing trainable parameter count while enabling efficient model adaptation on edge devices.

```
class Adapter(nn.Module):
    def __init__(self, dim, bottleneck_dim):
        super().__init__()
        # Project from full dimension to bottleneck (e.g., 768 -> 16)
        self.down = nn.Linear(
            dim, bottleneck_dim
        )  # W1: learns compression
        # Project back to original dimension (e.g., 16 -> 768)
        self.up = nn.Linear(
            bottleneck_dim, dim
        )  # W2: learns expansion
        self.activation = nn.ReLU()

    def forward(self, x):
        # Residual connection: original + low-rank adaptation
        # Only adapter params trained; base model frozen
        return x + self.up(self.activation(self.down(x)))
```

This adapter adds a small residual transformation to a frozen layer. When inserted into a larger model, only the adapter parameters are trained.

11.3.2.3 Edge personalization

Adapters enable an efficient multi-tenant architecture for edge intelligence, where a single frozen backbone supports multiple personalized contexts. Consider a 10M parameter vision model deployed on a smartphone with 8 GB RAM. Full fine-tuning would require storing a separate 40 MB copy of the model weights for each personalization context (home, office, outdoor), rapidly consuming the device's storage. A rank-64 adapter introduces only approximately 50,000 parameters (roughly 200 KB) per context—a 200$\times$ reduction in storage. This efficiency allows the device to store dozens of specialized adapters that share the same frozen backbone, swapping them dynamically based on the user's location or activity.

This **adapter switching** pattern transforms the smartphone from a static inference engine into a context-aware learning system. When the user enters a low-light environment, the system can load the appropriate adapter into the vision pipeline instead of replacing the entire backbone. The modularity extends naturally to federated learning: in the 10M-parameter example above, transmitting a 200 KB adapter is far cheaper than transmitting a 40 MB full-model update over a bandwidth-constrained cellular connection. Residual-adapter studies show that small trainable modules can adapt a shared visual backbone across multiple visual domains (Rebuffi et al. 2017), making them a useful design point between personalization quality and system resources. The frozen backbone helps preserve general visual representations, while adapters provide a bounded place for user- or context-specific updates without exposing the entire model to catastrophic forgetting, the loss of previously learned general behavior when new updates overwrite it (formalized in section 11.4.2 later in this chapter).

In smartphone camera pipelines, environmental lighting, user preferences, and lens distortion vary between users. A shared model can be frozen and fine-tuned per-device using a few residual modules, allowing lightweight personalization without destabilizing the base model. In voice-based systems, adapter modules reduce word error rates in personalized speech recognition without

retraining the full acoustic model. They also allow easy rollback or switching between user-specific versions—a critical operational capability when local adaptation occasionally degrades rather than improves performance.

11.3.2.4 Performance vs. resource trade-offs

Selecting the right adaptation strategy requires quantitative analysis of the resource-expressivity spectrum. For a 10M parameter model, the trade-offs are concrete. Bias-only updates are the most efficient, requiring just 50 KB of trainable parameters and negligible compute overhead, but they struggle to adapt to structural domain shifts like changing sensor geometries or new object categories. At the other extreme, full fine-tuning offers maximum expressivity but demands 40 MB or more of gradients and optimizer states, making it infeasible for background training on typical mobile devices. Low-rank techniques (LoRA) and residual adapters occupy the strategic middle ground: a rank-16 LoRA configuration requires approximately 2 MB of trainable memory, while a standard residual adapter needs approximately 5 MB.

The decision between LoRA and residual adapters often hinges on inference latency constraints. LoRA matrices can be mathematically merged into the frozen backbone weights during inference ($W_{\text{final}} = W_{\text{frozen}} + UV^\top$), resulting in zero additional inference latency. This makes LoRA ideal for latency-critical paths like real-time video processing where every millisecond matters. Residual adapters, conversely, remain distinct modules ($y = f(x) + \text{Adapter}(x)$), adding 1–3 percent to inference latency due to the extra forward pass through the adapter layers. However, this separation enables the hot-swapping capability described earlier: switching adapters requires loading only the small adapter weights rather than recomputing merged matrices.

Implementing these adaptation techniques requires system-level support for dynamic computation graphs and the ability to selectively inject trainable parameters. Not all deployment environments or inference engines support such capabilities natively. TensorFlow Lite and Open Neural Network Exchange (ONNX) Runtime Mobile provide support for adapter-style architectures, but custom inference stacks on microcontrollers may require manual implementation of the adapter forward pass.

System architects should apply a device-tier framework to this decision. On flagship phones (Tier 1) with dedicated Neural Engines and 8+ GB RAM, residual adapters provide a strong balance of modularity and speed, enabling context-aware switching with minimal latency impact. On mid-range devices (Tier 2) with 4–8 GB RAM, LoRA is attractive because weight merging eliminates runtime overhead entirely. For ultra-constrained IoT endpoints (Tier 3) with less than 1 GB RAM, bias-only updates may be the only practical option. This tiered approach ensures that the adaptation mechanism matches the physical reality of the hardware, preventing thermal throttling while delivering meaningful personalization at each capability level.

Checkpoint 11.2: Edge personalization

Verify your understanding of efficient on-device adaptation:

- ☐ **Adapter switching**: Explain why Adapter Switching is more storage-efficient than maintaining separate full-model copies for different user contexts.
- ☐ **Adapter size**: Compare the size of a **rank-64 adapter** to a standard 10M parameter vision model.
- ☐ **Residual vs. LoRA**: Identify the deployment tier (Tier 1, 2, or 3) where **Residual Adapters** are preferred over LoRA and justify the choice.
- ☐ **Adapter bandwidth**: Evaluate the claim that transmitting adapter weights in federated learning consumes more bandwidth than transmitting the entire model.

11.3.3 Sparse updates

Sparse updates are the highest-expressivity end of the model adaptation hierarchy. Instead of adding new parameters or choosing a fixed subset in advance, the system dynamically identifies which existing parameters provide the greatest adaptation benefit for a specific task or user. This

preserves more of full fine-tuning's flexibility while keeping the update footprint small enough for edge deployment.

The sparse-update decision goes beyond updating fewer parameters. Training remains resource-intensive even after model-adaptation techniques restrict learning to a small subset. Sparse updating addresses the remaining cost by selecting only task-relevant parameters, reducing memory and compute while preserving meaningful personalization.

The key insight is that layers do not contribute equally to local performance gains. If the system can identify the minimal subset with the highest adaptation value, it can spend scarce training capacity where it changes the model's behavior most.

11.3.3.1 Sparse update design

Let a neural network be defined by parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_{N_L}\}$ across N_L layers. In standard fine-tuning, we compute gradients and perform updates on all parameters:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, \quad \text{for } i = 1, \ldots, N_L$$

In task-adaptive sparse updates, we select a small subset $\mathcal{S} \subset \{1, \ldots, N_L\}$ such that only parameters in $\mathcal{S}$ are updated:

$$\theta_i \leftarrow \begin{cases} \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}, & \text{if } i \in \mathcal{S} \\ \theta_i, & \text{otherwise} \end{cases}$$

The challenge lies in selecting the optimal subset $\mathcal{S}$ given memory and compute constraints. A principled strategy is contribution analysis, an empirical method that estimates how much each layer contributes to downstream performance improvement. For example, one can measure the marginal gain from updating each layer independently:

1. Freeze the entire model.
2. Unfreeze one candidate layer.
3. Finetune briefly and evaluate improvement in validation accuracy.
4. Rank layers by performance gain per unit cost (e.g., per KB of trainable memory).

This layer-wise profiling yields a ranking from which $\mathcal{S}$ can be constructed subject to a memory budget.

A concrete example is TinyTrain, a method designed to allow rapid adaptation on-device (Kwon et al. 2024). TinyTrain combines offline preparation with a task-adaptive sparse-update method that dynamically selects layers or channels to update using user data, memory limits, and compute limits. At runtime, the system updates only the selected subset rather than treating the whole network as trainable.

In implementation, selective layer updating turns the ranking into `requires_grad` decisions or equivalent graph masks. Listing 11.3 extends this pattern to profiling-driven layer selection, demonstrating how hardware profiling determines which layers to update based on contribution scores and memory constraints.

Listing 11.3: Selective Layer Updating: This technique allows fine-tuning specific layers of a pretrained model while keeping others frozen, optimizing computational resources for targeted improvements. Source: PyTorch Documentation.

```
# Selective layer update based on contribution analysis
# Layers selected via profiling: conv2 and fc have highest
# accuracy-per-KB impact for this task

for name, param in model.named_parameters():
    if "conv2" in name or "fc" in name:
        param.requires_grad = True  # Train high-impact layers
    else:
        param.requires_grad = False  # Freeze low-impact layers

# Result: ~10% of params trainable, ~60% of adaptation quality
# Memory savings: gradient storage only for selected layers
```

TinyTrain makes the runtime motivation concrete. Consider a scenario where a user wears an augmented reality headset that performs real-time object recognition. As lighting and environments shift, the system must adapt to maintain accuracy, but training must occur during brief idle periods or while charging. TinyTrain addresses this constraint through task-adaptive sparse updates: the deployment process selects a small update set that fits the device's memory and compute budget. The approach keeps adaptation fast, energy-efficient, and memory-aware when the target task and device profile match the method's assumptions.

Task-adaptive sparse updates still introduce system-level trade-offs. Contribution analysis has a nontrivial computational cost even when it occurs during pretraining or initial profiling, so the deployment pipeline must budget for it. Stability also requires attention: if too few parameters are selected for updating, the model may underfit the target distribution, so teams need validation thresholds for the selected subset before deployment. The selection policy must also account for hardware-specific execution costs because some parameters show high contribution scores but prove expensive to update on certain architectures. Despite these trade-offs, task-adaptive sparse updates provide a practical mechanism to scale adaptation across deployment contexts from microcontrollers to mobile devices (Diao et al. 2023).

11.3.3.2 Adaptation strategy comparison

Each adaptation strategy offers a distinct balance between expressivity, resource efficiency, and implementation complexity. Bias-only adaptation is the most lightweight approach, updating only scalar offsets in each layer while freezing all other parameters. This reduces memory requirements and computational burden, making it suitable for devices with tight memory and energy budgets. However, its limited expressivity means it is best suited to applications where the pretrained model already captures most of the relevant task features and only minor local calibration is required.

Residual adaptation, often implemented via adapter modules, introduces a small number of trainable parameters into the frozen backbone of a neural network. This allows for greater flexibility than bias-only updates, while still maintaining control over the adaptation cost. Because the backbone remains fixed, training can be performed efficiently and safely under constrained conditions. This method supports modular personalization across tasks and users, making it a favorable choice for mobile settings where moderate adaptation capacity is needed.

Task-adaptive sparse updates offer the greatest potential for task-specific finetuning by selectively updating only a subset of layers or parameters based on their contribution to downstream performance. While this method allows expressive local adaptation, it requires a mechanism for layer selection, through profiling, contribution analysis, or meta-training, which introduces additional complexity. Nonetheless, when deployed carefully, it allows for dynamic trade-offs between accuracy and efficiency, particularly in systems that experience large domain shifts or evolving input conditions. Table 11.4 contrasts adaptation strategy trade-offs across trainable parameters, memory overhead, expressivity, use-case suitability, and system requirements, revealing how the optimal choice depends on application domain, available hardware, latency constraints, and expected distribution shift.

Table 11.4: Adaptation Strategy Trade-Offs: Table entries characterize three approaches to model adaptation, bias-only updates, residual adapters, and sparse layer updates, by quantifying their impact on trainable parameters, memory overhead, expressivity, suitability for different use cases, and system requirements. These characteristics reveal the inherent trade-offs between model flexibility, computational cost, and performance when deploying machine learning systems in dynamic environments.

Technique	Trainable Parameters	Memory Overhead	Expressivity	Use Case Suitability	System Requirements
Bias-Only Updates	Bias terms only	Minimal	Low	Simple personalization; low variance	Extreme memory/compute limits
Residual Adapters	Adapter modules	Moderate	Moderate to High	User-specific tuning on mobile	Mobile-class SoCs with runtime support
Sparse Layer Updates	Selective parameter subsets	Variable	High (task-adaptive)	Real-time adaptation; domain shift	Requires profiling or meta-training

While freezing weights and training adapters solves the memory bottleneck, it leaves us with a statistical problem. We have created a highly efficient learning mechanism, but edge devices rarely have the thousands of labeled examples required to properly train even these small adapters. We must now tackle the second pillar of on-device learning: data efficiency.

11.4 Data Efficiency

A user corrects their smartphone keyboard's autocorrect perhaps three times a day. We cannot ask them to type 10,000 labeled sentences to train our language model. Data efficiency at the edge means learning aggressively from a sparse, noisy, and uncurated trickle of implicit user feedback, maximizing the signal extracted from each interaction.

The systems engineering challenge centers on a critical trade-off: data collection cost vs. adaptation quality. Edge devices face severe data acquisition constraints that reshape learning system design in ways not encountered in centralized training. Understanding and navigating these constraints requires systematic analysis of four interconnected engineering dimensions:

- Acquisition cost includes user friction, energy consumption, storage overhead, and privacy risk. A voice assistant learning from audio samples must balance improvement potential against battery drain and user comfort with always-on recording.
- Collection depth determines whether the system spends limited capacity on broad coverage or detailed examples. A mobile keyboard can collect many shallow typing patterns or fewer detailed interaction sequences, each strategy implying different learning approaches.
- Learning urgency determines whether the system must adapt from minimal examples immediately, as in emergency response scenarios, or can accumulate evidence over time, as in user preference learning.
- Lifecycle integration determines whether data efficiency techniques fit the model adaptation approaches from section 11.3, federated coordination (section 11.5), and operational monitoring for model deployment and lifecycle management.

These engineering constraints create a systematic trade-off space where different data efficiency approaches serve different combinations of constraints. Rather than choosing a single technique, successful on-device learning systems typically combine multiple approaches, each addressing specific aspects of the data scarcity challenge.

The strategy choice depends on which scarcity binds first. Few-shot learning fits cases where a few labeled or weakly labeled examples must drive personalization. Streaming updates fit settings where useful data arrives incrementally and cannot be batched in advance. Experience replay spends memory to stabilize continual updates by reusing scarce examples. Data compression reduces the storage cost of those histories so replay and lightweight training remain feasible within edge memory budgets.

11.4.1 Few-shot learning and data streaming

Together, these techniques turn data scarcity into an allocation problem: spend labels, memory, compute, and privacy budget where they produce the most reliable local signal. In conventional machine learning workflows, effective training typically requires large labeled datasets, carefully curated and preprocessed to ensure sufficient diversity and balance. On-device learning, by contrast, must often proceed from only a handful of local examples, collected passively through user interaction or ambient sensing, and rarely labeled in a supervised fashion. These constraints motivate two complementary adaptation strategies: few-shot learning, in which models generalize from a small, static set of examples, and streaming adaptation, where updates occur continuously as data arrives.

Few-shot adaptation is particularly relevant when the device observes a small number of labeled or weakly labeled instances for a new task or user condition (Wang et al. 2020). In such settings, it is often infeasible to perform full finetuning of all model parameters without overfitting. Instead, methods such as bias-only updates, adapter modules, or prototype-based classification are employed to make use of limited data while minimizing capacity for memorization. Let $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{K}$ denote a K-shot dataset of labeled examples collected on-device. The model update is viable only if three constraints remain bounded:

- Gradient work stays small: $K_{\text{steps}} \ll 100$.

- Updated state stays compact: $\|\theta_{\text{updated}}\| \ll \|\theta\|$.
- Prior task knowledge remains preserved, avoiding catastrophic forgetting.

Keyword spotting (KWS) systems are a common target for data-efficient on-device adaptation; Speech Commands provides a standard dataset for training and evaluating limited-vocabulary KWS models under on-device constraints (Warden 2018). These models are used to detect fixed phrases, including phrases like "Hey Siri"[25] or "OK Google", with low latency and high reliability. A typical KWS model consists of a pretrained acoustic encoder (e.g., a small convolutional or recurrent network that transforms input audio into an embedding space) followed by a lightweight classifier. In commercial systems, the encoder is trained centrally using thousands of hours of labeled speech across multiple languages and speakers. However, supporting custom wake words (e.g., "Hey Jarvis") or adapting to underrepresented accents and dialects is often infeasible via centralized training due to data scarcity and privacy concerns.

Few-shot adaptation solves this problem by finetuning only the output classifier or a small subset of parameters, including bias terms, using just a few example utterances collected directly on the device. For example, a user might provide 5–10 recordings of their custom wake word. These samples are then used to update the model locally, while the main encoder remains frozen to preserve generalization and reduce memory overhead. This allows personalization without requiring additional labeled data or transmitting private audio to the cloud.

The approach is computationally efficient and aligned with privacy-preserving design principles. Because only the output layer is updated, often involving a single gradient step or prototype computation, the total memory footprint and runtime compute are compatible with mobile-class devices or even microcontrollers.

Beyond static few-shot learning, many on-device scenarios benefit from streaming adaptation, where models must learn incrementally as new data arrives (Hayes et al. 2020). Streaming adaptation generalizes this idea to continuous, asynchronous settings where data arrives incrementally over time. Let $\{x_t\}_{t=1}^{\infty}$ represent a stream of observations. In streaming settings, the model must update itself after observing each new input, typically without access to prior data, and under bounded memory and compute. The computable loss may come from implicit feedback, pseudo-labels, self-supervised objectives, reconstruction error, or a task-specific proxy rather than explicit human labels. The model update can be written generically as:

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(x_t; \theta_t)$$

where η_t is the learning rate at time t. This form of adaptation is sensitive to noise and drift in the input distribution, and thus often incorporates mechanisms such as learning rate decay, meta-learned initialization, or update gating to improve stability.

Aside from KWS, practical examples of these strategies abound. In wearable health devices, a model that classifies physical activities may begin with a generic classifier and adapt to user-specific motion patterns using only a few labeled activity segments. In smart assistants, user voice profiles are fine-tuned over time using ongoing speech input, even when explicit supervision is unavailable. In such cases, local feedback, including correction, repetition, or downstream task success, can serve as implicit signals to guide learning.

Few-shot and streaming adaptation form the foundation for more advanced memory and replay strategies that address the stability challenges of continuous on-device learning.

11.4.2 Experience replay

Definition 11.2: Catastrophic forgetting

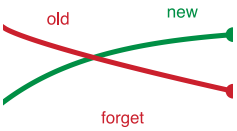
Catastrophic Forgetting is the phenomenon in which a neural network trained sequentially on new tasks or data distributions rapidly loses performance on previously learned tasks, because gradient updates that optimize the new objective overwrite weight configurations that supported the old ones.

1. **Significance**: Without explicit mitigation, fine-tuning a deployed classifier on a new domain can sharply reduce prior-task accuracy, even when the new domain is closely

25 **"Hey Siri" Power Budget**: The always-on detector consumes $<$1 mW while monitoring audio continuously, using a compact acoustic model on a low-power always-on processor rather than the main application cores. The constraint envelope is extreme: $<$100 ms detection latency, $<$0.1 false activations per hour, and $>$95 percent true positive rate across accents and noise conditions. Few-shot personalization must improve accuracy within this fixed power and model-size budget, meaning adaptation can tune classifier weights but cannot expand the model.

related to the old one (Kirkpatrick et al. 2017). This is the central failure mode of on-device and federated learning: every device that adapts locally to its user's recent data risks erasing the general capabilities the centralized training run paid for, and the loss is invisible until inference encounters out-of-recent-context inputs.

2. **Distinction**: Unlike overfitting (a generalization failure on a single fixed task) and distribution shift (a deployment-time mismatch between training and inference data), forgetting is a training-time phenomenon caused by the sequential structure of the update stream itself; the model's capacity is sufficient, but its weights cannot encode the new objective without disturbing the encoding of the old one.
3. **Common pitfall**: A frequent misconception is that a small experience-replay buffer is sufficient protection. Naive replay of 100 to 1,000 stored samples can mask short-horizon forgetting on a held-out validation set yet still degrade rare-class or long-tail performance, because the buffer's sampling distribution under-represents the tails. Production deployments require capacity-proportional buffers, prioritized replay, or weight-regularization methods such as elastic weight consolidation (section 11.7.2).



Learning the new task erodes accuracy on the old one.

Experience replay addresses catastrophic forgetting in continuous learning scenarios by maintaining a buffer of representative examples from previous learning episodes. This technique, originally developed for reinforcement learning (Mnih et al. 2015), proves essential in on-device learning where sequential data streams can cause models to overfit to recent examples. Here, experience replay addresses the immediate stability need; bio-inspired lifelong learning (section 11.7.2) addresses the same stability problem through local plasticity and consolidation mechanisms.

Unlike server-side replay strategies that rely on large datasets and extensive compute, on-device replay must operate with extremely limited capacity, often with tens or hundreds of samples, and must avoid interfering with user experience (Rolnick et al. 2019). Buffers may store only compressed features or distilled summaries, and updates must occur opportunistically (e.g., during idle cycles or charging). These system-level constraints reshape how replay is implemented and evaluated in the context of embedded ML.

Let $\mathcal{M}$ represent a memory buffer that retains a fixed-size subset of training examples. At time step t, the model receives a new data point (x_t, y_t) and appends it to $\mathcal{M}$. A replay-based update then samples a batch $\{(x_i, y_i)\}_{i=1}^{k}$ from $\mathcal{M}$ and applies a gradient step:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \left[\frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(x_i, y_i; \theta_t) \right]$$

where θ_t are the model parameters, η is the learning rate, and $\mathcal{L}$ is the loss function. Over time, this replay mechanism allows the model to reinforce prior knowledge while incorporating new information.

A practical on-device implementation might use a ring buffer to store a small set of compressed feature vectors rather than full input examples. Listing 11.4 implements this minimal replay buffer design, demonstrating how a circular storage mechanism enables efficient memory management while balancing historical knowledge retention with new information incorporation.

Listing 11.4: Replay Buffer: Implements a circular storage mechanism for efficient memory management in constrained environments. This approach allows models to efficiently retain and sample from recent data points, balancing the need to use historical information while incorporating new insights.

```
# Ring buffer for experience replay on memory-constrained devices
# Stores compressed features (not raw inputs) to minimize footprint
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity  # Fixed size, e.g., 1000 examples
        self.buffer = []
        self.index = 0  # Circular write pointer

    def store(self, feature_vec, label):
        if len(self.buffer) < self.capacity:
            self.buffer.append((feature_vec, label))  # Fill phase
        else:
            self.buffer[self.index] = (
                feature_vec,
                label,
            )  # Overwrite oldest
        self.index = (self.index + 1) % self.capacity  # Wrap around

    def sample(self, k):
        # Random sampling prevents recency bias during replay
        return random.sample(self.buffer, min(k, len(self.buffer)))
```

This implementation maintains a fixed-capacity cyclic buffer, storing compressed representations (e.g., last-layer embeddings) and associated labels. Such buffers are useful for replaying adaptation updates without violating memory or energy budgets.

In TinyML applications[26], experience replay has been applied to problems such as gesture recognition, where devices must continuously improve predictions while observing a small number of events per day. Instead of training directly on the streaming data, the device stores representative feature vectors from recent gestures and uses them to finetune classification boundaries periodically. Similarly, in on-device keyword spotting, replaying past utterances can improve wake-word detection accuracy without the need to transmit audio data off-device.

While experience replay improves stability in data-sparse or nonstationary environments, it introduces several trade-offs. Storing raw inputs may breach privacy constraints or exceed storage budgets, especially in vision and audio applications. Replaying from feature vectors reduces memory usage but may limit the richness of gradients for upstream layers. Write cycles to persistent flash memory, which are frequently necessary for long-term storage on embedded devices, can also raise wear-leveling concerns. These constraints require careful co-design of memory usage policies, replay frequency, and feature selection strategies, particularly in continuous deployment scenarios.

[26] **TinyML Scale**: Over 100 billion microcontrollers ship annually, but fewer than 1 percent support on-device learning due to memory ($<256KB$) and power (<$1 mW) constraints. The gap between deployed hardware and ML-capable hardware defines the engineering challenge: enabling experience replay and adaptation on chips costing under $1 requires algorithms designed for kilobytes, not gigabytes.

11.4.3 Data compression

In many on-device learning scenarios, the raw training data may be too large, noisy, or redundant to store and process effectively. This motivates the use of compressed data representations, where the original inputs are transformed into lower-dimensional embeddings or compact encodings that preserve salient information while minimizing memory and compute costs.

Compressed representations serve two complementary goals: they reduce the footprint of stored data, allowing devices to maintain longer histories or replay buffers under tight memory budgets (Hayes et al. 2020), and they simplify the learning task by projecting raw inputs into more structured feature spaces, often learned via pretraining or meta-learning, in which efficient adaptation is possible with minimal supervision.

One common approach is to encode data points using a pretrained feature extractor and discard the original high-dimensional input. For example, an image x_i might be passed through a CNN to produce an embedding vector $z_i = f(x_i)$, where $f(\cdot)$ is a fixed feature encoder. This embedding captures visual structure (e.g., shape, texture, or spatial layout) in a compact representation, usually ranging from 64 to 512 dimensions, suitable for lightweight downstream adaptation.

Mathematically, training can proceed over compressed samples (z_i, y_i) using a lightweight decoder or projection head. Let θ represent the trainable parameters of this decoder model, which is typically a small neural network that maps from compressed representations to output predictions. As each example is presented, the model parameters are updated using gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(g(z_i; \theta), y_i)$$

Six components define this update rule:

- z_i is the compressed representation of the i-th input,
- y_i is the corresponding label or supervision signal,
- $g(z_i; \theta)$ is the decoder's prediction,
- $\mathcal{L}$ is the loss function measuring prediction error,
- η is the learning rate, and
- ∇_θ denotes the gradient with respect to the parameters θ.

This formulation highlights how only a compact decoder model, which has the parameter set θ, needs to be trained, making the learning process feasible even when memory and compute are limited.

Advanced approaches extend beyond fixed encoders when replay storage, rather than model compute, becomes the binding constraint. They learn discrete or sparse dictionaries: bases for recurring sensor-trace patterns that can be stored more compactly than raw replay examples. A dataset of sensor traces can be factorized as $X \approx \Phi_{\text{dict}} C$, where Φ_{dict} is a dictionary of basis patterns and C is a block-sparse coefficient matrix indicating which patterns are active in each example. By updating only a small number of dictionary atoms or coefficients, the model adapts with minimal replay-buffer overhead.

Compressed representations prove useful in privacy-sensitive settings, as they allow raw data to be discarded or obfuscated after encoding. Compression acts as an implicit regularizer, smoothing the learning process and mitigating overfitting when only a few training examples are available.

In practice, these strategies have been applied in domains such as keyword spotting, where raw audio signals are first transformed into Mel-frequency cepstral coefficients (MFCCs)[27], a compact, lossy representation of the power spectrum of speech. These MFCC vectors serve as compressed inputs for downstream models, enabling local adaptation using only a few kilobytes of memory.

[27] **MFCCs (Mel-Frequency Cepstral Coefficients)**: Audio features that apply mel-scale frequency warping (emphasizing lower frequencies where speech concentrates) followed by cepstral analysis, reducing a 320-sample audio window (20 ms at 16 kHz) from 640 bytes to approximately 50 bytes in 12–13 coefficients. This 13× compression preserves speech intelligibility while making on-device keyword spotting feasible in kilobytes of memory, a compression ratio that directly determines replay buffer capacity for few-shot voice adaptation.

11.4.4 Data efficiency strategy comparison

Few-shot learning, experience replay, and compressed data representations each address different facets of on-device adaptation when data is scarce or streaming. Their effectiveness depends on system-level factors: memory capacity, data availability, task structure, and privacy requirements.

Few-shot adaptation excels when a small but informative set of labeled examples is available, particularly when personalization or rapid task-specific tuning is required. It minimizes compute and data needs, but its effectiveness depends on the quality of pretrained representations and the alignment between the initial model and the local task.

Experience replay addresses continual adaptation by mitigating forgetting and improving stability, especially in nonstationary environments. It allows reuse of past data, but requires memory to store examples and compute cycles for periodic updates. Replay buffers may also raise privacy or longevity concerns, especially on devices with limited storage or flash write cycles.

Compressed data representations reduce the footprint of learning by transforming raw data into compact feature spaces. This approach supports longer retention of experience and efficient finetuning, particularly when only lightweight heads are trainable. Compression can introduce information loss, and fixed encoders may fail to capture task-relevant variability if they are not well-aligned with deployment conditions. Table 11.5 summarizes the on-device learning trade-offs across data requirements, memory overhead, and use case suitability for each technique.

Table 11.5: On-Device Learning Trade-Offs: Technique choice depends on the binding edge constraint: few-shot adaptation minimizes labeled-data needs, replay improves stability under drift at the cost of memory, and compressed representations extend retention when raw data is too large or too sensitive to keep.

Technique	Data Requirements	Memory/Compute Overhead	Use Case Fit
Few-Shot Adaptation	Small labeled set (K-shots)	Low	Personalization, quick on-device finetuning
Experience Replay	Streaming data	Moderate (buffer & update)	Non-stationary data, stability under drift
Compressed Representations	Unlabeled or encoded data	Low to Moderate	Memory-limited devices; privacy-sensitive contexts

In practice, these methods are not mutually exclusive. Many real-world systems combine them to achieve robust, efficient adaptation. For example, a keyword spotting system may use compressed audio features (e.g., MFCCs), finetune a few parameters from a small support set, and maintain a replay buffer of past embeddings for continual refinement.

Efficient adaptation and data strategies enable a single device to learn from its user. Millions of isolated devices learning individually, however, waste a massive opportunity to share their localized insights. Aggregating this decentralized intelligence without compromising user privacy requires the mathematics of federated learning.

11.5 Federated Learning: Algorithms

Suppose a hospital wants to train an AI on patient records to predict complications, but stringent privacy laws make it illegal to move the raw patient data to a central server. This is cross-silo federated learning: a small number of reliable institutions collaborate without centralizing their data. The challenge is to enable multiple hospitals to collaboratively train a global model without ever sharing their raw data. Federated learning solves this by moving the computation to the data, broadcasting the model to the edge, and only aggregating the resulting model updates.

Consider a voice assistant deployed to 10 million homes. This is cross-device federated learning: many unreliable, intermittently connected devices contribute small updates when local conditions permit. Each device adapts locally to its user's voice, accent, and vocabulary. Device A learns that "data" is pronounced / de tə/, Device B learns / dætə/. Device C encounters the rare phrase "machine learning" frequently (tech household), while Device D never sees it (nontech household). After six months of isolated local adaptation, four failure modes emerge:

- Each device excels at one user's patterns but generalizes poorly beyond them.
- Rare vocabulary is learned on some devices and forgotten on others.
- Local biases accumulate without correction from the broader population.
- Valuable insights discovered on one device benefit no other devices.

Individual on-device learning, while effective for local personalization, faces fundamental limitations when devices operate in isolation. Each device observes only a narrow slice of the full data distribution, limiting generalization. Device capabilities vary dramatically, creating learning imbalances across the population. Valuable insights learned on one device cannot benefit others, reducing overall system intelligence. Without coordination, models may diverge or degrade over time due to local biases.

Federated learning addresses these coordination constraints through privacy-preserving collaboration: devices contribute to collective intelligence without sharing raw data. The approach transforms data locality from a limitation into a privacy feature, allowing systems to learn from population-scale data while keeping individual information secure. That promise only holds if the system treats gradients and participation metadata as sensitive outputs, because those signals can still reveal information about local users.

Definition 11.3: Federated learning

Federated Learning is a decentralized training paradigm where distributed devices collaboratively train a shared model using local data while exchanging only model updates (gradients or weights).

1. **Significance**: It transforms the constraint of data locality into a privacy feature. Within the iron law, federated learning is constrained by the Wide-Area Bandwidth (BW) and the extreme Heterogeneity of the Fleet, where device-specific efficiency (η_{hw}) and availability can vary by orders of magnitude.
2. **Distinction**: Unlike Centralized Training, where data is moved to the compute, federated learning moves compute to the data, ensuring that raw information never leaves the device.
3. **Common pitfall**: A frequent misconception is that federated learning is "inherently private." In reality, model updates themselves can leak information through Gradient Inversion attacks, requiring additional protections like differential privacy or secure aggregation.

The operational payoff is not abstract: moving compact model updates instead of private observations can shrink the bandwidth budget by orders of magnitude for camera-personalization workloads.

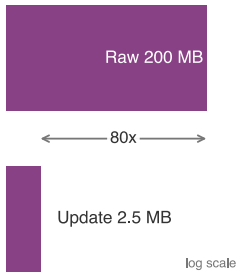
Napkin Math 11.3: Model updates vs. raw data

Problem: A team is designing a federated camera-personalization system that learns from 200 MB of compressed user images per week. Should the system upload the raw images to the cloud for training, or use federated learning to send model updates instead?

Math: Bandwidth efficiency is the ratio of raw data volume to model update size.

1. **Raw Data Upload**: 200 MB/week.
2. **Federated Update**: A compressed model update (5M params) is only 2.5 MB.
3. **Bandwidth Reduction**: 200 MB/2.5 MB = 80× savings.

Systems insight: Federated learning is a *network multiplier*. For a user on a limited cellular plan, uploading 200 MB of raw data is expensive and slow. Uploading a 2.5 MB update is nearly invisible. Shifting compute to the data reduces network load by 80×, enabling continuous learning even in bandwidth-constrained environments. In the Machine Learning Fleet, this is how systems scale to large user populations without bankrupting the networking budget or violating user privacy.



Federated learning ships updates, not raw data, cutting network load.

The three-phase evolution introduced in figure 11.3 (local-only, centralized cloud, federated) now resolves into federated learning's defining position. Figure 11.8 restates that comparison with the emphasis the algorithms section needs: offline learning trains centrally and deploys static models; on-device learning adapts locally but in isolation, sharing no insight across users; and federated learning bridges the two by coordinating updates globally while raw data stays local, so it benefits from distributed model improvements without centralizing the data that produced them.

11.5.1 Privacy-preserving collaborative learning

Across application domains such as Gboard's keyboard personalization, wearable health monitoring, and voice interfaces, compact on-device updates solve only the data-movement side of the problem. Federated learning (FL) adds the coordination layer: it trains a shared model across a population of devices without transferring raw data to a central server (McMahan et al. 2017). Unlike traditional centralized training pipelines, which require aggregating all training data in a single location, federated learning distributes the training process itself. Each participating device computes updates

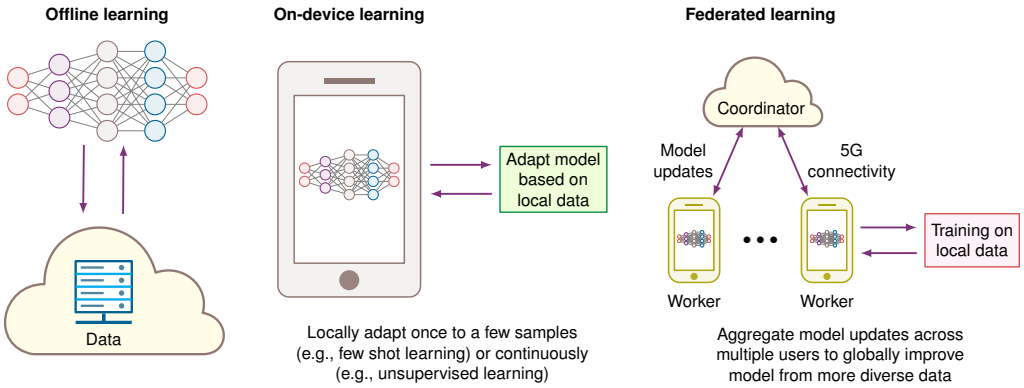


Figure 11.8: Federated learning balances data privacy with collective model improvement by coordinating local training across distributed devices, unlike offline learning's centralized approach or on-device learning's isolated adaptation. Each paradigm handles data location and model update strategies differently, revealing the trade-offs between personalization, data security, and global knowledge sharing.

based on its local data and contributes to a global model through an aggregation protocol, typically coordinated by a central server.

This shift aligns closely with mobile, edge, and embedded systems because it preserves data locality while still improving the shared model. However, the privacy and personalization benefits are real only if the system also handles client variability, communication efficiency, and non-i.i.d. data distributions[28] through specialized protocols and coordination mechanisms.

Three areas define federated learning in on-device settings: the core learning protocols that govern coordination across devices, strategies for scheduling and communication efficiency, and approaches to personalization. Privacy mechanisms such as secure aggregation and differential privacy wrap these areas rather than replacing them: they determine what the server can infer from updates and how much noise the training loop must absorb.

[28] **Non-IID Data**: "Independently and Identically Distributed." In the cloud, we shuffle data to ensure every batch represents the global distribution. On the edge, each user's data is inherently biased (non-IID): a user in Tokyo mostly types Japanese, while a user in London types English. This bias creates *gradient divergence*, making standard FedAvg 2–5× slower to converge than centralized SGD.

11.5.2 Learning protocols

The protocols that make distributed coordination practical at scale must solve three engineering challenges simultaneously: ensuring that local training produces compatible updates despite non-IID data distributions, aggregating those updates efficiently despite bandwidth constraints, and coordinating timing across devices with heterogeneous availability patterns.

11.5.2.1 Local training

Local training is the point where privacy preservation becomes a systems boundary. Individual devices compute model updates from private data, but every local choice affects the validity of the aggregate: which base model the device starts from, which local examples it samples, how much optimization it performs, and what update it sends back. The loop is therefore useful to present as an ordered protocol rather than as an abstract definition:

1. **Model Initialization**: Each device initializes its local model parameters, often by downloading the latest global model from the server.
2. **Local Data Sampling**: The device samples a subset of its local data for training. This data may be non-IID, meaning that it may not be uniformly distributed across devices.
3. **Local Training**: The device performs a number of training iterations on its local data, updating the model parameters based on the computed gradients.
4. **Model Update**: After local training, the device computes a model update (e.g., the difference between the updated and initial parameters) and prepares to send it to the server.
5. **Communication**: The device transmits the model update to the server, typically using a secure communication channel to protect user privacy.
6. **Model Aggregation**: The server aggregates the updates from multiple devices to produce a new global model, which is then distributed back to the participating devices.

The process repeats iteratively, with devices periodically downloading the latest global model and performing local training. Update frequency varies based on system constraints, device availability, and communication costs.

11.5.2.2 Federated aggregation protocols

The central coordination mechanism in federated learning allows devices with small, local datasets to collaboratively train a shared model. Client devices perform local training and transmit model updates to a central server, which aggregates them into a refined global model and redistributes it for the next training round. The cyclical procedure decouples learning from centralized data collection, suiting environments where user data is private, bandwidth is constrained, and device participation is sporadic.

The most widely used baseline for this process is Federated Averaging (FedAvg)[29], which has become a canonical algorithm for federated learning (McMahan et al. 2017). In FedAvg, each device trains its local copy of the model using stochastic gradient descent (SGD) on its private data.

Formally, let $\mathcal{D}_k$ denote the local dataset on client k, and let θ_k^t be the parameters of the model on client k at round t. Each client performs E epochs of SGD on its local data, yielding an update θ_k^{t+1}. The central server then aggregates these updates as:

$$\theta^{t+1} = \sum_{k=1}^{C} \frac{n_k}{n} \theta_k^{t+1}$$

where $n_k = |\mathcal{D}_k|$ is the number of samples on device k, $n = \sum_k n_k$ is the total number of samples across participating clients, and C is the number of active devices in the current round.

[29] **FedAvg**: McMahan et al. (2017) averages model weights instead of individual gradients, allowing each client to perform multiple local SGD steps (typically 1–20) before a single upload. This reduces communication by 10–100× compared to distributed SGD, making federated learning viable over mobile networks where upload bandwidth is the binding constraint. The trade-off: more local steps improve communication efficiency but increase divergence on non-IID data, requiring careful tuning per deployment.

11.5.2.3 Straggler mitigation

In a fleet of millions of heterogeneous devices, waiting for every selected client to report its update is impractical. Network latency, battery depletion, or background process contention can cause some devices ("stragglers") to take 10× longer than average. At the level of round semantics, the consequence is direct: the weighted aggregation above cannot proceed until enough updates arrive, so the round duration is set by the slowest devices the server is still waiting on. The scheduling answer to this, over-selection, is a client-scheduling policy and is developed with the rest of client scheduling in section 11.6.1.

This cyclical coordination protocol forms the foundation of federated learning. Figure 11.9 breaks down the Federated Averaging protocol into four phases: client selection (with over-selection to handle stragglers), local training on private data, parameter upload with optional compression, and weighted aggregation that produces the updated global model.

This basic structure introduces a number of design choices and trade-offs. The number of local epochs E impacts the balance between computation and communication: larger E reduces communication frequency but risks divergence if local data distributions vary too much. The selection of participating clients affects convergence stability and fairness. In real-world deployments, not all devices are available at all times, and hardware capabilities may differ substantially, requiring robust participation scheduling and failure tolerance.

11.5.2.4 Federated learning convergence analysis

Whether federated learning converges and how fast it reaches acceptable accuracy are essential questions for system design. Unlike centralized training where convergence depends primarily on learning rate and batch size, federated learning convergence depends on the interplay between communication rounds, local computation, client participation, and data heterogeneity. These factors determine whether a federated deployment will reach target accuracy in hours, days, or never. Section C.3.4 develops the weak-scaling argument for why adding more participating devices grows total work faster than it shrinks wall-clock time, which sets the diminishing return that bounds how much client participation can accelerate convergence here.

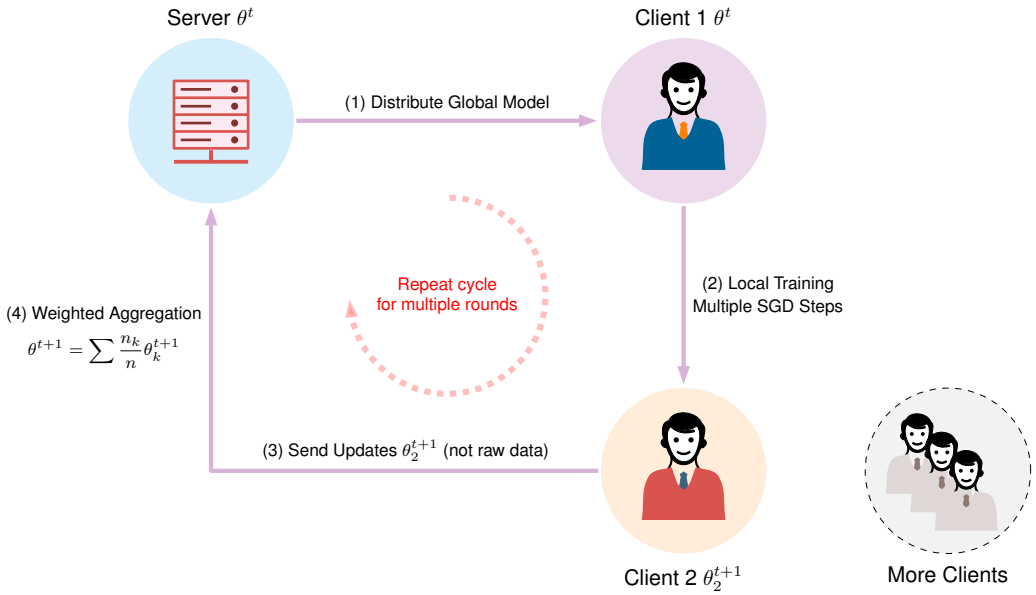


Figure 11.9: Federated Averaging Protocol: The four-phase cycle of federated learning. (1) Server broadcasts global model to participating clients. (2) Clients train locally on private data. (3) Clients upload model updates (gradients or weights). (4) Server aggregates updates into improved global model.

Convergence rate fundamentals A useful systems abstraction is to treat the product of participating clients, local epochs, and communication rounds as the effective amount of federated computation. McMahan et al. (2017) define FedAvg in terms of client participation, local minibatch/epoch work, and repeated communication rounds. For the chapter's systems model, an idealized IID scaling heuristic summarizes the optimality gap as:

$$\mathbb{E}[F(\theta^R) - F(\theta^\star)] \leq \mathcal{O}\left(\frac{1}{\sqrt{CER}}\right)$$

where F is the global objective, θ^R is the model after R rounds, $\theta^\star$ is the reference optimum of that objective, the expectation is over client sampling and stochastic local training, C is the number of clients participating per round, E is the number of local epochs each client performs, and R is the total number of communication rounds. Under this heuristic, convergence improves with the square root of total computation: doubling clients, local epochs, or rounds each provides diminishing returns. To halve the optimality gap, the system must quadruple total computation.

The product CER represents total client-epochs, the fundamental unit of federated computation. A system with 10 clients performing 5 local epochs over 100 rounds ($CER = 5000$) achieves similar convergence to one with 50 clients performing 2 epochs over 50 rounds ($CER = 5000$), assuming identical data distributions. This equivalence enables flexible resource allocation: bandwidth-constrained deployments can compensate with more local computation, while computation-constrained devices can rely on more frequent communication.

Non-IID data impact The federated scaling heuristic assumes IID data across clients, an assumption that rarely holds in practice. Surveys and empirical studies identify non-IID data as a core federated-learning convergence challenge and quantify it with measures such as local dissimilarity, earth mover's distance, or weight divergence (T. Li et al. 2020; Zhao et al. 2018). To reason about this systems effect, let β_{het} denote an abstract heterogeneity penalty that grows as local client objectives drift away from the global objective:

$$\mathbb{E}[F(\theta^R) - F(\theta^\star)] \leq \mathcal{O}\left(\frac{1+\beta_{\text{het}}}{\sqrt{CER}} + \frac{\beta_{\text{het}}^2 E^2}{R}\right)$$

Here, β_{het} is a compact way to represent client drift in the chapter's systems model rather than a universal metric shared by all analyses. When data is close to IID, β_{het} is near zero and the bound reduces toward the standard $\mathcal{O}(1/\sqrt{CER})$ rate. When data is highly heterogeneous, larger β_{het} values significantly slow convergence.

The second term $\frac{\beta_{\text{het}}^2 E^2}{R}$ reveals a critical interaction: more local epochs E amplify the heterogeneity penalty. Each additional local step moves the local model further from the global optimum before aggregation, and these deviations compound across heterogeneous clients. This creates a communication-computation trade-off that depends on data distribution characteristics.

Quantifying heterogeneity in practice Real-world federated deployments exhibit β_{het} values that vary dramatically by application domain. Keyboard prediction across users speaking the same language typically shows $\beta_{\text{het}} \approx 0.3$-$0.8$, as vocabulary and typing patterns vary but share common linguistic structure. Cross-language keyboard prediction increases to $\beta_{\text{het}} \approx 1.5$-$3.0$ due to fundamentally different character distributions and word patterns. Health monitoring with diverse patient populations can reach $\beta_{\text{het}} \approx 2.0$-$5.0$ when physiological baselines vary dramatically across age groups, fitness levels, and medical conditions.

A production federated deployment with 100 clients makes the round-complexity trade-off concrete: 100 total clients in the population, $C = 10$ clients selected per round, $E = 5$ local epochs per round, target optimality gap $\epsilon = 0.01$, and two scenarios comparing IID data ($\beta_{\text{het}} = 0$) vs. moderately non-IID data ($\beta_{\text{het}} = 1.5$).

IID case ($\beta_{\text{het}} = 0$): Using the convergence bound $\epsilon \leq \frac{\sigma}{\sqrt{CER}}$ where σ captures gradient variance (typically $\sigma \approx 1$ for normalized objectives), we solve for required rounds:

$$R_{\text{IID}} \geq \frac{\sigma^2}{C \cdot E \cdot \epsilon^2} = \frac{1}{10 \cdot 5 \cdot 0.0001} = 200 \text{ rounds}$$

With 10 clients per round and 5 local epochs, this represents $200 \times 10 \times 5 = 10{,}000$ total client-epochs of computation.

Non-IID case ($\beta_{\text{het}} = 1.5$): The heterogeneity penalty requires satisfying both terms of the bound. With $E = 5$, the variance term requires 1,250 rounds, while the heterogeneity term dominates:

$$\frac{\beta_{\text{het}}^2 E^2}{R} \leq \epsilon \implies R \geq \frac{\beta_{\text{het}}^2 E^2}{\epsilon} = \frac{2.25 \times 25}{0.01} = 5{,}625 \text{ rounds}$$

This represents a roughly 28.1× increase in communication rounds compared to the IID case. The non-IID scenario requires $5{,}625 \times 10 \times 5 = 281{,}250$ client-epochs, demonstrating how data heterogeneity dominates convergence costs.

The quadratic dependence on E suggests reducing local computation when heterogeneity is high. With $E = 2$ instead of $E = 5$, the heterogeneity term shrinks to 900 rounds while the variance term grows to 3,125, so the variance term now dominates:

$$R \geq \max\left(\frac{(1+\beta_{\text{het}})^2}{C \cdot E \cdot \epsilon^2}, \frac{\beta_{\text{het}}^2 E^2}{\epsilon}\right) = \max\left(\frac{2.5^2}{10 \cdot 2 \cdot 0.0001}, \frac{2.25 \times 4}{0.01}\right) = \max(3{,}125, 900) = 3{,}125 \text{ rounds}$$

This reduces total rounds by 1.80× at the cost of 2.5× more communication per unit of local computation. The total client-epochs become $3{,}125 \times 10 \times 2 = 62{,}500$, a 4.5× reduction from the $E = 5$ non-IID case. This illustrates why adaptive local epoch selection based on estimated data heterogeneity significantly improves federated learning efficiency.

Communication-computation trade-off The interaction between local epochs and communication rounds creates a fundamental design trade-off, visualized in figure 11.10. More local epochs reduce communication frequency (the computational pull) but increase client drift (the statistical penalty), while fewer local epochs maintain tighter synchronization at higher communication cost.

The optimal operating point depends on deployment-specific factors. Communication cost (bandwidth, energy) favors larger E, while convergence speed under heterogeneity favors smaller E. Practical systems often use adaptive strategies that start with small E and increase as the model approaches convergence and client drift diminishes.

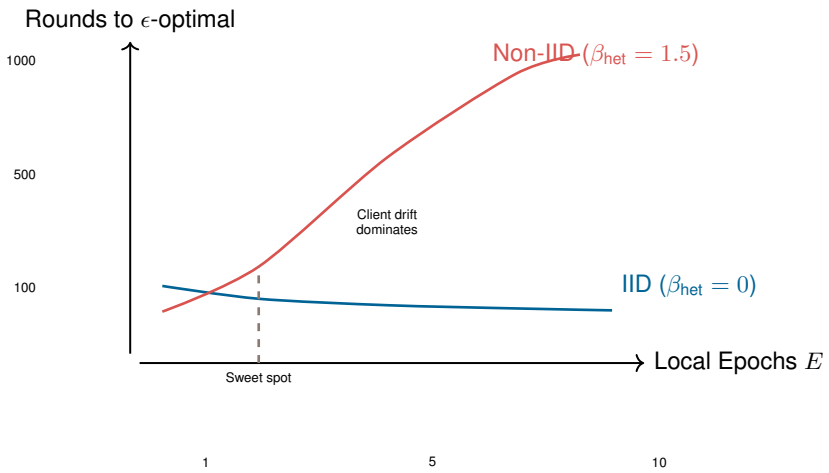


Figure 11.10: Communication-Computation Trade-Off: For IID data, increasing local epochs consistently reduces total communication with minimal penalty. For non-IID data, client drift causes convergence degradation beyond an optimal point (typically $E \in [2, 5]$), making aggressive local computation counterproductive. System designers must estimate data heterogeneity to select appropriate operating points.

When federated learning works Federated learning achieves practical convergence only when several deployment conditions are favorable, a theme emphasized in surveys of federated-learning systems and open problems (Kairouz and McMahan 2021):

- Heterogeneity must remain bounded enough that client drift does not dominate training time. When populations are highly heterogeneous, clustering clients into more homogeneous groups or using personalization may be preferable to one global model.
- Client participation must be sufficient to provide stable aggregate updates. Very low participation makes gradient estimates noisy and exacerbates selection bias.
- Each participating client needs enough local data for meaningful update computation. Tiny local batches can contribute high-variance updates that destabilize training.

When these conditions are violated, alternative approaches become necessary. Severe heterogeneity suggests hierarchical federated learning with regional aggregation or fully personalized models without global aggregation. Very low participation rates indicate the need for asynchronous protocols that do not require synchronized rounds. Clients with minimal local data benefit from few-shot adaptation techniques rather than gradient-based training.

These mathematical aggregation protocols prove that decentralized learning is theoretically possible. Executing federated averaging across ten reliable nodes in a lab is trivial; executing it across ten million smartphones dropping on and off cellular networks is a different engineering problem. Robust federated systems must translate the algorithm into orchestration, scheduling, and failure-handling mechanisms that assume unreliable clients from the start.

11.6 Federated Learning: Systems at Scale

In a textbook algorithm, all 100 edge devices compute their updates perfectly and report back to the server exactly at the same time. In reality, a federated learning round spanning a million smartphones must deal with devices overheating, losing Wi-Fi, entering power-saving mode, or powering off. Building federated systems at scale means engineering an orchestrator that tolerates asynchronous dropouts and extreme stragglers as standard operating conditions.

11.6.1 Client scheduling

Federated learning operates under the assumption that clients, or devices that hold local data, periodically become available for participation in training rounds. In real-world systems, client availability is intermittent and variable. Devices may be turned off, disconnected from power, lacking

network access, or otherwise unable to participate at any given time. As a result, client scheduling plays a central role in the effectiveness and efficiency of distributed learning.

At a baseline level, federated ML systems define eligibility criteria for participation. Devices must meet minimum requirements such as being plugged in, connected to Wi-Fi, and idle, to avoid interfering with user experience or depleting battery resources. These criteria determine which subset of the total population is considered "available" for any given training round.

Beyond these operational filters, devices also differ in their hardware capabilities, data availability, and network conditions. Some smartphones contain many recent examples relevant to the current task, while others have outdated or irrelevant data. Network bandwidth and upload speed may vary widely depending on geography and carrier infrastructure. As a result, selecting clients at random can lead to poor coverage of the underlying data distribution and unstable model convergence.

Availability-driven selection introduces participation bias. Clients with favorable conditions are more likely to participate repeatedly. These favorable conditions include frequent charging, high-end hardware, and consistent connectivity. Meanwhile, others are systematically underrepresented. This can skew the resulting model toward behaviors and preferences of a privileged subset of the population, raising both fairness and generalization concerns.

The severity of participation bias becomes apparent when examining real deployment statistics. Studies of federated learning deployments show that the most active 10 percent of devices can contribute to over 50 percent of training rounds, while the bottom 50 percent of devices may never participate at all. This creates a feedback loop: models become more strongly optimized for users with high-end devices and stable connectivity, potentially degrading performance for resource-constrained users who need adaptation the most. A keyboard prediction model might become biased toward the typing patterns of users with flagship phones who charge overnight, missing important linguistic variations from users with budget devices or irregular charging patterns.

To address these challenges, systems must balance scheduling efficiency with client diversity. A key approach involves using stratified or quota-based sampling to ensure representative client participation across different groups. Some systems implement "fairness budgets" that track cumulative participation and actively prioritize underrepresented devices when they become available. Others use importance sampling techniques to reweight contributions based on estimated population statistics rather than raw participation rates. For instance, asynchronous buffer-based techniques allow participating clients to contribute model updates independently, without requiring synchronized coordination in every round (Nguyen et al. 2021). This model has been extended to incorporate staleness awareness (Rodio and Neglia 2024) and fairness mechanisms (Ma et al. 2024), preventing bias from over-active clients who might otherwise dominate the training process.

These fairness mechanisms sit alongside adaptive client selection strategies. Federated ML systems can prioritize clients with underrepresented data types, target geographies or demographics that are less frequently sampled, and use historical participation data to enforce fairness constraints. Predictive modeling can anticipate later client availability or success rates, improving training throughput.

Selected clients perform one or more local training steps on their private data and transmit their model updates to a central server. These updates are aggregated to form a new global model. Typically, this aggregation is weighted, where the contributions of each client are scaled, for example, by the number of local examples used during training, before averaging. This ensures that clients with more representative or larger datasets exert proportional influence on the global model.



Over-selection closes the round on the first K updates.

Scheduling must also bound how long a round waits on its slowest participants. As established under section 11.5.2.3, stragglers can take $10\times$ longer than the median device, and the round cannot aggregate until enough updates arrive.

To prevent the global model update from stalling, production FL systems employ **Over-Selection**. The server selects a candidate pool size $K_{\text{candidates}}$ larger than the target number of updates K_{target} (typically $K_{\text{candidates}} \approx 1.3 \times K_{\text{target}}$). The server aggregates updates from the first K_{target} responders and discards the rest. This approach bounds the round duration by the speed of the K_{target}-th fastest device rather than the absolute slowest, dramatically accelerating convergence wall-clock time.

These scheduling decisions directly impact convergence rate, model generalization, energy consumption, and overall user experience. Poor scheduling results in excessive stragglers, overfitting to narrow client segments, or wasted computation. Client scheduling is therefore a core component of

system design in federated learning, demanding both algorithmic insight and infrastructure-level coordination.

Checkpoint 11.3: Federated system design

You are architecting a federated learning system for a fleet of 10 million mobile devices. The data is highly non-IID (users have distinct, clustered typing patterns), and the network environment is constrained (1-10 Mbps). Before choosing optimizations for this scenario, classify the local bottlenecks and state which system constraint dominates each decision.

Consider three design decisions:

1. **Aggregation**: Select plain FedAvg, reduced local epochs, or personalization/clustering for the non-IID population, and justify how the choice controls client drift.
2. **Privacy**: Decide whether the deployment needs formal differential-privacy noise or whether secure aggregation and data minimization are sufficient for the stated threat model; explain the expected convergence-time cost qualitatively.
3. **Communication**: Choose between gradient quantization (e.g., 8-bit) and structured sparsity under constrained bandwidth, and justify the bandwidth/accuracy trade-off.

11.6.2 Bandwidth-aware update compression

Update compression is a bottleneck diagnosis before it is a technique choice: if each round transmits full model weights or gradients, mobile bandwidth and battery budgets determine convergence before the optimizer does[30]. The design decision is which information can be quantized, sparsified, or kept local while preserving enough gradient fidelity for FedAvg to converge.

The compression decision separates into three levers, each changing a different part of the communication bottleneck: model compression reduces bits per update, selective sharing reduces which parameters travel, and architectural partitioning keeps private state local. Model compression methods aim to reduce the size of transmitted updates through quantization[31], sparsification, or subsampling, as figure 11.12 illustrates. Instead of sending full-precision gradients, a client transmits 8-bit quantized updates or communicates only the top-k gradient elements[32] with highest magnitude.

Selective update sharing further reduces communication by transmitting only subsets of model parameters or updates. In layer-wise selective sharing, clients update only certain layers, typically the final classifier or adapter modules, while keeping the majority of the backbone frozen. This reduces both upload cost and the risk of overfitting shared representations to nonrepresentative client data.

Split models and architectural partitioning divide the model into a shared global component and a private local component. Clients train and maintain their private modules independently while synchronizing only the shared parts with the server. This allows for user-specific personalization with minimal communication and privacy leakage.

All of these approaches operate within the FedAvg aggregation protocol (section 11.5.2.2): compression changes what each client transmits, not how the server combines updates. While figure 11.11 illustrates the fundamental trade-off between local computation and network bandwidth, other communication-efficient updates introduce their own trade-offs. Compression may degrade gradient fidelity, selective updates can limit model capacity, and split architectures may complicate coordination. As a result, effective federated learning requires careful balancing of bandwidth constraints, privacy concerns, and convergence dynamics, a balance that depends heavily on the capabilities and variability of the client population. Section D.5 works the compression break-even threshold through a worked example, quantifying when the bandwidth saved by a given compression ratio outweighs the accuracy lost to reduced gradient fidelity.

Once the system chooses how much local work to perform, update compression determines how much each round must transmit.

30 **Wireless Upload Asymmetry**: LTE uploads average 5–10 Mbps vs. 50+ Mbps downloads, creating the asymmetric bottleneck that dominates federated learning design. Transmitting a 50 MB model update consumes approximately 100 mAh (2–3 percent of battery) and takes 40–80 seconds over LTE. Low-power protocols are far worse: LoRaWAN maxes at 50 kbps with 1 percent duty cycle, making uncompressed updates physically impossible. This asymmetry is why gradient compression is not optional but architecturally required.

31 **Gradient Quantization**: Converts FP32 gradients to lower precision (INT8, INT4, or 1-bit) before transmission. Techniques like QSGD and signSGD show that quantized or sign-only updates can greatly reduce communication while relying on stochastic quantization or error compensation to preserve convergence behavior (Alistarh et al. 2017; Bernstein et al. 2018). The achieved byte reduction and accuracy impact depend on the optimizer, model, data heterogeneity, and residual-handling scheme.

32 **Gradient Sparsification**: Transmits only the top 1–10 percent of gradients by magnitude, exploiting the observation that most gradient elements are near-zero and contribute minimally to convergence. Local gradient accumulation stores the untransmitted residuals until they grow large enough to send, achieving 10–100× compression while preserving training quality, a pattern used in sparsified SGD and deep gradient compression (Stich et al. 2018; Lin et al. 2018). The trade-off: aggressive sparsification can bias updates toward frequently active parameters, potentially reducing personalization for rare user patterns.

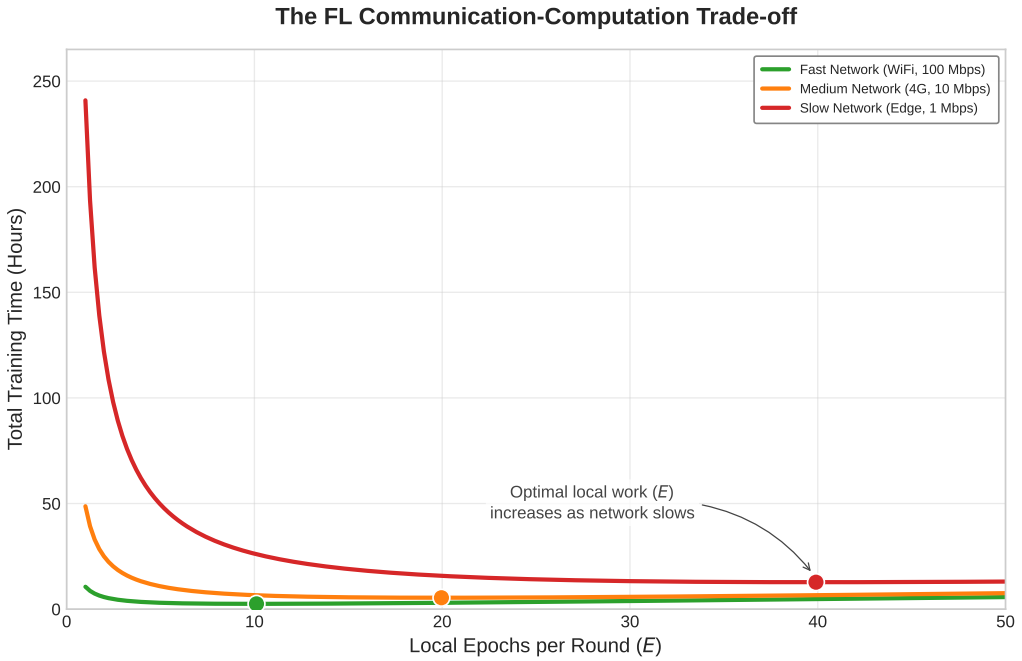


Figure 11.11: The Communication-Computation Trade-Off in Federated Learning: As network bandwidth decreases (Fast to Slow), the optimal number of local epochs shifts rightward to amortize the high cost of communication over more computation. However, excessive local computation eventually increases total time due to model drift (requiring more global rounds to converge).

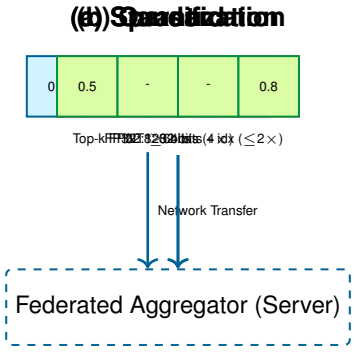


Figure 11.12: Gradient Compression Techniques: (a) Standard updates transmit full-precision values. (b) Quantization maps values to low-precision buckets (e.g., FP32 to INT8), reducing bandwidth. (c) Sparsification transmits only the most significant (Top-k) gradients, exploiting that many updates are near-zero.

11.6.3 Federated personalization

While compression and communication strategies improve scalability, they do not address an important limitation of the global federated learning paradigm, its inability to capture user-specific variation. In real-world deployments, devices often observe distinct and heterogeneous data distributions. A one-size-fits-all global model may underperform when applied uniformly across diverse users. This motivates the need for personalized federated learning, where local models are adapted to user-specific data without compromising the benefits of global coordination.

Let θ_k denote the model parameters on client k, and θ_{global} the aggregated global model. For K clients, traditional FL seeks to minimize a global objective:

$$\min_{\theta} \sum_{k=1}^{K} w_k \mathcal{L}_k(\theta)$$

where $\mathcal{L}_k(\theta)$ is the local loss on client k, and w_k is a weighting factor (e.g., proportional to local dataset size). However, this formulation assumes that a single model θ can serve all users well. In practice, local loss landscapes $\mathcal{L}_k$ often differ significantly across clients, reflecting non-IID data distributions and varying task requirements.

Personalization modifies this objective to allow each client to maintain its own adapted parameters θ_k, optimized with respect to both the global model and local data:

$$\min_{\theta_1,\ldots,\theta_K} \sum_{k=1}^{K} \left(\mathcal{L}_k(\theta_k) + \lambda_{\text{reg}} \cdot \mathcal{R}(\theta_k, \theta_{\text{global}}) \right)$$

Here, $\mathcal{R}$ is a regularization term that penalizes deviation from the global model, and λ_{reg} controls the strength of this penalty. This formulation allows local models to deviate as needed, while still benefiting from global coordination.

Real-world use cases illustrate the importance of this approach. Consider a wearable health monitor that tracks physiological signals to classify physical activities. While a global model may perform reasonably well across the population, individual users exhibit unique motion patterns, gait signatures, or sensor placements. Personalized finetuning of the final classification layer or low-rank adapters allows improved accuracy, particularly for rare or user-specific classes.

Several personalization strategies have emerged to address the trade-offs between compute overhead, privacy, and adaptation speed. One widely used approach is local finetuning, in which each client downloads the latest global model and performs a small number of gradient steps using its private data. While this method is simple and preserves privacy, it may yield suboptimal results when the global model is poorly aligned with the client's data distribution or when the local dataset is extremely limited.

Another effective technique involves personalization layers, where the model is partitioned into a shared backbone and a lightweight, client-specific head, typically the final classification layer (Arivazhagan et al. 2019). Only the head is updated on-device, reducing memory usage and training time. This approach is particularly well-suited for scenarios in which the primary variation across clients lies in output categories or decision boundaries.

Clustered federated learning offers an alternative by grouping clients according to similarities in their data or performance characteristics, and training separate models for each cluster. This strategy can enhance accuracy within homogeneous subpopulations but introduces additional system complexity and may require exchanging metadata to determine group membership.

Finally, meta-learning approaches, such as Model-Agnostic Meta-Learning (MAML)[33], aim to produce a global model initialization that can be quickly adapted to new tasks with just a few local updates (Finn et al. 2017). This technique is especially useful when clients have limited data or operate in environments with frequent distributional shifts.

Each of these strategies reflects a different point in the personalization trade-off space. Examine table 11.6 to see how compute overhead, privacy guarantees, and adaptation latency vary across local finetuning, personalization layers, clustered federated learning, and meta-learning approaches.

[33] **MAML**: Finn et al. (2017) optimizes for a model initialization from which one or a few gradient steps can yield good performance on a new task, with experiments including one-shot and few-shot settings. Meta-training is more expensive than ordinary finetuning because it differentiates through the adaptation process, but deployment adaptation can be limited to a small number of local updates. This makes MAML relevant for edge scenarios where inference compute is cheap but training data is scarce.

Table 11.6: Personalization Trade-Offs: Federated learning strategies balance personalization with system costs, impacting compute overhead, privacy preservation, and adaptation speed for diverse client populations. This table summarizes how local finetuning, personalization layers, clustered learning, and meta-learning each navigate this trade-off space, enabling tailored models while considering practical deployment constraints.

Strategy	Personalization Mechanism	Compute Overhead	Privacy Preservation	Adaptation Speed
Local Finetuning	Gradient descent on local loss postaggregation	Low to Moderate	High (no data sharing)	Fast (few steps)

Strategy	Personalization Mechanism	Compute Overhead	Privacy Preservation	Adaptation Speed
Personalization Layers	Split model: shared base + user-specific head	Moderate	High	Fast (train small head)
Clustered FL	Group clients by data similarity, train per group	Moderate to High	Medium (group metadata)	Medium
Meta-Learning	Train for fast adaptation across tasks/devices	High (meta-objective)	High	Very Fast (few-shot)

Selecting the appropriate personalization method depends on deployment constraints, data characteristics, and the desired balance between accuracy, privacy, and computational efficiency. Figure 11.13 contrasts three architectural approaches: full fine-tuning offers maximum expressivity at high compute cost, head-only adaptation provides low-cost but limited adaptation, and adapter-based methods balance efficiency with deep representation adaptation through small trainable modules.

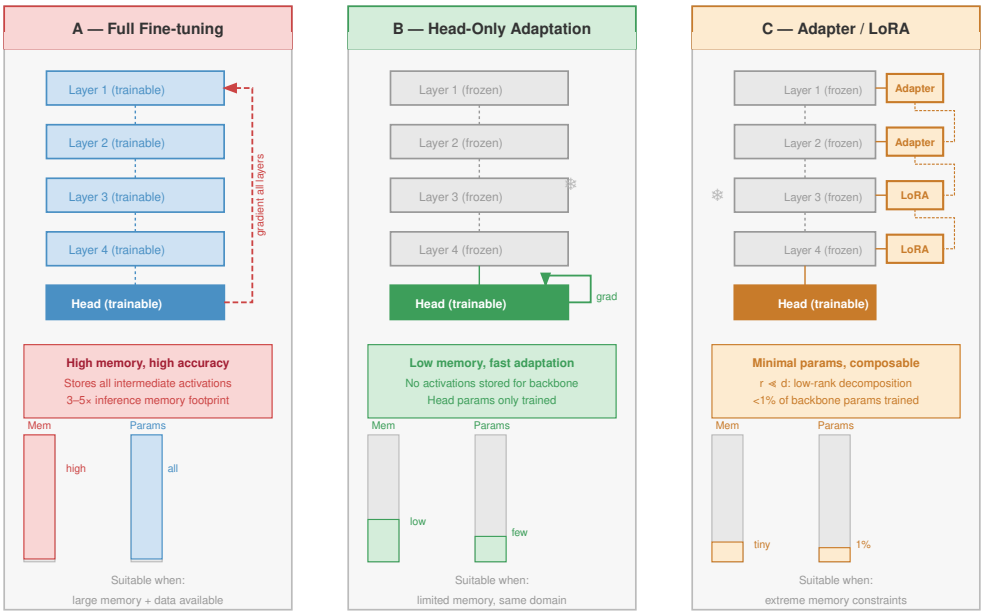


Figure 11.13: Federated Personalization Architectures: Architectural strategies for adapting global models to local data. (a) Full Fine-tuning updates all model parameters, offering maximum expressivity but high compute cost. (b) Head-Only Adaptation updates only the final classifier layers while keeping the feature extractor frozen, suitable for resource-constrained devices. (c) Adapter-Based Learning (for example, LoRA) inserts small trainable modules into a frozen backbone, balancing efficiency with the ability to adapt deep representations.

11.6.4 Federated privacy

The critical distinction figure 11.13 reveals is the trade-off between adaptation expressivity and compute cost: full fine-tuning updates every parameter but demands resources unavailable on most edge devices, while adapter-based methods achieve deep representation adaptation at a fraction of the memory and compute budget. While federated learning is often motivated by privacy concerns, as it involves keeping raw data localized instead of transmitting it to a central server, the paradigm introduces its own set of security and privacy risks. Although devices do not share their raw data, the transmitted model updates (such as gradients or weight changes) can inadvertently leak information

34 **Model Inversion Attack**: Reconstructs training data from model outputs or gradients by optimizing inputs that maximize confidence scores (Fredrikson et al. 2015; Zhu et al. 2019). In federated learning, gradient updates can leak more information than model outputs alone because they expose training-step information directly. Defenses such as gradient perturbation, secure aggregation, and differential privacy add protocol, utility, communication, or compute trade-offs whose magnitude depends on the threat model and deployment.

about the underlying private data. Techniques such as model inversion attacks[34] and membership inference attacks[35] demonstrate that adversaries may partially reconstruct or infer properties of local datasets by analyzing these updates.

To mitigate such risks, federated ML systems can employ protective measures. **Secure aggregation**[36] protocols ensure that individual model updates are encrypted and aggregated in a way that the server only observes the combined result, not any individual client's contribution. **Differential privacy**[37] techniques inject carefully calibrated noise into updates to mathematically bound the information that can be inferred about any single client's data.

While these techniques enhance privacy, they introduce additional system complexity and trade-offs between model utility, communication cost, and robustness. Figure 11.14 illustrates the secure aggregation protocol, showing how pairs of clients exchange shared random masks that cancel out during server-side summation, revealing only the aggregate gradient without exposing individual contributions.

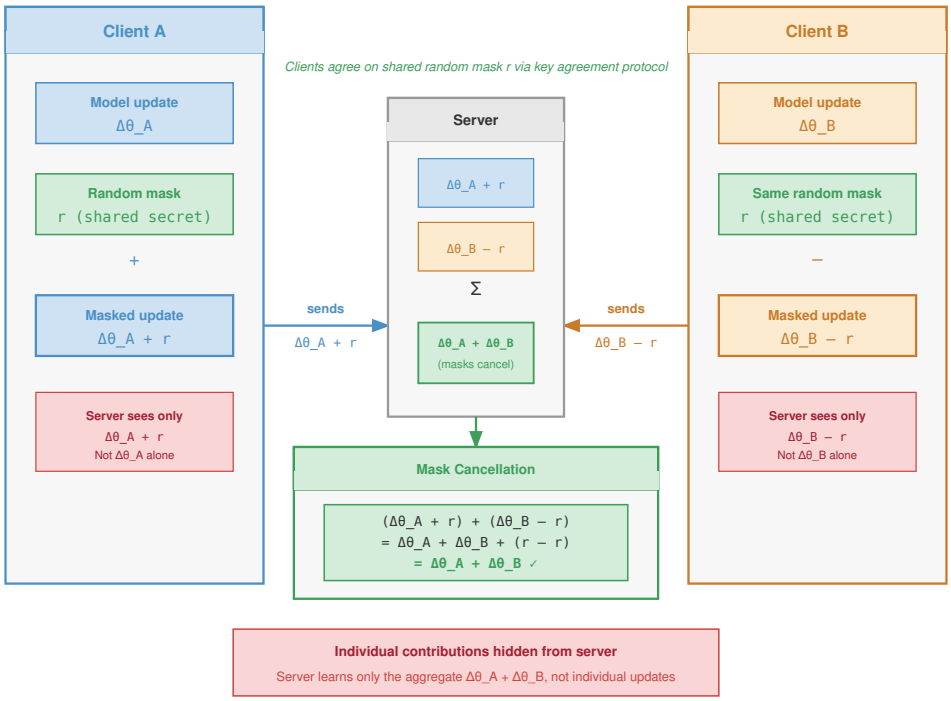


Figure 11.14: Secure Aggregation Protocol: A simplified view of how cryptographic masking protects individual updates. Pairs of clients agree on shared random masks that are added by one and subtracted by the other. The central server sums the masked updates; the masks mathematically cancel out in the aggregate, revealing the global update sum without ever exposing the raw value of any single client's contribution.

11.6.5 Large-scale device orchestration

That algebraic cancellation is what makes secure aggregation composable with differential privacy for layered defense: the protocol hides individual contributions first, then privacy noise can bound what the aggregate still reveals. Federated learning transforms machine learning into a massive distributed systems challenge that extends far beyond traditional algorithmic considerations. Coordinating thousands or millions of heterogeneous devices with intermittent connectivity requires sophisticated distributed systems protocols that handle Byzantine failures [clients sending arbitrary or malicious updates], network partitions [temporary loss of connectivity between participants], and communication efficiency at unprecedented scale. These challenges fundamentally differ from

35 **Membership Inference Attack**: Determines whether specific data points were used to train a model by exploiting the confidence gap between training data (high confidence, low loss) and unseen data (Shokri et al. 2017). Published attacks can perform substantially above chance on overfitted models. In federated learning on personal data, this means an adversary could infer whether a specific patient's records trained a health model, making regularization and differential privacy architecturally important whenever the threat model includes training-data disclosure.

36 **Secure Aggregation**: Cryptographic protocol where client pairs exchange random masks that cancel in the server-side sum, revealing only the aggregate gradient without exposing individual contributions (Bonawitz et al. 2017). Production-scale federated systems such as Gboard combine secure aggregation with eligibility checks, drop-out handling, and round scheduling (Bonawitz et al. 2019). The communication expansion and minimum-participation requirements depend on protocol parameters and the security model, so aggregation systems must wait for enough eligible devices before releasing a round.

37 **Differential Privacy (DP)**: A mathematical framework that bounds information leakage by adding calibrated noise to computations. In edge learning, that noise becomes a systems cost because noisier updates usually require more rounds or more data to reach the same utility. Chapter 13 develops the formal privacy-budget machinery.

the controlled environments of data center distributed training, where high-bandwidth networks and reliable infrastructure enable straightforward coordination protocols.

11.6.5.1 Network and bandwidth optimization

The communication bottleneck represents the primary scalability constraint in federated learning systems. Quantifying the actual transfer requirements exposes the design constraints on model architectures, update compression strategies, and client participation policies that determine system viability.

The federated communication hierarchy reveals the severe bandwidth constraints under which distributed learning must operate. Full model synchronization can require tens to hundreds of megabytes per training round for common deep models, which is a poor fit for intermittent mobile uplinks. Federated-optimization work therefore studies communication reduction through structured updates, quantization, sparsification, and selective transmission (Konečný et al. 2016; McMahan et al. 2017). Practical deployments often push this further by transmitting only adapters, heads, or compressed update summaries rather than complete model states. Communication frequency introduces a critical trade-off between model update freshness, where more frequent updates enable faster adaptation to changing conditions, and network efficiency constraints that limit sustainable bandwidth consumption.

Network infrastructure constraints directly impact participation rates and overall system viability. Mobile upload capacity varies sharply by geography, carrier, radio generation, congestion, and whether the device remains idle long enough to complete the transfer. A multi-megabyte update may be tolerable for a high-end device on a stable connection but impractical for a low-end device on a congested uplink. This variance in network capability necessitates adaptive communication strategies that optimize for lowest-common-denominator connectivity while enabling high-capability devices to contribute more effectively.

The relationship between communication requirements and participation rates exhibits sharp threshold effects. Large model transfers sharply reduce sustained client participation because devices must remain idle, connected, sufficiently charged, and willing to spend upload bandwidth for the whole transfer window. Keeping updates small, often through adapter-only sharing or aggressive compression, increases the eligible device pool and shortens round duration. This communication efficiency directly translates to model quality improvements: higher participation rates provide better statistical diversity and more robust gradient estimates for global model updates. The compression techniques that make these small updates possible (gradient quantization, sparsification, and top-k selection with error accumulation) are the same ones established in section 11.6.2; here they serve the participation argument rather than the bandwidth argument, because shrinking the update is what widens the eligible device pool.

11.6.5.2 Asynchronous device synchronization

Federated learning operates at the complex intersection of distributed systems and machine learning, inheriting fundamental challenges from both domains while introducing unique complications that arise from the mobile, heterogeneous, and unreliable nature of edge devices. Federated learning must contend with Byzantine fault tolerance requirements that extend beyond typical distributed systems challenges. Device failures occur frequently as clients crash, lose power, or disconnect during training rounds due to battery depletion or network connectivity issues, far more common than server failures in traditional distributed training. Malicious updates present security concerns as adversarial clients can provide corrupted gradients deliberately designed to degrade global model performance or extract private information from the aggregation process. Robust aggregation protocols implementing Byzantine-resilient averaging preserve useful model updates despite compromised or unreliable participants, though these protocols introduce significant computational overhead. The coordination problem is not consensus in the Paxos or Raft sense; it is deciding which client updates are eligible, how stale updates are weighted, and which aggregation rule can tolerate faulty participants without letting them steer the global model.

Network partitions pose particularly acute challenges for federated coordination protocols. Unlike traditional distributed systems operating within reliable data center networks, federated learning must gracefully handle prolonged client disconnection events where devices may remain offline

for hours or days while traveling, in poor coverage areas, or simply powered down. Asynchronous coordination protocols enable continued training progress despite missing participants, but must carefully balance staleness (accepting potentially outdated contributions) against freshness (prioritizing recent but potentially sparse updates).

Fault recovery and resilience strategies form an essential layer of federated learning infrastructure. Checkpoint synchronization through periodic global model snapshots enables recovery from server failures and provides rollback points when corrupted training rounds are detected, though checkpointing large models across millions of devices introduces substantial storage and communication overhead. Partial update handling ensures systems gracefully handle incomplete training rounds when significant subsets of clients fail or disconnect mid-training, requiring careful weighting strategies to prevent bias toward more reliable device cohorts. State reconciliation protocols enable clients rejoining after extended offline periods, potentially days or weeks, to efficiently resynchronize with the current global model while minimizing communication overhead that could overwhelm bandwidth-constrained devices. Dynamic load balancing addresses uneven client availability patterns that create computational hotspots, requiring intelligent load redistribution across available participants to maintain training throughput despite time-varying participation rates.

The asynchronous nature of federated coordination introduces additional complexity in maintaining training convergence guarantees. Traditional synchronous training assumes all participants complete each round, but federated systems must handle stragglers[38] and dropouts gracefully.

[38] **Straggler Problem**: In data center training, stragglers slow synchronous progress; in federated learning, the problem is amplified by heterogeneous hardware, intermittent connectivity, and eligibility windows (Bonawitz et al. 2019; Kairouz and McMahan 2021). Solutions like asynchronous aggregation and bounded staleness (ignoring updates older than k rounds) improve throughput but introduce a fairness trade-off: aggressive straggler mitigation biases models toward users with high-end devices, potentially degrading performance for the most constrained users who need personalization most.

11.6.5.3 Managing million-device heterogeneity

Million-device orchestration is a tiering decision: the system must decide which clients can train, which can only report lightweight updates, and how much bias that selection introduces. The difficulty is that hardware capabilities, network conditions, data distributions, and availability patterns vary simultaneously, challenging traditional distributed machine learning assumptions about homogeneous participants operating under similar conditions.

Real-world federated learning deployments face multi-dimensional device heterogeneity that creates extreme variation across every system dimension. Computational variation spans about 1,166.7× differences in processing power between flagship smartphones running at 35 TOPS and IoT microcontrollers operating at just 0.03 TOPS, fundamentally limiting what models can train on different device tiers. Memory constraints exhibit even more dramatic differences in available RAM across device categories, up to roughly 65,536× between 256 KB microcontrollers and 16 GB premium smartphones, determining whether devices can perform any local training at all or must rely purely on inference. Energy limitations force training sessions to be carefully scheduled around charging patterns, thermal constraints, and battery preservation requirements: background adaptation often targets roughly 500–1000 mW, even though phones may sustain 2–3 W for foreground ML workloads and burst higher briefly. Network diversity introduces orders-of-magnitude performance differences as Wi-Fi, 4G, 5G, and satellite connectivity exhibit vastly different bandwidth (ranging from 1 Mbps to 1 Gbps), latency (10 ms to 600 ms), and reliability characteristics that determine feasible update frequencies and compression requirements.

Adaptive coordination protocols address this heterogeneity through sophisticated tiered participation strategies that optimize resource utilization across the device spectrum. High-capability devices such as flagship smartphones can perform complex local training with large batch sizes and multiple epochs, while resource-constrained IoT devices contribute through lightweight updates, specialized subtasks, or even simple data aggregation. This creates a natural computational hierarchy where powerful devices act as "super-peers" performing disproportionate computation, while edge devices contribute specialized local knowledge and coverage.

The scale challenges extend far beyond device heterogeneity to fundamental coordination overhead limitations. Traditional distributed consensus algorithms such as Raft or PBFT are designed for dozens of nodes in controlled environments, but federated learning requires coordination among millions of participants across unreliable networks. This necessitates hierarchical coordination architectures where regional aggregation servers reduce communication overhead by performing local consensus before contributing to global aggregation. Edge computing infrastructure provides natural hierarchical coordination points, enabling federated learning systems to use existing content delivery networks (CDNs) and mobile edge computing (MEC) deployments for efficient gradient aggregation.

Federated systems can implement sophisticated client selection strategies that balance statistical diversity with practical constraints. Random sampling ensures unbiased representation but may select many low-capability devices, while capability-based selection improves training efficiency but risks statistical bias. Hybrid approaches use stratified sampling across device tiers, ensuring both statistical representativeness and computational efficiency. These selection strategies must also consider temporal patterns: office workers' devices may be available during specific hours, while IoT sensors provide continuous but limited computational resources.

With robust orchestration managing the heterogeneity and intermittency of millions of devices, the individual components for decentralized AI are in place. Each pillar so far has been treated as a discrete capability a device invokes, but a deployed edge model does not adapt once and stop; it keeps learning for the lifetime of the deployment, which is where the three pillars stop being separable.

11.7 Continual Adaptation Under Edge Budgets

Continual adaptation on edge devices has to fit inside power, memory, and communication budgets that leave little room for waste. The model adaptation, data efficiency, and federated coordination pillars each attack one slice of that budget, but a model that keeps learning after deployment must respect all of them at once. The human brain operates at approximately 20 watts while continuously learning from limited supervision without catastrophic forgetting[39]. This comparison does not make biology a deployment recipe; it explains why edge systems keep returning to sparsity, replay, and opportunistic updates.

Biological analogies are useful only when they clarify engineering choices. Sparse representations mirror the selective parameter updates that keep mobile adaptation low cost. Replay buffers parallel memory-consolidation mechanisms that stabilize continual learning. Local-global coordination echoes the need for individual adaptation without abandoning population-level improvement. The point is to recognize why the same constraints keep producing similar design patterns.

For edge systems, the useful comparison is not the neuron count; it is the budget discipline. A 20 W learning system, with roughly 10 W available for active learning and memory consolidation, sits in the same order of magnitude as the power a mobile device can allocate during charging. Sparse, distributed representations reduce memory traffic because only 1–2 percent of neurons activate during a cognitive task. Few-shot learning and continual consolidation point to the same systems target: adapt from little local evidence without overwriting the representation that still serves the original task. Hierarchical processing adds the final lesson, because reusable features reduce the amount of work each new local context must perform.

Two implementation pressures follow from that comparison. Sparse representations map naturally to selective parameter updates and pruned architectures, which keep mobile adaptation within memory and energy limits. Event-driven processing maps to opportunistic scheduling, where the device spends energy on learning only when new input, idle time, power state, and thermal headroom make the update worth running. The analogy is useful only because it sharpens these engineering constraints.

11.7.1 Unlabeled data exploitation strategies

Unlabeled sensor streams turn the efficiency argument into a storage and scheduling decision. Mobile devices continuously collect visual data from cameras, temporal patterns from accelerometers, spatial patterns from GPS, and interaction patterns from touchscreen usage, all of which can support self-supervised learning. The engineering question is which traces encode stable structure and which should expire before they crowd out model state, replay buffers, or user-facing storage.

The scale of mobile sensor data makes that decision concrete across four common stream types:

- Visual streams from cameras operating at 30 frames per second provide approximately 2.6 million frames daily, offering abundant data for contrastive learning approaches that learn visual representations by comparing augmented versions of the same image[40].
- Motion streams from accelerometers sampling at 100 Hz generate 8.6 million data points daily, capturing temporal patterns suitable for learning representations of human activities and device movement.

39 **Brain Power Efficiency**: At 20 W, the brain processes approximately 10^{15} operations per second, equivalent to about 50 TOPS/W under this coarse operation-counting model. Compared with the 1–5 TOPS/W mobile-NPU range used earlier in this chapter, that is roughly 10–50× higher energy efficiency, though biological synaptic events and digital operations are only an approximate analogy. This efficiency derives from extreme sparsity (only 1–2 percent of 86 billion neurons active simultaneously) and event-driven processing that activates computation only on input change. These two principles directly inform edge ML design: sparse activation patterns and opportunistic training scheduling are not heuristics but attempts to approach biological efficiency limits.

40 **Mobile Sensor Data Scale**: A typical smartphone generates 2–4 GB of sensor data daily: cameras (1–2 GB), accelerometers (approximately 50 MB), GPS (approximately 10 MB), and touch events (approximately 5 MB). Contrastive learning can extract useful representations from a small retained sample of this data, making self-supervised on-device learning feasible without labels or cloud processing. The constraint becomes storage policy, not data availability: deciding what to retain and what to discard before memory fills.

- Location traces from GPS sensors enable spatial representation learning and behavioral prediction by capturing movement patterns and frequently visited locations without requiring explicit labels.
- Interaction patterns from touch events, typing dynamics, and app usage sequences create rich behavioral embeddings that reveal user preferences and habits, enabling personalized model adaptation without manual annotation.

Contrastive learning[41] from temporal correlations offers particularly promising opportunities for exploiting this sensor data. Consecutive frames from mobile cameras or augmented views of the same frame naturally provide positive pairs for visual representation learning. Negative examples come from different frames, other devices' samples, or replay/memory queues that represent dissimilar scenes.

The biological inspiration extends to continual learning without forgetting. Brains continuously integrate new experiences while retaining decades of memories through mechanisms like synaptic consolidation and replay. On-device systems must implement analogous mechanisms: elastic weight consolidation[42] (Kirkpatrick et al. 2017) prevents catastrophic forgetting by protecting weights important for previous tasks, experience replay maintains stability during adaptation by interleaving new training with replayed examples from previous tasks, and progressive neural architectures allocate new capacity for new tasks rather than forcing all knowledge into fixed-capacity networks (Rusu et al. 2016). That last option is attractive when interference is severe, but it consumes additional memory and therefore fits only devices with enough spare capacity.

41 | **Contrastive Learning**: Self-supervised technique that learns representations by distinguishing similar (positive) from dissimilar (negative) pairs without labels. SimCLR (T. Chen et al. 2020) demonstrates that large batches of augmented views can learn strong visual representations without human labels. For edge devices, the transferable systems idea is to use unlabeled sensor streams for representation learning or pretraining, then spend scarce labels only on lightweight local adaptation; the exact label savings depend on the task, retention policy, and compute budget.

42 | **EWC (Elastic Weight Consolidation)**: Estimates per-parameter importance for previous tasks using the Fisher Information Matrix, then penalizes changes to important parameters during new learning. This adds only 10–15 percent computation per training step but requires storing one importance score per parameter (4 bytes each), roughly doubling weight memory. The trade-off vs. replay buffers is explicit: EWC trades memory proportional to model size for memory proportional to data history, favoring large-model-small-data edge scenarios.

11.7.2 Lifelong adaptation without forgetting

Real-world on-device deployment demands continual adaptation to changing environments, user behavior, and task requirements. This presents the fundamental challenge of the stability-plasticity trade-off: models must remain *stable* enough to preserve existing knowledge while *plastic* enough to learn new patterns.

Continual learning on edge devices faces several interconnected challenges that compound the difficulty of distributed adaptation:

- Catastrophic forgetting occurs when new learning overwrites previously acquired knowledge, causing models to lose performance on earlier tasks as they adapt to new ones. The problem is particularly severe when devices cannot access historical training data.
- Task interference emerges when multiple learning objectives compete for limited model capacity, forcing difficult trade-offs between different capabilities that the model must maintain simultaneously.
- Data distribution shift manifests as deployment environments differ significantly from training conditions, requiring models to adapt to new patterns while maintaining performance on the original distribution.
- Resource constraints limit the available solutions, as limited memory prevents storing all historical data for replay-based approaches that work well in centralized settings but exceed edge device capabilities.

Meta-learning approaches address these challenges by learning algorithms themselves rather than just learning specific tasks. Model-Agnostic Meta-Learning (MAML) trains models to quickly adapt to new tasks with minimal data, exactly the capability required for personalized on-device adaptation where collecting large user-specific datasets is impractical. Few-shot learning techniques enable rapid specialization from small user-specific datasets, allowing models to personalize based on just a handful of examples while maintaining general capabilities learned during pretraining.

The theoretical foundation points to a compact design rule: durable on-device learning minimizes the adaptation footprint while preserving enough plasticity to track local change. Sparse model architectures reduce memory and compute requirements, self-supervised objectives use abundant unlabeled sensor data, and meta-learning enables efficient personalization from limited user interactions.

That rule first changes what the device is allowed to update. Full-model fine-tuning is typically infeasible on edge platforms, so localized update strategies, including bias-only optimization, residual

adapters, and lightweight task-specific heads, keep model specialization within resource constraints while reducing the risk of overfitting or instability. When personalization is required, restricting updates to lightweight components constrains catastrophic forgetting, reduces memory overhead, and accelerates adaptation without destabilizing core model representations. The feasibility of this strategy depends critically on the strength of offline pretraining (Bommasani et al. 2021): pretrained models must encapsulate generalizable features so the device spends its limited energy specializing behavior rather than relearning representations from scarce local data.

Once the update footprint is bounded, the runtime must decide when the update can execute and whether it should be shared. Opportunistic scheduling defers local updates to periods when the device is idle, connected to external power, and operating on a reliable network, minimizing the impact of background training on latency, battery consumption, and thermal performance. In decentralized or federated learning contexts, the same budget pressure applies to communication: quantized gradient updates, sparsified parameter sets, and selective model transmission allow large heterogeneous fleets to coordinate without overwhelming bandwidth or energy budgets (Konečný et al. 2016).

Stability then becomes a state-management problem. Replay buffers, support sets, adaptation logs, and model update metadata must be protected against unauthorized access or tampering because they encode the local evidence that shaped the model. Lightweight encryption or hardware-backed secure storage can reduce that exposure, but security controls do not prove that adaptation remains beneficial. Lightweight validation techniques, including confidence scoring, drift detection heuristics (Gama et al. 2014), and shadow model evaluation, monitor adaptation dynamics and can trigger rollback before severe degradation occurs. Robust rollback procedures require a trusted baseline checkpoint that can be restored quickly, especially in safety-important and regulated domains where failure recovery must be provable.

Privacy and compliance requirements close the loop. User consent, data minimization, retention limits, and the right to erasure must be designed into the adaptation pipeline rather than added after deployment. Meeting regulatory obligations at scale demands on-device learning workflows that preserve auditable autonomy: the model can adapt in place, but the system still retains enough control to explain, bound, and reverse that adaptation when necessary.

Consider figure 11.15 for a systematic decision framework: the flowchart guides practitioners through key decision points about adaptation complexity, compute availability, and data sharing requirements, mapping these choices to concrete implementation strategies from bias-only updates to full federated learning with privacy measures.

A design rule and a decision flowchart describe what a single device should do. Running that adaptation across a fleet, on a release cadence, with monitoring and rollback, is a separate discipline: it turns these continual-learning choices into operations that must survive CI/CD, versioning, and the absence of centralized labels.

11.8 Production Integration

It is one thing to write a proof-of-concept script that fine-tunes an adapter on a Raspberry Pi. It is an entirely different discipline to ship that capability to 50 million devices within a strict CI/CD pipeline, ensuring that a bad gradient update does not brick the app. Production integration is where the theoretical elegance of edge intelligence collides with the unforgiving realities of mobile software engineering.

11.8.1 MLOps integration challenges

Integrating on-device learning into MLOps changes what operations must validate: a hierarchy of backbones, adapters, policies, and local states spread across heterogeneous devices, in place of a single model artifact in one controlled environment. Traditional continuous integration pipelines, model versioning systems, and monitoring infrastructure provide essential foundations, but they assume centralized data access, controlled deployment environments, and unified monitoring. Edge learning breaks those assumptions, requiring operational practices that preserve reliability while privacy-preserving coordination and local adaptation continue after deployment.

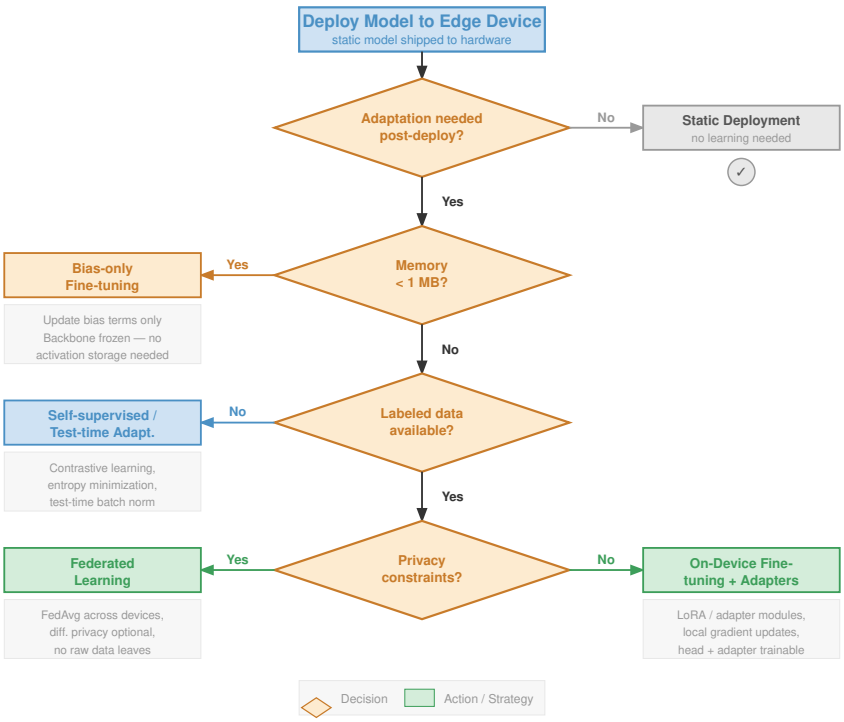


Figure 11.15: On-Device Learning Design Flow: A decision framework for implementing on-device learning. The flowchart maps design choices regarding adaptation complexity (bias-only vs. adapters), compute availability (head-only vs. full fine-tuning), and data sharing policies (localized vs. federated) to specific architectural strategies.

11.8.1.1 Deployment pipeline transformations

Traditional MLOps deployment pipelines release one validated model artifact into uniform infrastructure. On-device learning turns that artifact into a policy bundle because device class determines what safe adaptation means. Microcontrollers receive bias-only updates, mid-range phones use LoRA adapters, and flagship devices perform selective layer updates. The deployable unit includes adaptation policies, initial model weights, and device-specific optimization configurations, so CI/CD must validate the policy that selects among them, not only the weights.

That shift also changes versioning. While centralized systems maintain a single model version, on-device learning systems must simultaneously track multiple versioning dimensions. The pretrained backbone distributed to all devices represents the base model version, which serves as the foundation for all local adaptations. Different update mechanisms deployed per device class constitute adaptation strategies, varying from simple bias adjustments on microcontrollers to full layer fine-tuning on flagship devices. Local model states naturally diverge from the base as devices encounter unique data distributions, creating device-specific checkpoints that reflect individual adaptation histories. Federated learning rounds that periodically synchronize device populations establish aggregation epochs, marking discrete points where distributed knowledge converges into updated global models. A robust deployment implements tiered versioning schemes where base models evolve slowly, often through scheduled updates, while local adaptations occur continuously, creating a hierarchical version space rather than the linear version history familiar from traditional deployments.

11.8.1.2 Monitoring system evolution

Traditional monitoring practices aggregate metrics from centralized inference servers. On-device learning monitoring must operate within fundamentally different constraints that reshape how systems observe, measure, and respond to model behavior across distributed device populations.

Privacy-preserving telemetry represents the first fundamental departure from traditional monitoring. Collecting performance metrics without compromising user privacy requires federated analytics where devices share only aggregate statistics or differentially private summaries. Systems cannot simply log individual predictions or training samples as centralized systems do. Instead, devices report distribution summaries such as mean accuracy and confidence histograms rather than per-example metrics. When formal privacy guarantees are required, reported statistics need differential privacy mechanisms that bound information leakage through carefully calibrated noise addition. Secure aggregation protocols prevent the server from observing individual device contributions, reducing the risk that the aggregation process itself reconstructs private information from any single device's data.

Drift detection presents additional challenges without access to ground truth labels. Traditional monitoring compares model predictions against labeled validation sets maintained on centralized infrastructure. On-device systems must detect drift using only local signals available during deployment:

- Confidence calibration tracks whether predicted probabilities match empirical frequencies, detecting degradation when the model's confidence estimates become poorly calibrated to actual outcomes.
- Input distribution monitoring detects when feature distributions shift from training data through statistical techniques that require no labels.
- Task performance proxies use implicit feedback such as user corrections or task abandonment as quality signals that indicate when the model fails to meet user needs.
- Shadow baseline comparison runs a frozen base model alongside the adapted model to measure divergence, flagging cases where local adaptation degrades rather than improves performance relative to the known-good baseline.

Heterogeneous performance tracking addresses a third critical challenge: global averages mask critical failures when device populations exhibit high variance. Monitoring systems must segment performance across multiple dimensions to identify systematic issues that affect specific device cohorts:

- Capability-based performance gaps reveal when flagship devices achieve substantially better results than budget devices, indicating that adaptation strategies may need adjustment for resource-constrained hardware.
- Regional bias issues surface when models perform well in some geographic markets but poorly in others, potentially reflecting data distribution shifts or cultural factors not captured during initial training.
- Temporal patterns emerge when performance degrades for devices running stale base models that have not received recent updates from federated aggregation.
- Participation inequality becomes visible when comparing devices that adapt frequently against those that rarely participate in training, revealing potential fairness issues in how learning benefits are distributed across the user population.

These slices keep cohort-specific failures visible instead of letting strong devices hide weak-device regressions inside a global average.

11.8.1.3 Continuous training orchestration

Traditional continuous training executes scheduled retraining jobs on centralized infrastructure with predictable resource availability and coordinated execution. On-device learning transforms this into continuous distributed training where millions of devices train independently without global synchronization, creating orchestration challenges that require fundamentally different coordination strategies.

Asynchronous device coordination represents the first major departure from centralized training. Millions of devices train independently on their local data, but the orchestration system cannot rely on synchronized participation. In a representative mobile deployment, only a minority of devices may be available in any training round because of network connectivity limitations, battery constraints, and varying usage patterns. The system must exhibit straggler tolerance, ensuring that slow devices on limited hardware or poor network connections cannot block faster devices from progressing with their local adaptations. Devices often operate on different base model versions simultaneously, creating version skew that the aggregation protocol must handle gracefully without forcing all devices to maintain identical model states. State reconciliation becomes necessary when devices reconnect after extended offline periods, potentially days or weeks, requiring the system to integrate their accumulated local adaptations despite having missed multiple federated aggregation rounds.

Resource-aware scheduling ensures that training respects both device constraints and user experience:

- Orchestration policies implement opportunistic training windows that execute adaptation only when the device is idle, charging, and connected to Wi-Fi, avoiding interference with active user tasks or consuming metered cellular data.
- Thermal budgets suspend training when device temperature exceeds manufacturer-specified thresholds, preventing user discomfort and hardware damage from sustained computational loads.
- Battery preservation policies limit training energy consumption to less than 5 percent of battery capacity per day, ensuring that on-device learning does not noticeably impact device runtime from the user's perspective.
- Network-aware communication compresses model updates aggressively when devices must use metered connections, trading computational overhead for reduced bandwidth consumption to minimize user data charges.

Convergence assessment without global visibility poses the final orchestration challenge. Traditional training monitors loss curves on centralized validation sets, providing clear signals about training progress and convergence. Distributed training must assess convergence through indirect signals aggregated across the device population:

- Federated evaluation aggregates validation metrics from devices that maintain local held-out sets, providing approximate measures of global model quality despite incomplete device participation.
- Update magnitude tracking monitors how much local gradients change the global model in each aggregation round, with diminishing update sizes signaling potential convergence.
- Participation diversity ensures broad device representation in aggregated updates, preventing convergence metrics from reflecting only a narrow subset of the deployment environment.
- Temporal consistency detects when model improvements plateau across multiple aggregation rounds, indicating that the current adaptation strategy has exhausted its potential gains and may require adjustment.

Together, these signals substitute population-level progress checks for the centralized loss curves unavailable in federated settings.

11.8.1.4 Validation strategy adaptation

Traditional validation approaches assume access to held-out test sets and centralized evaluation infrastructure where model quality can be measured directly against known ground truth. On-device learning requires distributed validation that respects privacy and resource constraints while still providing reliable quality signals across heterogeneous device populations.

Shadow model evaluation provides a validation mechanism by maintaining multiple model variants on each device and comparing their behavior. Devices simultaneously run a baseline shadow model, a frozen copy of the last known-good base model that provides a stable reference point, alongside the locally-adapted version that reflects recent on-device training. Some systems also maintain the recent federated aggregation result as a global model variant, enabling comparison

between individual device adaptations and the collective knowledge aggregated from the entire device population. By comparing predictions across these variants on incoming data streams, systems detect when local adaptation degrades performance relative to established baselines. This comparison occurs continuously during normal operation, requiring no additional labeled validation data. When the adapted model consistently underperforms the baseline shadow, the system triggers automatic rollback to the known-good version, preventing performance degradation from persisting in production.

Confidence-based quality gates provide an additional validation signal when labeled validation data is unavailable. Without ground truth labels, systems use prediction confidence as a quality proxy that correlates with model performance. Well-calibrated models should exhibit high confidence on in-distribution samples that resemble their training data, with confidence scores that accurately reflect the probability of correct predictions. Confidence drops indicate either distributional shift, where input data no longer matches training distributions, or model degradation from problematic local adaptations. Threshold-based gating implements this validation mechanism by continuously monitoring average prediction confidence and suspending adaptation when confidence falls below baseline levels established during initial deployment. This approach catches many failure modes without requiring labeled validation data, though it cannot detect all performance issues since overconfident but incorrect predictions can maintain high confidence scores.

Federated A/B testing enables validation of new adaptation strategies or model architectures across distributed device populations. To validate proposed changes, systems implement distributed experiments that randomly assign devices to treatment and control groups while maintaining statistical balance across device tiers and usage patterns. Both groups collect federated metrics using privacy-preserving aggregation protocols that prevent individual device data from being exposed while enabling population-level comparisons. The system compares adaptation success rates, measuring how frequently local adaptations improve over baseline models, along with convergence speed that indicates how quickly devices reach optimal performance, and final performance metrics that reflect ultimate model quality after adaptation completes. Successful strategies demonstrating clear improvements in treatment groups are rolled out gradually across the device population, starting with small percentages and expanding only after confirming that benefits generalize beyond the experimental cohort.

These operational transformations necessitate new tooling and infrastructure that systematically extends traditional MLOps practices. The CI/CD pipelines, monitoring dashboards, A/B testing frameworks, and incident response procedures established for centralized deployments form the foundation for on-device learning operations. The federated learning protocols presented in section 11.5 provide the coordination mechanisms for distributed training, while section 11.10.3 addresses the observability gaps created by decentralized adaptation. The shadow validation approach operates as a continuous comparison pipeline running on each device (figure 11.16), where incoming data flows through both a frozen baseline and the locally adapted model before an arbiter decides whether to accept or roll back adaptations.

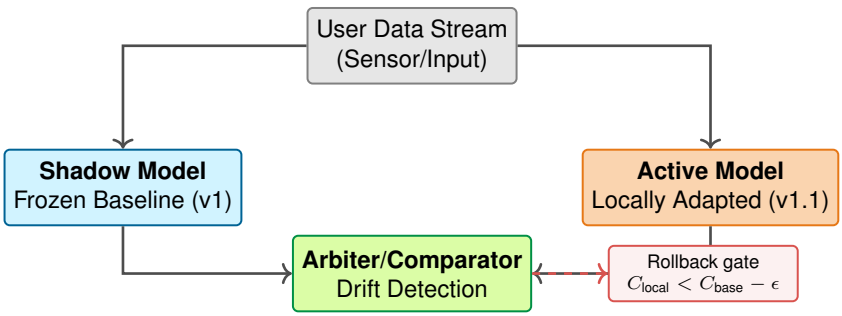


Figure 11.16: On-Device Shadow Validation: To detect model drift without labels, a known-good Shadow Model runs in parallel with the Active Model. An on-device arbiter compares their predictions and confidence scores. If the locally adapted model consistently shows lower confidence than the frozen baseline, the system detects personalization drift and can trigger an automatic rollback.

Successful on-device learning deployments build on proven MLOps methodologies while adapting them to the unique challenges of distributed, heterogeneous learning environments. This evolutionary approach ensures operational reliability while enabling the benefits of edge learning. With the pillars built and the operational scaffolding in place, the remaining question is how the three fit together in a single fleet rather than as separate techniques.

11.9 Putting the Pillars Together

The integration is concrete: rather than deploy one adaptation policy everywhere, bound the risks by matching each pillar to device capability. Consider a production voice assistant deployment across 50 million devices, where each layer constrains what the others are allowed to do.

The chapter built three pillars separately: model adaptation, data efficiency, and federated coordination. Treated as isolated techniques, each one solves a local constraint but leaves the others free to misbehave. The production voice assistant shows the central payoff: integrate all three by device tier, so the policy that decides how much a device may adapt is the same policy that decides how much it may remember and how it may participate in the fleet.

The model adaptation layer stratifies techniques by device capability, matching sophistication to available resources. Flagship phones representing the top 20 percent of the deployment use LoRA adapters of rank 32 that enable sophisticated voice pattern learning through high-dimensional parameter updates. Mid-tier devices comprising 60 percent of the fleet employ adapters of rank 16 that balance adaptation expressiveness with the tighter memory constraints typical of mainstream smartphones. Budget devices making up the remaining 20 percent rely on bias-only updates that stay comfortably within 1 GB memory limits while still enabling basic personalization.

The data efficiency layer implements adaptive strategies across the entire device population while respecting individual resource constraints. All devices implement experience replay[43], but with device-appropriate buffer sizes, 10 MB on budget devices vs. 100 MB on flagship models, ensuring that memory-constrained devices can still benefit from replay-based learning. Few-shot learning enables rapid adaptation to new users within their first 5–10 interactions, reducing the cold-start problem[44] that plagues systems requiring extensive training data. Streaming updates accommodate continuous voice pattern evolution as users' speaking styles naturally change over time or as they use the assistant in new acoustic environments.

The federated coordination layer orchestrates privacy-preserving collaboration across the device population. Devices participate in federated training rounds opportunistically based on connectivity status and battery level, ensuring that coordination does not degrade user experience. LoRA adapters aggregate efficiently with just 50 MB per update compared to 14 GB for full model synchronization, making federated learning practical over mobile networks. Privacy-preserving aggregation protocols ensure that individual voice patterns never leave devices while still enabling population-scale improvements in accent recognition and language understanding that benefit all users.

These layers only work as a system when the integration policy protects against capability mismatch, component failure, resource conflicts, and untested emergent behavior. Table 11.7 maps edge-learning integration controls for each system risk: hierarchical capability matching, graceful degradation, explicit priority policy, and performance validation.

[43] **Experience Replay**: Borrowed from reinforcement learning (Mnih et al. 2015), stores past input-output pairs in a buffer and interleaves them with new training data to reduce catastrophic forgetting in continual learning (Rolnick et al. 2019). Memory-efficient edge implementations may store compressed embeddings or summaries rather than raw inputs.

[44] **Cold-Start Problem**: New privacy pressure lawful collects complete gradient, formal predictions, sampling global and representation context, or explicit user input. Multi-turned initial offers (MAML) or pre-trained encoders can bootstrap personalization, but the fundamental trade-off remains: faster cold-start requires more information from the global model, which may not represent the new user's distribution. Few-shot adaptation from 5–10 interactions offers one balance of speed and local relevance.

Table 11.7: Edge-Learning Integration Controls: A tiered edge-learning deployment needs explicit controls for capability matching, graceful degradation, resource conflicts, and validation across device and network combinations.

Integration risk	Control	Why it matters
Capability mismatch	Hierarchical capability matching	Deploys more sophisticated techniques on capable devices while preserving basic functionality across the full device spectrum.
Component failure	Graceful degradation	Keeps local adaptation useful when connectivity is poor or battery constraints force the system into a minimal adaptation mode.
Resource conflict	Explicit priority policy	Prevents model adaptation and replay buffers from competing for the same memory, energy, or latency budget without a predefined owner.
Emergent behavior	Performance validation	Tests combinations of device tier, network state, and adaptation policy because integrated systems fail in ways individual techniques do not expose.

This integrated approach transforms on-device learning from a collection of techniques into a coherent systems capability that provides robust personalization within real-world deployment constraints. A tiered adaptation strategy (figure 11.17) maps these techniques to device capabilities.

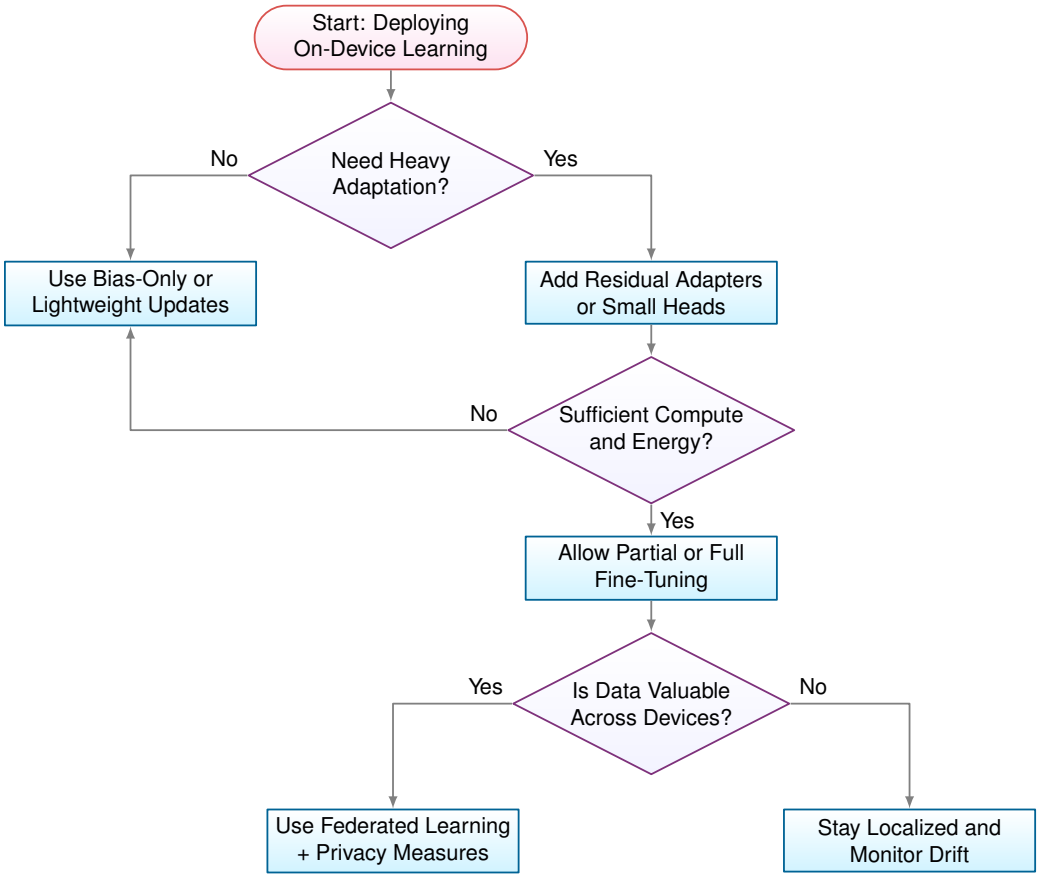


Figure 11.17: Tiered Adaptation Strategy: Decision flowchart for selecting on-device learning techniques based on adaptation complexity, compute budget, and data sharing requirements. Lightweight devices use bias-only updates, capable devices add residual adapters, and devices with sufficient compute allow full fine-tuning. Data value across devices determines whether to use federated learning or stay localized.

Checkpoint 11.4: Integrating the three pillars across device tiers

The chapter built three pillars for on-device learning: model adaptation (bias-only updates, adapters, full fine-tuning), data efficiency (few-shot, experience replay, contrastive), and federated coordination (aggregation, secure aggregation, drift handling). Figure 11.17 maps each technique to device-class capability. Test whether you can integrate the three pillars across the same deployment.

- ☐ **Device-tier capability matching**: For a fleet with Cortex-M0+ wake-word sensors, Cortex-A78 smartphones, and edge servers, map each device tier to the adaptation technique it supports in figure 11.17 and state what fails when one technique is pushed across all three tiers.
- ☐ **Fairness in federated participation**: When 90 percent of a federated round's compute comes from the top 10 percent of devices by capability, infer the implication for the global

model's representation of long-tail user behavior and identify the defense from the data efficiency pillar.

- ☐ **Drift detection without labels**: For a deployed personalization model with no ground-truth labels at the edge, describe how the device can detect enough local distribution shift to warrant a federated round using only signals available to the on-device inference path.
- ☐ **Secure aggregation's participation floor**: With a secure-aggregation floor of 100 devices and only 30 active devices in a given hour, determine the system response and identify which pillar must compensate.
- ☐ **The auditable-autonomy trade-off**: Separate the signals a central operator can observe from the device state that must remain hidden, and locate the catastrophic forgetting risk in that split.

The hardest edge-learning failures appear when model adaptation, data efficiency, and federated coordination interact rather than when any one pillar fails alone. The techniques above solve individual constraints, but they also create interaction failures in real deployments: an adapter can fit memory yet overfit sparse local data, a replay buffer can stabilize learning yet leak sensitive history, and a federated round can improve the global model while amplifying participation bias. These limits provide critical context for deciding when on-device learning is appropriate and when a simpler strategy is safer.

That interaction is what separates edge learning from conventional centralized training. Controlled training environments can standardize hardware, curate data, and validate updates before release. Edge deployments inherit heterogeneous devices, fragmented data, and limited visibility at the same time, so the next design question is how each source of variation changes the boundaries of the adaptation strategies, data efficiency methods, and coordination mechanisms developed earlier in the chapter.

11.10 Engineering Challenges and Mitigations

A billion-user predictive text model that learns from live keyboard inputs can be poisoned if a coordinated group of malicious users repeatedly types a specific slur and those local updates enter the global federated aggregate. The tiered integration policy bounds where each pillar may act, but it does not erase the failure surfaces that exposing an adapting model to the real world creates: device and data heterogeneity, non-IID distributions, limited observability, resource contention, silent failures, and the security and compliance risks of post-release adaptation. The remainder of this section works through those challenges and the engineering mitigations each one demands, beginning with the variation that is hardest to design around because it is present from the first rollout.

11.10.1 Device and data heterogeneity management

Heterogeneity is not background variation; it determines which devices can train, which can validate, and which can only contribute lightweight signals. The quantitative spread established in section 11.6.5.3 gives the resource envelope. Here the management problem is reproducibility: a single rollout must behave consistently across smartphones, wearables, IoT sensors, and microcontrollers whose hardware tiers, software stacks, network connectivity, and power availability differ sharply.

Hardware tiers still shape the execution path. ARM Cortex-M-class devices, mobile-class A-series CPUs, and phones with dedicated NPUs[45] may all participate in the same federated population, but each tier supports a different adaptation envelope. The system must therefore assign model formats, training algorithms, validation duties, and update frequencies by device class rather than treating the fleet as uniform.

Software heterogeneity compounds the challenge. Devices may run different versions of operating systems, kernel-level drivers, and runtime libraries. Some environments support optimized ML runtimes like TensorFlow Lite[46] Micro or ONNX Runtime Mobile[47], while others rely on custom

45 **ARM Cortex Spectrum**: Representative endpoints span nearly six orders of magnitude, from Cortex-M-class devices around 48 MHz, 32 KB RAM, and 10 µW power to mobile-class devices around 3 GHz, 16 GB RAM, and 5 W (Warden and Situnayake 2020; Lai et al. 2018). Federated-learning surveys and production-system reports identify this kind of client heterogeneity and availability skew as a core systems challenge (Kairouz and McMahan 2021; Bonawitz et al. 2019). The design consequence is tiering: quantized inference on the smallest microcontrollers, lightweight bias-only updates on larger embedded or wearable tiers, and fuller adapter or backpropagation loops only on mobile-class SoCs.

46 **TensorFlow Lite**: Google's mobile inference runtime, optimized for ARM with specialized 8-bit and 16-bit kernels that achieve 3× faster inference than full TensorFlow. TFLite Micro extends to microcontrollers with $<$1 MB memory. The systems consequence for federated learning: runtime differences between TFLite and other frameworks can cause numerically different gradient updates from identical models, introducing aggregation noise that compounds across heterogeneous device fleets.

47 **ONNX Runtime Mobile**: Microsoft's cross-platform inference engine using the vendor-neutral Open Neural Network Exchange format, enabling training in one framework and deployment across iOS, Android, and embedded Linux. The mobile variant achieves 2–3× faster inference than TFLite on some models through operator fusion. For federated learning, ONNX's portability reduces the numerical divergence problem: a single model representation across heterogeneous devices produces more consistent gradient updates during aggregation.

inference stacks or restricted APIs. These discrepancies can lead to subtle inconsistencies in behavior, especially when models are compiled differently or when floating-point precision varies across platforms.

Connectivity and uptime add the scheduling dimension. Some devices are intermittently connected, plugged in only occasionally, or operating under strict bandwidth constraints. Others have continuous power and reliable networking, but still prioritize user-facing responsiveness over background learning. These differences complicate coordinated learning because update eligibility changes with the device's current state, not only with its hardware tier.

System fragmentation closes the loop by making reproducibility and testing a fleet property rather than a lab property. With such a wide range of execution environments, consistent model behavior is difficult to guarantee and failures are difficult to reproduce. Monitoring, validation, and rollback therefore become more important, while also becoming harder to implement uniformly across the fleet.

The consequence is visible in a federated learning deployment for mobile keyboards. A high-end smartphone might feature 8 GB of RAM, a dedicated AI accelerator, and continuous Wi-Fi access. In contrast, a budget device may have just 2 GB of RAM, no hardware acceleration, and rely on intermittent mobile data. These disparities influence how long training runs can proceed, how frequently models can be updated, and even whether training is feasible at all. To support such a range, the system must dynamically adjust training schedules, model formats, and compression strategies, ensuring equitable model improvement across users while respecting each device's limitations.

11.10.2 Non-IID data distribution challenges

In centralized machine learning, data can be aggregated, shuffled, and curated to approximate independent and identically distributed (IID) samples, a key assumption underlying many learning algorithms. On-device and federated learning systems fundamentally challenge this assumption, requiring algorithms that can handle highly fragmented and non-IID data across diverse devices and contexts.

The statistical implications of this fragmentation create cascading challenges throughout the learning process. Gradients computed on different devices may conflict, slowing convergence or destabilizing training. Local updates risk overfitting to individual client idiosyncrasies, reducing performance when aggregated globally. The diversity of data across clients also complicates evaluation, as no single test set can represent the true deployment distribution.

Operationally, non-IID data is a control problem as much as a statistical one. Clustered federated learning, personalization layers, stratified client sampling, per-cohort validation, importance weighting, and adaptive aggregation schemes provide partial controls, but the optimal mix depends on which population slices are underrepresented and which update conflicts are destabilizing the global model.

11.10.3 Distributed system observability

Observability is the control plane that keeps local adaptation from becoming invisible. Traditional centralized MLOps monitors collect prediction, label, and performance telemetry in one place; that approach becomes impractical when devices operate intermittently connected and data cannot be centralized. Edge systems still need drift detection and performance monitoring, but those techniques must work through privacy-preserving summaries, local gates, and partial population signals.

The visibility shift matters because on-device models keep changing after release. In centralized systems, model updates can be evaluated against held-out validation sets before promotion. In on-device systems, internal updates may proceed inside highly diverse and disconnected environments, creating the central safety question for edge observability: whether local adaptation is improving the model or silently moving it away from the intended behavior.

A core difficulty lies in the absence of centralized validation data. In traditional workflows, models are trained and evaluated using curated datasets that serve as proxies for deployment conditions. On-device learners, by contrast, adapt in response to local inputs, which are rarely labeled and may not be systematically collected. As a result, the quality and direction of updates, whether

they enhance generalization or cause drift, are difficult to assess without interfering with the user experience or violating privacy constraints.

The risk of model drift is especially pronounced in streaming settings, where continual adaptation may cause a slow degradation in performance. For instance, a voice recognition model that adapts too aggressively to background noise may eventually overfit to transient acoustic conditions, reducing accuracy on the target task. Without visibility into the evolution of model parameters or outputs, such degradations can remain undetected until they become severe.

Mitigating this problem requires mechanisms for on-device validation and update gating. The shadow-model and checkpoint-rollback mechanisms introduced in section 11.8.1.4 are the practical answer to this observability gap: compare the adapted model against a stable baseline, suspend adaptation when confidence falls below the deployment threshold, and retain a known-good state that can be restored when local learning degrades behavior. The observability challenge is deciding which lightweight signals are trustworthy enough to trigger those gates without collecting the raw user data that would make validation straightforward.

In some cases, federated validation offers a partial solution. Devices can share anonymized model updates or summary statistics with a central server, which aggregates them across users to identify global patterns of drift or failure. While this preserves some degree of privacy, it introduces communication overhead and may not capture rare or user-specific failures.

Update monitoring and validation in on-device learning require rethinking traditional evaluation practices. Instead of centralized test sets, systems must rely on implicit signals, runtime feedback, and conservative adaptation policies to ensure robustness. The absence of global observability reflects a deeper systems challenge: aligning local adaptation with global reliability.

11.10.3.1 Performance evaluation in dynamic environments

Systematic approaches for measuring ML system performance include inference latency, throughput, energy efficiency, and accuracy metrics. These benchmarking methodologies provide foundations for characterizing model performance, but they were designed for static inference workloads. On-device learning requires extending these metrics to capture adaptation quality and training efficiency through training-specific benchmarks.

Beyond traditional inference metrics, adaptive systems require measures that explain whether local learning is worth the resource cost. Adaptation efficiency measures how much accuracy improves per training sample consumed. A system that gains 2 percent accuracy from 100 local samples is more useful on a device than one that needs 500 samples for the same improvement, because fewer samples mean faster personalization, less local storage, and fewer training windows competing with user-facing work.

Device-envelope fit measures whether the improvement stays inside resource budgets. Memory-constrained convergence evaluates validation loss under a fixed RAM budget, such as "convergence within 512 KB training footprint," while energy-per-update measures millijoules consumed per gradient step; background adaptation commonly budgets 500–1000 mW, translating to roughly 1.8–3.6 kJ/hour of adaptation before noticeably affecting battery life. Time-to-adaptation extends the same idea to scheduling by measuring wall-clock time from new data to measurable improvement, including waiting for idleness, charging status, and thermal headroom rather than only raw accelerator throughput.

Personalization quality measures whether adaptation improves the right behavior without damaging the base model. Per-user performance delta compares the adapted model against the global baseline on user-specific holdout data, and deployments typically need statistically significant gains, often above 2 percent accuracy, before accepting the compute and energy overhead. Personalization-privacy trade-off measures accuracy gain per unit of local data exposure, while catastrophic forgetting rate measures degradation on the original task after local adaptation; many systems target less than 5 percent accuracy loss on the original task so personalization does not erase general capability.

When devices coordinate through the federated protocols examined in section 11.5, the metrics shift from one device to the population. Communication efficiency measures accuracy improvement per byte transmitted, capturing whether gradient compression and selective updates make mobile deployment practical; compressed-update designs can greatly reduce transmitted bytes, but accuracy retention depends on the model, optimizer, data heterogeneity, and compression rule (Konečný et al.

2016; Kairouz and McMahan 2021). Straggler impact measures convergence delay caused by slow or unreliable devices, while aggregation quality tracks global model performance as participation rate changes. Together, these metrics reveal the deployment-specific minimum viable participation threshold below which federated learning becomes unstable.

These training-specific benchmarks complement inference metrics including latency, throughput, and accuracy, creating complete performance characterization for adaptive systems. Practical benchmarking must measure both dimensions: a system that achieves fast inference but slow adaptation, or efficient adaptation but poor final accuracy, fails to meet real-world requirements. The integration of inference and training benchmarks enables holistic evaluation of on-device learning systems across their full operational lifecycle.

11.10.4 Resource management

On-device learning introduces resource contention modes absent in conventional inference-only deployments. At deployment time, feasibility is no longer abstract; the runtime question is how to arbitrate scarce resources while the user is still interacting with the device. Many edge devices are provisioned to run pretrained models efficiently but are rarely designed with training workloads in mind. Local adaptation therefore competes for scarce resources, including compute cycles, memory bandwidth, energy, and thermal headroom, with other system processes and user-facing applications.

The most direct constraint is compute availability. Training involves additional forward and backward passes through the model, which can exceed the cost of inference. Even when only a small subset of parameters is updated, for instance, in bias-only or head-only adaptation, backpropagation must still traverse the relevant layers, triggering increased instruction counts and memory traffic. On devices with shared compute units (for example, mobile SoCs or embedded CPUs), this demand can delay interactive tasks, reduce frame rates, or impair sensor processing.

Energy consumption compounds this problem. Adaptation typically involves sustained computation over multiple input samples, which taxes battery-powered systems and may lead to rapid energy depletion. For instance, performing a single epoch of adaptation on a microcontroller-class device can consume several millijoules[48], an appreciable fraction of the energy budget for a duty-cycled system operating on harvested power. This necessitates careful scheduling, such that learning occurs only during idle periods, when energy reserves are high and user latency constraints are relaxed.

From a memory perspective, training incurs higher peak usage than inference, especially because backpropagation must retain or recompute intermediate activations[49]; efficient on-device training systems reduce this activation burden and limit the amount of trainable state (Cai, Gan, Zhu, et al. 2020; Kwon et al. 2024).

These resource demands must also be balanced against quality of service (QoS)[50] goals. Users expect edge devices to respond reliably and consistently, regardless of whether learning is occurring in the background. Any observable degradation, including dropped audio in a wake-word detector or lag in a wearable display, can erode user trust.

In some deployments, adaptation is further gated by cost constraints imposed by networked infrastructure. For instance, devices may offload portions of the learning workload to nearby gateways or cloudlets[51], introducing bandwidth and communication trade-offs.

The cost of on-device learning is not solely measured in FLOPs or memory usage. It manifests as a complex interplay of system load, user experience, energy availability, and infrastructure capacity. Addressing these challenges requires co-design across algorithmic, runtime, and hardware layers, ensuring that adaptation remains unobtrusive, efficient, and sustainable under real-world constraints.

11.10.5 Identifying and preventing system failures

Those same constraints also make failures hard to see: system failures in on-device learning emerge slowly, locally, and often without centralized evidence. Based on documented challenges in federated learning research (Kairouz and McMahan 2021) and known risks in adaptive systems, several categories of failures warrant careful consideration.

The most fundamental risk in on-device learning is unbounded adaptation drift, where continuous learning without constraints causes models to gradually diverge from their intended behavior.

48 **Microcontroller Power Budget**: A microcontroller drawing 10 mW during training consumes 36 J per hour, so a small 200 mAh coin-cell battery (about 2,200 J of usable energy at 3 V) sustains roughly 60 hours of continuous compute. Indoor energy harvesting provides only 10–100 μW continuously, two to three orders of magnitude below the training draw, so real deployments duty-cycle: training 10 seconds per hour consumes only about 0.1 J per session and limits adaptation to a few hundred gradient steps per session. This energy ceiling, not compute throughput, is the binding constraint that determines which algorithms are feasible on harvested-power devices.

49 **Activation Caching**: Forward pass activations must be stored or recomputed for backpropagation, and for many CNNs this activation memory can exceed the model-weight footprint. On devices with $<$512 KB RAM, activation storage alone can exceed total memory, precluding multi-layer adaptation entirely unless gradient checkpointing (Chen et al. 2016), low-rank updates, or sparse-update methods are employed.

50 **QoS (Quality of Service) for Edge ML**: Latency guarantees that on-device training must not violate: voice assistants, video applications, and wearables each have tight interactive response budgets. Background training that noticeably increases latency, drops frames, or delays sensor processing can damage user trust. Systems enforce this through priority scheduling where inference preempts training, and resource governors throttle learning when QoS metrics approach deployment-specific thresholds.

Consider a hypothetical keyboard prediction system that learns from all user inputs, including corrections. It might begin incorporating typos as valid suggestions, leading to progressively degraded predictions. This risk becomes acute in health monitoring applications where gradual changes in user baselines could be learned as "normal," potentially causing the system to miss important anomalies that would have been detected by a static model. The insidious nature of this drift is that it occurs slowly and locally, making detection difficult without proper monitoring infrastructure.

Beyond individual device drift, federated learning systems face the challenge of participation bias amplification at the population level. Devices with reliable power and connectivity participate more frequently in federated rounds (T. Li et al. 2020). This uneven participation creates scenarios where models become more strongly optimized for users with high-end devices while performance degrades for those with limited resources. The resulting feedback loop exacerbates digital inequality: better-served users receive better models, while underserved populations experience declining performance, reducing their engagement and further diminishing their representation in training rounds (Wang et al. 2021). These fairness and bias amplification concerns highlight the ethical implications of distributed learning systems.

These systematic biases interact with data quality issues to create autocorrection feedback loops, particularly in text-based applications. When systems cannot distinguish between intended inputs and corrections, they may develop unexpected behaviors. Frequently corrected domain-specific terminology might be incorrectly learned as errors, leading to inappropriate suggestions in professional contexts. This problem compounds the drift issue: not only do models adapt to individual quirks, but they may also learn from their own mistakes when users accept autocorrections without realizing the system is learning from these interactions.

The interconnected nature of these failure modes, from individual drift to population bias to data quality degradation, underscores the importance of implementing comprehensive safety mechanisms. Successful deployments require bounded adaptation ranges to prevent unbounded drift, stratified sampling to address participation bias, careful data filtering to avoid learning from corrections as ground truth, and shadow evaluation against static baselines to detect degradation. While specific production incidents are rarely publicized due to competitive and privacy concerns, the research community has identified these patterns as critical areas requiring systematic mitigation strategies (T. Li et al. 2020; Kairouz and McMahan 2021).

11.10.6 Production deployment risk assessment

The deployment of adaptive models on edge devices introduces challenges that extend beyond technical feasibility. In domains where compliance, auditability, and regulatory approval are necessary, including healthcare, finance, and safety-important systems, on-device learning poses a core tension between system autonomy and control.

In traditional machine learning pipelines, all model updates are centrally managed, versioned, and validated. The training data, model checkpoints, and evaluation metrics are typically recorded in reproducible workflows that support traceability. When learning occurs on the device itself, however, this visibility is lost. Each device may independently evolve its model parameters, influenced by unique local data streams that are never observed by the developer or system maintainer.

This autonomy creates a validation gap. Without access to the input data or the exact update trajectory, it becomes difficult to verify that the learned model still adheres to its original specification or performance guarantees. This is especially problematic in regulated industries, where certification depends on demonstrating that a system behaves consistently across defined operational boundaries. A device that updates itself in response to real-world usage may drift outside those bounds, triggering compliance violations without any external signal.

The lack of centralized oversight complicates rollback and failure recovery. If a model update degrades performance, it may not be immediately detectable, particularly in offline scenarios or systems without telemetry. By the time failure is observed, the system's internal state may have diverged significantly from any known checkpoint, making diagnosis and recovery more complex than in static deployments. Recovery therefore depends on robust safety mechanisms, such as conservative update thresholds, rollback caches, or dual-model architectures that retain a verified baseline.

51 **Cloudlet**: A small-scale compute tier located close to users, often near access networks or enterprise sites, provides lower-latency offload than a distant cloud region. This intermediate tier enables hybrid on-device learning: time-sensitive inference runs locally while aggregation, validation, or heavier adaptation work can offload nearby when the network and trust model allow it. The trade-off is deployment complexity: cloudlets add an orchestration layer that must decide per-update whether to compute locally, offload nearby, or defer to cloud.

In addition to compliance challenges, on-device learning introduces new security vulnerabilities. Because model adaptation occurs locally and relies on device-specific, potentially untrusted data streams, adversaries may attempt to manipulate the learning process by tampering with stored data, such as replay buffers, or by injecting poisoned examples during adaptation, to degrade model performance or introduce vulnerabilities. Any locally stored adaptation data, such as feature embeddings or few-shot examples, must be secured against unauthorized access to prevent unintended information leakage.

Maintaining model integrity over time is particularly difficult in decentralized settings, where central monitoring and validation are limited. Autonomous updates could, without external visibility, cause models to drift into unsafe or biased states. These risks are compounded by compliance obligations such as the GDPR's right to erasure: if user data subtly influences a model through adaptation, tracking and reversing that influence becomes complex.

The same local-state problem becomes safety-critical when perception and control loops rely on local models. A human fallback is not a safety mechanism unless the fallback itself is engineered as part of the system.

War Story 11.1: The edge model with a human fallback

Context: A 2015 Tesla Model S operating with Traffic-Aware Cruise Control and Autosteer performed perception and control decisions locally on the vehicle, with the human driver expected to supervise (National Transportation Safety Board 2017).

Failure mode: In the May 7, 2016 Williston, Florida crash, the automation operated outside a safety envelope that reliably ensured driver engagement. The NTSB found the driver's overreliance on automation contributed to the fatal collision.

Consequence: The incident became a reference case for operational design domains, driver-monitoring requirements, and the limits of treating human takeover as a generic fallback for edge autonomy.

Systems lesson: Edge intelligence is a closed-loop system in a physical environment. Latency, perception, user attention, and handoff design must be engineered together; local inference alone does not make a deployment safe.

The immediate design obligation is to treat model state, replay buffers, and adaptation triggers as security-sensitive assets. Edge learning cannot rely on central inspection after the fact; it needs local integrity checks, protected storage, and conservative update policies before adaptation begins.

Privacy regulations also interact with on-device learning in nontrivial ways. While local adaptation can reduce the need to transmit sensitive data, it may still require storage and processing of personal information, including sensor traces or behavioral logs, on the device itself. These privacy considerations require careful attention to security frameworks and regulatory compliance. Depending on jurisdiction, this may invoke additional requirements for data retention, user consent, and auditability. Systems must be designed to satisfy these requirements without compromising adaptation effectiveness, which often involves encrypting stored data, enforcing retention limits, or implementing user-controlled reset mechanisms.

Lastly, the emergence of edge learning raises open questions about accountability for debugging and failure analysis in decentralized training systems (Kairouz and McMahan 2021). When a model adapts autonomously, the question of who is responsible for detecting and correcting its behavior remains unresolved. If an adapted model makes a faulty decision, such as misdiagnosing a health condition or misinterpreting a voice command, the root cause may lie in local data drift, poor initialization, or insufficient safeguards. Without standardized mechanisms for capturing and analyzing these failure modes, root-cause analysis can be difficult.

Addressing these deployment and compliance risks requires tooling, protocols, and design practices that support auditable autonomy, the ability of a system to adapt in place while still satisfying external requirements for traceability, reproducibility, and user protection. For any deployment that permits post-release adaptation, these challenges are central to both system architecture and governance frameworks.

11.10.7 Engineering challenge synthesis

The deployment risks, failure modes, and compliance concerns compound the technical challenges discussed throughout this chapter. These interconnected issues, from drift and bias amplification to validation gaps and accountability questions, represent the full range of challenges that on-device learning systems must navigate. Effective system design depends on how these challenges interact across hardware heterogeneity, data fragmentation, observability limitations, and regulatory compliance requirements.

System heterogeneity complicates deployment and optimization by introducing variation in compute, memory, and runtime environments. Non-IID data distributions challenge learning stability and generalization, especially when models are trained on-device without access to global context. The absence of centralized monitoring makes it difficult to validate updates or detect performance regressions, and training activity must often compete with core device functionality for energy and compute. Finally, postdeployment learning introduces complications in model governance, from auditability and rollback to privacy assurance.

These on-device learning challenges are not isolated; they interact in ways that influence the viability of different adaptation strategies. Table 11.8 synthesizes these interconnected issues, mapping each challenge category to its root cause and system-level implications for on-device learning deployments.

Table 11.8: On-Device Learning Challenges: System heterogeneity, non-IID data, and limited resources introduce unique challenges for deploying and adapting machine learning models on edge devices, impacting portability, stability, and governance. The table details root causes of these challenges and their system-level implications, highlighting trade-offs between model performance and resource constraints.

Challenge	Root Cause	System-Level Implications
System Heterogeneity	Diverse hardware, software, and toolchains	Limits portability; requires platform-specific tuning
Non-IID and Fragmented Data	Localized, user-specific data distributions	Hinders generalization; increases risk of drift
Limited Observability and Feedback	No centralized testing or logging	Makes update validation and debugging difficult
Resource Contention and Scheduling	Competing demands for memory, compute, and battery	Requires dynamic scheduling and budget-aware learning
Deployment and Compliance Risk	Learning continues postdeployment	Complicates model versioning, auditing, and rollback

11.10.8 Foundations for robust AI systems

The operational challenges and failure modes reveal vulnerabilities that extend beyond deployment concerns into fundamental system reliability. Poisoning, inversion, drift, and validation failures also exist in centralized ML systems, but local adaptation and federated coordination amplify them by making failures harder to observe, attribute, and roll back across millions of heterogeneous devices.

Local failures can propagate silently across device populations, unlike failures in centralized systems where errors are usually localized and observable. A corrupted adaptation on one device, if aggregated through federated learning, can poison the global model. Hardware faults that would trigger errors in centralized infrastructure may silently corrupt gradients on edge devices with minimal error detection capabilities.

Federated coordination mechanisms create new attack surfaces while enabling collaborative learning. Adversarial clients can inject poisoned gradients[52] designed to degrade global model performance. Model inversion attacks can extract private information from shared updates despite aggregation. The distributed nature of on-device learning makes these attacks both easier to execute, because client devices can be compromised, and harder to detect, because no centralized validation set sees every update.

On-device systems must handle distribution shifts and environmental changes without access to labeled validation data. Models may confidently drift into failure modes, adapting to local biases or temporary anomalies. The non-IID data distributions across devices mean that local drift on individual devices may not trigger global alarms, allowing silent degradation.

52 **Byzantine Fault Tolerance in FL**: Named after the Byzantine Generals Problem (Lamport et al. 1982), this property enables useful aggregation despite malicious or faulty participants. Krum's original analysis requires conditions such as $2f + 2 < n$ (Blanchard et al. 2017), while coordinate-wise median and trimmed-mean aggregators use different robustness assumptions; the tolerated Byzantine fraction depends on the aggregation rule, dimensionality, and statistical model. These defenses can increase communication and compute overhead because they require extra validation, robust statistics, or redundant participation. That overhead is the price of trust in open federated systems where any client device could be compromised.

These reliability threats demand systematic approaches that keep local adaptation bounded even when devices are faulty, malicious, or operating under environmental shift. The local lesson is that robust aggregation, drift checks, conservative update policies, secure aggregation, and differential privacy are part of the edge-learning architecture, not optional add-ons after deployment; the broader treatment of adversarial robustness as its own discipline is deferred to a later chapter. What remains, and what the next chapter takes up, is the operational problem these safeguards leave open: sustaining the adaptation, monitoring, and lifecycle of a heterogeneous fleet at scale, day after day.

11.11 Fallacies and Pitfalls

A team attempts to port their massive cloud-based recommendation engine directly to a mobile app, assuming a flagship smartphone chip can handle a few epochs of backpropagation. The app instantly crashes from out-of-memory errors, and the phone's battery plummets by 20 percent. This disaster highlights how applying data center intuition to the edge leads directly to catastrophic failure, bringing us to the domain's most common fallacies and pitfalls.

Fallacy: *On-device learning provides the same adaptation capabilities as cloud-based training.*

Teams expect local learning to match centralized training's model improvements, ignoring fundamental resource constraints. On-device learning amplifies resource needs by 4–12× compared to inference-only deployment due to activation caching, gradient storage, optimizer state, and bidirectional memory traffic (section 11.3). Local datasets are typically small, biased, and nonrepresentative, while compute budgets remain orders of magnitude below cloud GPUs. A smartphone with 8 GB RAM and 5 W power budget cannot replicate the adaptation achieved by data center systems with terabytes of memory and kilowatts of power. Effective on-device learning requires designing adaptation strategies that provide meaningful improvements within these constraints rather than attempting to replicate cloud-scale learning capabilities.

Pitfall: *Assuming federated learning automatically preserves privacy without additional safeguards.*

Practitioners believe that keeping data on local devices inherently provides privacy protection, ignoring what can be inferred from model updates. Gradient and parameter updates leak significant information about local training data through various inference attacks, while device participation patterns reveal sensitive information about users and activities. Stronger privacy preservation requires additional mechanisms: differential privacy provides a tunable probabilistic bound on how much the output distribution can change because of one individual's data, while secure aggregation protocols prevent parameter inspection during coordination (Bonawitz et al. 2017) (as figure 11.14 illustrates with cryptographic masking). Data locality alone is insufficient for privacy protection.

Fallacy: *Resource-constrained adaptation always produces better personalized models than generic models.*

This belief assumes that any local adaptation is beneficial regardless of the quality or quantity of local data available. On-device learning with insufficient, noisy, or biased local data can actually degrade model performance compared to well-trained generic models (section 11.4). Small datasets may not provide enough signal for meaningful learning, while adaptation to local noise can harm generalization. Effective on-device learning systems must include mechanisms to detect when local adaptation is beneficial and fall back to generic models when local data is inadequate for reliable learning.

Pitfall: *Ignoring heterogeneity across different device types and capabilities.*

Teams design on-device learning systems assuming uniform hardware capabilities across deployment devices. Real-world deployments span diverse hardware with varying computational power, memory capacity, energy constraints, and networking capabilities. Edge device capabilities span nearly six orders of magnitude in memory and power, and four orders in CPU throughput: from 32 KB RAM microcontrollers to 16 GB smartphones, 48 MHz Cortex-M-class processors (~10 MIPS) to 3 GHz mobile-class processors (~100,000 MIPS), and 10 µW sensor nodes to 5 W flagship phones (Warden and Situnayake 2020; Lai et al. 2018). A learning algorithm that works well on high-end smartphones may fail catastrophically on resource-constrained IoT devices. Federated learning algorithms must account for heterogeneous clients and availability (Kairouz and McMahan 2021; Bonawitz et al. 2019), using selective participation, compression, and tiered aggregation strategies when the deployment spans 10,000× performance differences.

Fallacy: *Coordinating edge learning is just scaling up an aggregation server.*

This belief treats coordination as a stateless averaging problem, as if local updates arrive regularly, represent comparable device populations, and can be merged without operational context. At fleet scale, the aggregation server is only one component in a larger control loop: it must know which devices are eligible, which model version they hold, whether their power and network state can support participation, and whether the resulting update is safe to promote.

Pitfall: *Underestimating the complexity of coordinating learning across distributed edge systems.*

Many teams focus on individual device optimization without considering the system-level challenges of coordinating learning across thousands or millions of edge devices. Edge systems orchestration (section 11.6.5) must handle intermittent connectivity, varying power states, different time zones, and unpredictable device availability patterns that create complex scheduling and synchronization challenges. Device clustering, federated rounds coordination, model versioning across diverse deployment contexts, and handling partial participation from unreliable devices require sophisticated infrastructure beyond simple aggregation servers. Additionally, real-world edge deployments involve multiple stakeholders with different incentives, security requirements, and operational procedures that must be balanced against learning objectives. Effective edge learning systems require robust orchestration frameworks that can maintain system coherence despite constant device churn, network partitions, and operational disruptions.

11.12 Summary

Avoiding these pitfalls, from ignoring thermal throttling to underestimating the chaos of federated orchestration, is essential to architecting edge systems that survive the real world. Edge intelligence is the physics-limited boundary of the Machine Learning Fleet. Data center infrastructure and global serving systems assume abundant power, memory, and connectivity; edge deployments invert those assumptions with sensors that operate under microwatt power budgets, microcontrollers with kilobyte memory, and smartphones facing the mobile memory wall.

Three design levers make learning possible under those constraints. Model adaptation reduces the update footprint by freezing most of the network or using small adapters. Data efficiency extracts useful signal from few local samples or streaming experience. Federated coordination lets a population improve a shared model without centralizing raw data. Success at the edge requires more than algorithmic cleverness; it demands co-design between the software's mathematical requirements and the hardware's thermal, energy, and bandwidth realities.

Many devices run inference workloads locally, from keyboard prediction on smartphones to anomaly detection on industrial sensors. Edge intelligence addresses requirements that no amount of data center scaling can satisfy: latency-critical applications where cloud round-trips introduce unacceptable delays, privacy-sensitive domains where raw data must never leave the device, and connectivity-constrained environments where network access is intermittent or absent. As on-device capabilities advance through hardware specialization and algorithmic efficiency, the boundary between what requires cloud infrastructure and what can execute locally can shift, but the co-design principles remain central to production ML system architecture.

Key Takeaways: Physics sets the budget

- **The edge reverses the fleet**: Data centers fight to keep many accelerators busy; edge devices fight to learn under batteries, kilobytes, thermal throttling, and intermittent links. The same Compute, Communication, Coordination constraints produce a different discipline when scarcity, not scale, is binding.
- **Bandwidth beats advertised TOPS**: On-device LLMs are limited by mobile memory bandwidth, not NPU peak compute. The 30–50× gap between mobile RAM and datacenter HBM makes quantization and locality survival requirements for interactive decode.
- **Learning multiplies the footprint**: On-device training needs activations, gradients, optimizer state, and bidirectional traffic, raising resources 4–12× over inference-only deployment. Adaptation strategies must shrink the update, not merely compress the deployed model.

- **Adaptation needs three levers**: Bias-only updates, LoRA, sparse layers, few-shot data reuse, and experience replay each spend different memory, energy, and privacy budgets. Federated coordination adds population learning only when protocols handle non-IID data and client dropout.
- **Heterogeneity is normal operation**: Edge fleets span microcontrollers, phones, gateways, and NPUs with uneven availability and performance. Federated systems must treat stragglers, participation bias, rollback, and privacy-preserving observability as protocol design constraints, not deployment cleanup.

Everything in this volume so far has pushed in one direction, outward, toward more accelerators, more bandwidth, more power, the data center swelling into a single machine. The edge is that same physics read backward. The constraints have not changed identity, only sign: where the data center fights to keep ten thousand accelerators busy, the edge fights to do anything at all inside a battery and a few kilobytes. That sign flip is why a phone is not a small data center and a sensor is not a slow GPU; when scarcity rather than scale sets the terms, the techniques that win are the ones that get more from less. The same laws, run at the opposite extreme, produce a different discipline.

What's Next: From devices to operations

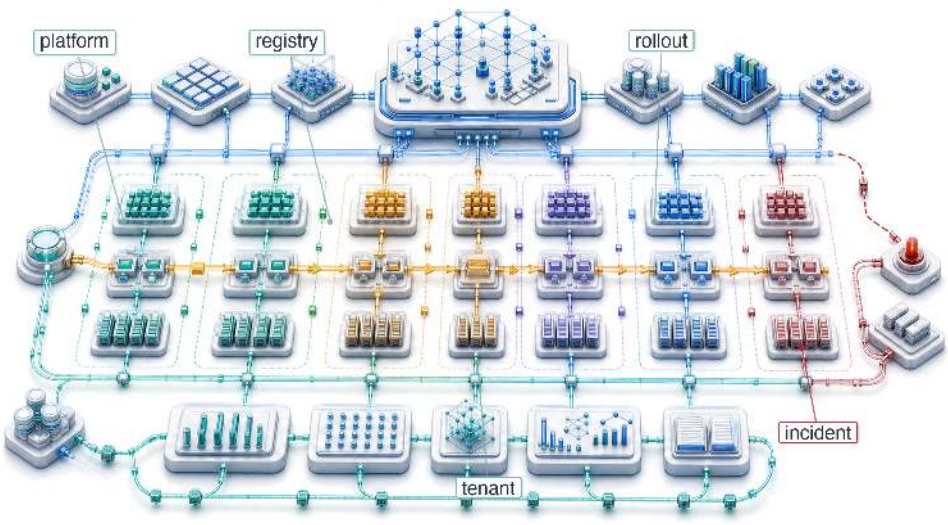
Edge intelligence pushes models to the boundary where hardware diversity, battery budgets, intermittent connectivity, and local autonomy all become first-order constraints. Deploying models across this spectrum, from data center GPUs to edge sensors, is only the beginning. Sustaining these deployments at production scale demands systematic operational practices. Chapter 12 examines the platform engineering, monitoring, and lifecycle management frameworks required to keep the entire Machine Learning Fleet running reliably.

Research Questions: For further inquiry

- How should a system decide which learning belongs on device, in the cloud, or in federated coordination?
- How do serving principles change when power, memory, privacy, and intermittent connectivity replace accelerator utilization as binding constraints?
- When does on-device adaptation justify the extra memory, energy, and training-state footprint over a static model?
- How should federated protocols handle non-IID data, dropout, stragglers, and privacy leakage without treating aggregation as sufficient?

12

ML Operations at Scale

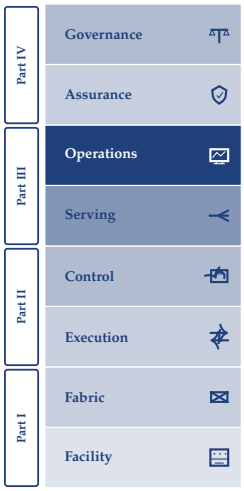


12.1 From Single-Model to Platform Operations
12.2 The N-Models Problem
12.3 Fleet Economics and Utilization
12.4 Multi-Model Management
12.5 CI/CD for ML at Scale
12.6 Monitoring at Scale
12.7 Platform Engineering
12.8 Feature Store Operations
12.9 Organizational Patterns
12.10 Case Studies
12.11 Production Debugging and Incident Response
12.12 Fallacies and Pitfalls
12.13 Summary

Purpose

Why do practices that work for managing one model collapse when organizations deploy hundreds?

One model is a project. A hundred models is a system of systems, where interactions, dependencies, and failures cascade in ways that per-model practices cannot anticipate or contain. A data pipeline change affects twelve models built by four teams, but no single team owns the impact assessment. A deployment failure requires coordinating rollbacks across interconnected services. Monitoring dashboards multiply until alert fatigue makes them useless. The practices that let a single team manage a single model (manual deployment, ad-hoc monitoring, spreadsheet tracking) become organizational liabilities at scale. Machine learning operations (MLOps) at scale is the recognition that model management must become infrastructure: shared platforms with consistent APIs, automated pipelines that enforce quality gates, monitoring systems that aggregate signals across the fleet, and governance frameworks that track dependencies between artifacts nobody remembers creating. Without this infrastructure, organizations drown in operational complexity while their ML investments depreciate. In C^3 terms, operations at scale transforms human coordination into automated compute, standardizing how the fleet is maintained and governed.



Learning Objectives

- Calculate return on investment, utilization, and total cost for shared ML platforms across model portfolios
- Design registries and release gates that preserve lineage, dependency safety, and rollback confidence
- Quantify technical debt using deployment velocity, incident rates, toil, and response-time metrics
- Evaluate monitoring and alerting hierarchies using signal quality, aggregation, and incident-response needs
- Design feature-store operations that manage freshness, ownership, compatibility, and training-serving skew
- Compare centralized, embedded, and hybrid platform teams for operating large model portfolios
- Diagnose incidents by tracing failures across data, platform, serving, model-control, and governance layers

12.1 From Single-Model to Platform Operations

Consider a team of five engineers maintaining a single recommendation model. When the model drifts, they manually retrain it. When the API latency spikes, they manually scale the instances. Now, scale that same team to support five hundred models across dozens of product surfaces. Manual intervention is no longer merely inefficient; it is mathematically impossible. The transition from single-model to platform operations replaces human-in-the-loop maintenance with automated, systemic governance.

The management layer is the fleet stack's control plane: the dashboard, steering, and maintenance system that keeps physical infrastructure, training systems, serving paths, edge deployments, and governance rules from drifting apart. Once models span data centers, serving platforms, and heterogeneous edge fleets, reliability depends less on any single model and more on the operational machinery that keeps the whole fleet observable, coordinated, and recoverable.

Distributed serving architectures handle massive request volumes, while edge deployment pushes intelligence to smartphones, microcontrollers, and federated fleets spanning billions of heterogeneous devices. The question now is what happens when organizations must sustain not one but hundreds of such systems across this entire spectrum. Managing individual models and operating enterprise-scale ML platforms are fundamentally different problems, separated by a phase transition in operational complexity. Platform operations absorbs that complexity through fleet economics and total-cost models, multi-model management, CI/CD, monitoring systems, and feature-store foundations that let hundreds of models share one coherent operational substrate.

Single-model MLOps focuses on continuous integration, deployment pipelines, and monitoring for individual models, and every organization that scales discovers its limits through experience. The first few models can be managed with spreadsheets, manual deployments, and ad hoc monitoring, with each model team developing its own practices optimized for its specific requirements. The approach works initially because the models operate independently: what happens to the recommendation system does not affect the fraud detection model.

Independence vanishes as model count grows. Models begin sharing data sources, and changes to upstream data pipelines cascade through multiple consumers. Infrastructure becomes contested: deployment of one model delays deployment of another. Monitoring dashboards multiply until no single team can observe the complete system state. On-call rotations expand from single-model responsibility to cross-model coordination that requires understanding interactions between systems developed by different teams with different assumptions.

Infrastructure efficiency compounds these coordination challenges. Production ML workloads rarely achieve high accelerator utilization because training jobs run intermittently and inference loads fluctuate with user traffic. A single model team might accept 20 percent accelerator utilization because optimizing further is not worth the engineering investment. Multiply by one hundred

models, and that underutilization represents millions of dollars in wasted infrastructure. Similarly, a single model's occasional production incident is manageable, but one hundred models with independent failure modes produce a constant stream of alerts that exhaust on-call engineers and mask genuine emergencies.

The organizational response is platform thinking. Rather than treating each model as an independent system with its own infrastructure, platforms provide shared services that amortize operational costs across the entire model portfolio. Feature stores[1] eliminate redundant feature computation. Unified deployment pipelines ensure consistent rollout practices. Centralized monitoring aggregates signals across models to detect system-wide issues and enable capacity planning. The challenge is designing, implementing, and operating these platforms so they scale with the portfolio rather than against it.

[1] **Feature Store**: A centralized repository that manages the computation, storage, and serving of ML features. The core systems problem it solves is training-serving skew; this chapter develops that failure mode and the platform invariant that contains it in section 12.8.

12.2 The N-Models Problem

A typical technology organization's journey with machine learning follows a predictable pattern. The first model might be a recommendation system for the homepage, followed by a search ranking model, then a fraud detection system, then content moderation. Each model team initially operates independently, developing bespoke pipelines for data processing, training, validation, and deployment. The absence of coordination overhead lets each team optimize for its specific requirements.

As the number of models grows, the problems that emerge are not *multiplicative* but *combinatorial*. One hundred models do not require 100 times the operational effort of one model; they introduce dependencies and interactions that create superlinear growth in operational complexity. Table 12.1 quantifies this growth across six operational dimensions, from deployment coordination that becomes critical path at scale to debugging complexity that demands distributed tracing across model boundaries.

Table 12.1: Operational Complexity Growth at Scale: Six dimensions of operational complexity across 1, 10, and 100 models. Deployment coordination evolves from nonexistent to critical path, monitoring dashboards become unmanageable without aggregation, and debugging shifts from local investigation to organization-wide distributed tracing requirements.

Operational Aspect	Single Model	10 Models	100 Models
Deployment coordination	None	Ad hoc	Critical path
Shared data dependencies	None	Some overlap	Dense graph
Monitoring dashboards	1	10	Unmanageable
On-call rotation scope	Single team	Multiple teams	Organization-wide
Infrastructure utilization	Often idle	Moderate sharing	Efficiency critical
Debugging complexity	Local	Cross-team	Distributed tracing required

Napkin Math 12.1: The sharing dividend

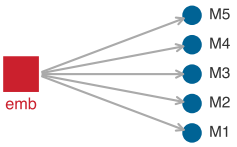
Problem: A platform team manages a fleet of 100 GPUs. Under dedicated per-team quotas, average idle time is 70 percent. Moving to a multi-tenant ML platform that shares resources across teams and uses idle training GPUs for inference reduces aggregate idle time to 30 percent. What hardware cost does the platform save?

Math: For a fixed active workload, required hardware is inverse to utilization, while useful work per GPU is proportional to utilization.

1. **Efficiency Gain**: 0.70 (Shared) / 0.30 (Dedicated) = 2.33× more work per GPU.
2. **Hardware Reduction**: 1 - (0.30/0.70) = 57.1 percent.
3. **Annual Savings**: 57 percent of $1.75M budget ≈ $1,000,000/year.

Systems insight: Multi-tenancy acts as an infrastructure multiplier. Breaking down resource silos reduces required hardware by 57 percent for the same workload; with the same hardware

budget, it raises useful work from 30 to 70 active GPU-equivalents. In the machine learning fleet, *statistical multiplexing* (the principle that different teams' peak demands rarely coincide) is the mechanism that makes shared platforms economically sustainable. The platform team's primary role is to harvest this sharing dividend and reinvest it into future capacity growth.



One upstream embedding update can degrade every dependent model.

The fundamental insight is that per-model operational practices do not compose. When Model A depends on features computed by Pipeline B, which uses embeddings from Model C, changes to any component cascade unpredictably. A seemingly innocuous update to Model C's embedding layer might shift the feature distributions that Model A depends upon, degrading its performance even though Model A itself has not changed. This cascading interdependence turns scale into a qualitatively different management problem.

Systems Perspective 12.1: The complexity explosion

Past a certain fleet size, the binding problem stops being individual model optimization and becomes system-level coordination: the interactions between models matter more than any single model does. Dependency graphs, shared features, and contention for the same infrastructure mean that the marginal cost of the hundredth model is dominated by how it couples to the other ninety-nine, not by the model itself.

Figure 12.1 visualizes this superlinear growth across three complexity dimensions. Monitoring alerts grow linearly with model count, but dependency conflicts grow quadratically as models share features, data sources, and infrastructure. The total operational load crosses team capacity around 50 models, the empirical threshold where organizations discover they need platform engineering.

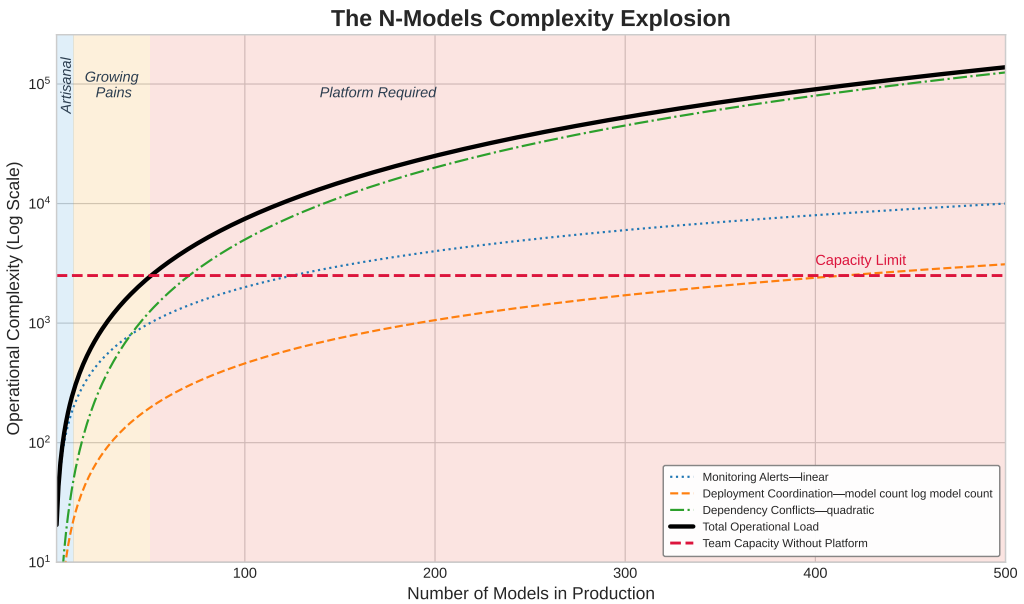


Figure 12.1: The N-Models Complexity Explosion: Monitoring alerts grow linearly with model count, deployment coordination grows as $\mathcal{O}(N_{\text{models}} \log N_{\text{models}})$, and dependency conflicts grow quadratically as models share features and data sources. The total operational load crosses team capacity around 50 models, marking the transition from artisanal model management to platform-required operations.

12.2.1 Quantifying platform economics

The economic case for platform operations rests on understanding both the costs of fragmented approaches and the returns from shared infrastructure. Equation 12.1 formalizes platform return on investment as the ratio of engineering time savings across all models to total platform cost:

$$\text{ROI}_{\text{platform}} = \frac{N_{\text{models}} \times T_{\text{saved}} \times C_{\text{engineer}}}{C_{\text{platform}}} \tag{12.1}$$

where N_{models} represents the number of models benefiting from the platform, T_{saved} is the engineering time saved per model per period, C_{engineer} is the fully-loaded cost per engineer hour, and C_{platform} is the total platform cost including development, infrastructure, and maintenance.

The equation reveals why platform investments make sense only at sufficient scale. For a small organization with five models, the denominator might exceed the numerator even with significant per-model savings. As model count grows, the numerator scales linearly with N_{models} while platform costs grow much more slowly, typically sublinearly due to infrastructure amortization.

> Napkin Math 12.2: The platform dividend
>
> **Problem**: An organization manages 50 models. A centralized ML Platform team costs $120,000/month. If the platform saves each model team 20 hours of manual toil per month, is the platform investment profitable?
>
> **Math**:
>
> 1. **Gross Monthly Savings**: 50 models $\times$ 20 hours/model $\times$ $150/hr = $150,000.
> 2. **Net Monthly Benefit**: $150,000 (Savings) - $120,000/month (Cost) = $30,000.
> 3. **ROI Ratio**: $150,000 / $120,000/month = 1.25.
>
> **Systems insight**: Platforms exhibit a scaling threshold. At 50 models, this platform earns a 25 percent return on its cost. However, if the organization only had 20 models, the savings would be only $60,000—a 50 percent loss on the platform team's salary. In MLOps, platform engineering is a fixed cost that pays off through variable savings. The right time to build a platform is when the "manual toil tax" across the model fleet exceeds the "platform maintenance tax."

12.2.1.1 From artisanal to industrial operations

The platform-dividend calculation gives the general rule; this worked example turns it into an operating decision. The question is not whether shared tooling is aesthetically cleaner, but whether the fixed platform cost is smaller than the manual toil it removes across the model fleet.

Consider an organization evaluating whether to build a centralized ML platform. Five parameters define the current state:

- 50 production models across 8 teams
- Each model requires 40 engineer-hours monthly for operational tasks
- Engineers cost $150 per hour fully loaded
- Platform development cost: $2 million amortized over 3 years
- Expected time savings: 30 hours per model per month postplatform

Before platform (annual operational cost):
$C_{\text{current}} = 50 \times 40 \times 12 \times 150 = \$3,600,000$
After platform (annual operational cost plus amortized platform cost):
$C_{\text{after}} = 50 \times 10 \times 12 \times 150 + \frac{2,000,000}{3} = \$900,000 + \$666,667 = \$1,566,666.7$
Annual savings reach $2,033,333.3, a 56.5 percent reduction in operational costs. The platform pays for itself within the first year.

The economic gap explains why large technology companies have invested heavily in ML platforms while smaller organizations often struggle to justify similar investments. The economic threshold

typically falls between 20 and 50 models, depending on model complexity and organizational structure.

Figure 12.2 visualizes this threshold effect by plotting platform ROI as a function of model count for two platform cost levels. A \$2M/year platform breaks even at approximately 20 models, while a more expensive \$5M/year enterprise platform requires roughly 50 models to justify the investment. Beyond break-even, ROI grows linearly because each additional model contributes the same per-model savings to the numerator of equation 12.1 while platform costs remain essentially fixed. At 100 models, the \$2M platform delivers 5× return on investment. This linearity is both the economic argument for platform investment and the explanation for why organizations that defer platform building until they are "at scale" often find themselves paralyzed by accumulated operational debt: the break-even point arrives earlier than intuition suggests.

The break-even count is not a universal constant; it moves with the two inputs the reader controls. The platform-dividend notebook above (a \$120K/month platform team saving 20 hours per model) and the worked example (a \$2M build amortized over three years, saving 30 hours per model) reach break-even at different model counts than this figure precisely because they assume different platform cost bases and different per-model savings. The figure's curves fix the platform cost at a round annual figure and assume \$100K of savings per model per year to isolate the threshold's shape; the notebook and worked example trade that roundness for explicit operating assumptions. All three obey the same equation 12.1: change the numerator (savings per model) or the denominator (platform cost) and the break-even point slides, but the linear post-break-even slope does not.

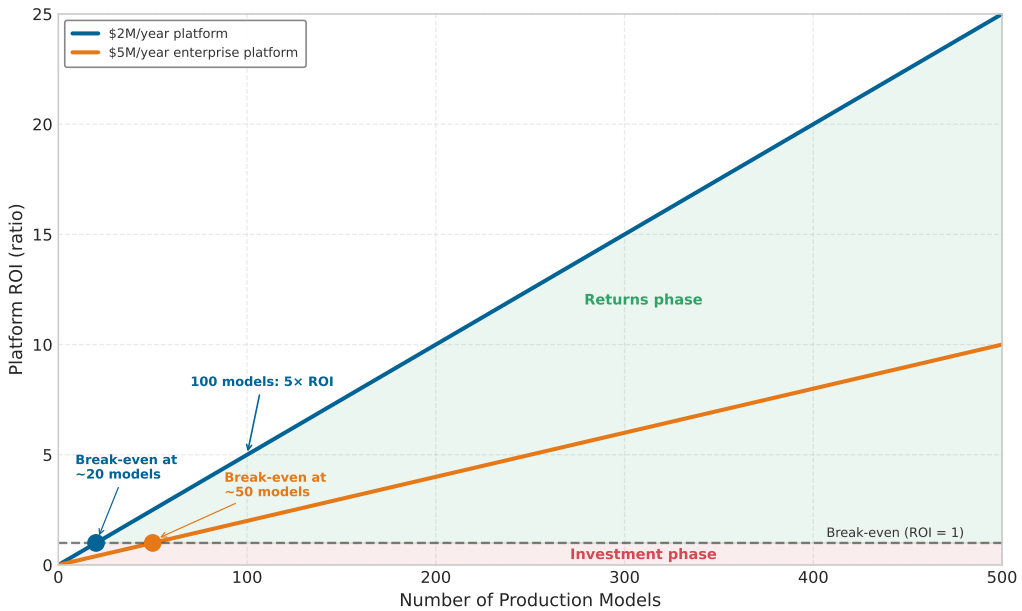


Figure 12.2: The Platform ROI Threshold: Platform return on investment as a function of model count, for two platform cost levels. A \$2M/year platform breaks even at approximately 20 models, while a \$5M/year enterprise platform requires roughly 50 models. Beyond break-even, ROI grows linearly—at 100 models, the $2 Mplatformdelivers 5×$ return.

Checkpoint 12.1: Platform ROI break-even

Equation 12.1 expresses platform return on investment as $N_{\text{models}} \times T_{\text{saved}} \times C_{\text{engineer}}/C_{\text{platform}}$. Figure 12.2 shows two cost curves and the linear ROI growth past break-even. Apply the formula to two concrete decisions.

- ☐ **Platform ROI**: For an organization with 30 models, each saving 200 engineer-hours per year at $150/hour, calculate the ROI of a $2M/year platform and a $5M/year enterprise platform.
- ☐ **Break-even count**: Using the same per-model savings ($30K/year, which is 200 hours at $150/hour), determine the model count where each platform breaks even, compare it to the rule-of-thumb 20 and 50 in figure 12.2, and infer the figure's implicit per-model savings.
- ☐ **Build now vs. wait**: For an organization with 8 models debating whether to build a $2M/year platform, use the linear ROI structure of equation 12.1 past break-even to state the strongest argument for building now and the future-growth assumption required for that argument.

Platform ROI is one lever; the cost of individual training runs and the capacity decisions that follow are another. A single 2,048-GPU H100 run makes that capacity question concrete and connects it to fleet economics, where utilization, checkpointing, and carbon accounting become platform-level decisions rather than isolated experiment costs.

12.2.2 Capacity planning and cost of training

Capacity planning for large-scale ML is an exercise in optimizing the economics of the GPU-hour. Consider a concrete case: a 30-day training run on a 256-node H100 cluster (2,048 GPUs) at an illustrative market rate of $2/GPU-hour represents a direct investment of roughly $2.95M. At this scale, the cost per training run (C_{run}) becomes a primary design lever that dictates the sizing of the entire fleet. Faster time-to-market pushes the planner toward more GPUs, but larger jobs also suffer diminishing parallel efficiency and a higher probability of hardware interruption. Total cost is therefore not merely a function of compute time; it must account for data staging, checkpointing, and the cost of recovery from interruptions (Chapter 7).

The drive for efficiency forces capacity to behave like a dynamic resource rather than a static allocation. Organizations must decide when to buy additional permanent capacity and when to recover the same effective throughput through better orchestration or model optimization. The decision also has an energy dimension: powering thousands of GPUs for months consumes enough electricity that financial cost and carbon cost move together. Capacity planning therefore becomes a strategic lever that balances performance, budget, energy use, and responsible engineering in the same decision. Cost visibility also sharpens the case for paying down technical debt: the same metrics that reveal runaway training costs expose the hidden cost of unversioned data, brittle pipelines, and manual toil.

12.2.3 Quantifying and managing ML technical debt

Technical debt at fleet scale is a prioritization problem: the platform must find which hidden dependency is slowing deployments, raising incident rates, or consuming the most engineering toil, and pay that down first. The four categories below (data, configuration, model, and infrastructure debt) locate where the drag originates, and quantifying each makes them comparable across the fleet so the worst offender can be addressed first.

Napkin Math 12.3: The maintenance dividend

Problem: A team spends 40 hours/month manually fixing "broken plumbing" (stale data, failed scripts, manual monitoring). A one-month intensive cleanup (160 hours) is projected to reduce this to 8 hours/month. Is the cleanup worth it over 3 years of model lifecycle?

Math:

1. **Status quo** (3 years): 36 months $\times$ 40 hours/month $\times$ $150/hr = $216000.
2. **Proactive Path**: (160 hours investment + 36 months $\times$ 8 hours/month) $\times$ $150/hr = $67200.

3. **Net Savings**: $216000 - $67200 = $148800.
4. **Dividend ratio**: 3.2×.

Systems insight: Proactive maintenance reduces total cost by a factor of 3.2× over the model lifecycle. In the ML Fleet, "Plumbing" is more important than "Pipes": an organization that ignores technical debt eventually spends its entire budget just keeping old models alive, leaving zero capacity for new development. The most successful teams treat refactoring as a high-yield investment, not a distraction.

The maintenance dividend becomes actionable only when the platform can compare unlike debts on a common operational scale. ML technical debt manifests in measurable symptoms that directly affect platform velocity and reliability (Sculley et al. 2015; Amershi et al. 2019).

12.2.3.1 Debt categories and measurement

The debt category matters because each failure mode leaves a different operational trace. Table 12.2 turns the taxonomy into a measurement map: data debt appears as incidents and manual pipeline intervention, configuration debt as unvalidated release surface, model debt as glue-code drag, and infrastructure debt as toil that consumes platform capacity.

Table 12.2: Technical Debt Measurement Map: Different debt types require different observable signals. A shared scoring process can compare them only after each category has been tied to the symptom and threshold that make the debt operationally visible.

Debt type	Operational symptom	Measurement signal	Warning threshold
Data debt	Unstable dependencies, missing versioning, weak validation	Data incidents per month and manual pipeline intervention	More than 10 data incidents/month or 30% manual runs.
Configuration debt	Ad-hoc files, duplicated parameters, absent validation	Deployment failures, lines of config, unvalidated parameters	More than 500 lines of unvalidated configuration per model.
Model debt	Glue code, undeclared consumers, tangled serving paths	Coupling score, undocumented consumers, trace time	More than 20% engineering time maintaining glue code.
Infrastructure debt	Brittle pipelines, manual deployment, environment drift	Toil hours, automation coverage, drift incidents	More than 50% platform capacity spent on toil.

12.2.3.2 Quantification metrics

Table 12.3 turns four debt metrics into a baseline-and-threshold map, pairing each symptom with the point at which it becomes an escalation signal.

Table 12.3: Technical Debt Prioritization Metrics: Operational debt becomes actionable when each metric has a healthy baseline and a threshold that turns latent friction into an engineering queue.

Metric	Signal	Healthy baseline	Warning threshold
Deployment velocity	Time from code commit to production deployment; exposes release friction.	Less than one day for inference code changes and less than one week for training changes.	More than two weeks indicates configuration complexity, brittle dependencies, or inadequate automation.
Incident rate	Reliability harm visible as incidents per 1000 deployments.	Fewer than 5 incidents per 1000 deployments.	More than 20 incidents indicates debt in testing, validation, or deployment procedures.
Toil percentage	Team capacity consumed by manual operational work.	Less than 20% of capacity spent on toil.	More than 50% indicates automation debt that prevents the team from improving the platform.

Metric	Signal	Healthy baseline	Warning threshold
Dependency staleness	Share of dependencies behind supported or current versions.	Less than 10% stale dependencies.	More than 30% indicates upgrade debt that increases security risk and limits performance improvements.

The worked example applies that threshold logic to rank debt by the capacity each fix returns to the platform team.

12.2.3.3 Worked example: ML debt audit and prioritization

An ML platform team supporting 40 production models with 15 engineers faces deployment velocity problems. New models require 6 weeks to reach production, frustrating both platform and model teams. The audit must rank the debts by how much platform capacity each one unlocks, not by which symptom is most visible.

The audit records the three debt categories in table 12.4 before scoring them. Each row pairs an operational symptom with the impact measure that makes the debt comparable across teams.

Table 12.4: ML Debt Audit Inputs: Initial observations for configuration debt, pipeline glue code, and monitoring debt before priority scoring. The table preserves the symptom, impact metric, technical measure, and estimated annual cost for each debt category.

Debt category	Symptom	Impact metric	Technical measure	Estimated annual cost
Configuration debt	Each model has custom YAML (YAML Ain't Markup Language) configuration files averaging 847 lines with no validation schema	35% of deployment delays result from late config errors	Manual config review required for every deployment	12 engineer-hours per deployment × 80 deployments/year = 960 hours
Pipeline glue code	Data preprocessing uses 23 different scripts with 62% code duplication	12 engineer-hours per week debugging pipeline breaks	No shared preprocessing library; each team implements custom logic	12 hours/week × 52 weeks = 624 hours
Monitoring debt	Each model uses ad-hoc monitoring, with no unified observability platform	Mean time to detect (MTTD) incidents is 4.2 hours	23 monitoring approaches across 40 models	Extended incident duration costs $50K per incident × 15 incidents/year = $750K

The scoring approach uses three criteria: impact severity, frequency, and resolution cost, each scored 1 to 3. Table 12.5 ranks the three debt categories observed earlier. Higher impact and frequency raise priority; higher resolution cost lowers it.

Table 12.5: Debt Prioritization Scoring: Three-criterion scoring (impact severity, frequency, and resolution cost) applied to three observed debt categories. Scores use Impact × Frequency/Resolution Cost, so slower or more expensive fixes are penalized. Configuration debt scores highest, justifying first-priority remediation.

Debt Category	Impact	Frequency	Resolution Cost	Total Score	Priority
Configuration	3 (High)	3 (Daily)	2 (Medium: 6 weeks)	4.5	**1st**
Monitoring	3 (High)	2 (Weekly)	2 (Medium: 8 weeks)	3	**2nd**
Pipeline Glue	2 (Medium)	2 (Weekly)	3 (High: 16 weeks)	1.3	**3rd**

The score makes configuration debt the first paydown target: build configuration schema validation and a templating system before addressing lower-scoring pipeline glue. The investment

requires 6 weeks of engineering effort to build the configuration system. It removes 35 percent of the recurring configuration-review toil: 12 engineer-hours per deployment $\times$ 80 deployments per year $\times$ 35 percent = 336 engineer-hours saved per year. At \$150/hour, that is \$50K in annual savings, so the investment pays back in about 8.6 months.

12.2.3.4 Decision framework

The general debt paydown decision extends the worked score with an explicit benefit estimate, where Impact captures severity, Frequency captures how often the debt appears, Benefit captures avoided operational cost, and Resolution Cost captures the engineering effort required:

$$\text{Paydown Priority} = \frac{\text{Impact} \times \text{Frequency} \times \text{Benefit}}{\text{Resolution Cost}} \tag{12.2}$$

The paydown-priority formula in equation 12.2 keeps automation work tied to operational value: high-impact, frequent, high-benefit fixes outrank expensive work whose payoff is speculative.

> Napkin Math 12.4: ROI of automation
>
> **Problem**: A team spends 10 hours of manual toil per model deployment. Investing 120 hours in a CI/CD pipeline is projected to reduce deployment toil to 0.5 hours. At 3 deploys/week, how long until the automation pays for itself?
>
> **Math**:
>
> 1. **Automation Cost**: 120 hours $\times$ \$150/hr = \$18000.
> 2. **Weekly Savings**: 9.5 hours/deploy $\times$ 3 deploys/week $\times$ \$150/hr = \$4275/week.
> 3. **Payback Period**: \$18000 / \$4275/week $\approx$ 4.2 weeks.
>
> **Systems insight**: Automation is a high-yield capital investment. A payback period of 4.2 weeks is an exceptional return on engineering time. In MLOps, "Toil" is the highest-interest technical debt an organization can carry: paying it down early yields massive dividends for the rest of the model's lifecycle.

The decision rule is asymmetric. Pay debt when this ratio exceeds the expected value from feature development; the configuration debt above qualifies because it has high impact (blocks deployments), high frequency (every deployment), high benefit (eliminates 35 percent of delays), and moderate cost (6 weeks). Defer debt when it is localized to a single team, frequency is low (monthly or less), system sunset is planned within 12 months, or resolution cost exceeds 6 months of engineering effort.

The same prioritization logic must become organizational habit, or the measured debt will reaccumulate between audits. Quantified debt first becomes a backlog item with affected systems, estimated impact, resolution cost, and priority score. Quarterly review then updates that backlog as platform needs change, keeping the ranking tied to current deployment velocity, incident rates, and toil rather than stale complaints.

A debt budget turns that ranking into capacity. Allocating 20 to 30 percent of sprint capacity to paydown makes remediation compete explicitly with feature work, while teams spending less than 10 percent on debt usually see debt grow faster than they can address it. Prevention moves the same reasoning upstream into code and design review: every proposed change should ask whether it introduces configuration complexity, hard-to-maintain data dependencies, or manual operational procedures, because preventing debt creation costs less than paying it down later.

12.2.4 How operations differ at scale

The operational requirements for multi-model platforms differ qualitatively from single-model operations. Table 12.6 contrasts these approaches across six dimensions, revealing that platform-scale deployment demands dependency-aware scheduling, monitoring must shift from *model-centric* to *system-centric* aggregation, and governance evolves from *team-specific* policies to *organization-wide* standards:

Table 12.6: Single-Model vs. Platform Operations: Six qualitative differences that emerge when scaling from one model to 100+. Deployment shifts from team-controlled rollouts to dependency-aware platform coordination, monitoring evolves from model-centric dashboards to system-level aggregation, and governance expands from team-specific policies to organization-wide automated enforcement.

Aspect	Single-Model Operations	Multi-Model Platform (100+)
Deployment	Simple rollout, team-controlled	Dependency-aware scheduling, platform-coordinated
Monitoring	Model-centric metrics	System-centric with model aggregation
Debugging	Local to model and data	Distributed tracing across model boundaries
Resource Management	Dedicated allocation	Shared pools with multi-tenant isolation
Governance	Team-specific policies	Organization-wide standards and automation
Organization	Single team ownership	Platform team plus consumer teams

The qualitative gap is most visible in deployment operations. Single-model deployment is straightforward: validate the new version, deploy to a canary, monitor for regressions, and proceed to full rollout. Platform-scale deployment must consider dependency ordering, where models that consume features from other models cannot be updated independently. Rollback coordination becomes essential, as reverting one model may require reverting dependent models. Resource contention arises when multiple deployments compete for GPU memory or network bandwidth. Blast radius management limits the impact of any single deployment failure.

For recommendation systems, this complexity is particularly acute. A typical recommendation request might involve 10–50 models executing in sequence or parallel: candidate retrieval models, ranking models, diversity filters, and business rule layers. Updating any component requires understanding its interactions with all others.

Monitoring requirements evolve similarly. At single-model scale, monitoring focuses on model-specific metrics: prediction accuracy, inference latency, and data drift indicators. At platform scale, this approach becomes untenable. With 100 models, 100 independent dashboards create information overload that prevents effective incident response.

Platform monitoring must therefore aggregate across models while maintaining the ability to drill down into specifics. This requires hierarchical metrics. Business metrics capture overall system health through revenue, engagement, and user satisfaction. Portfolio metrics aggregate model performance by domain or business unit. Model metrics track individual model accuracy, latency, and drift. Infrastructure metrics monitor GPU utilization, memory pressure, and network throughput.

12.2.4.1 Telemetry collection at scale

The transition to platform-scale observability requires a fundamental shift in telemetry paradigms at scale. When 10,000 edge nodes or hundreds of microservices generate logs simultaneously, the monitoring infrastructure itself can become a bottleneck, creating a "thundering herd" that overwhelms the network. Telemetry must be rigorously categorized and sampled to prevent the observability system from perturbing the production system. Table 12.7 separates the telemetry types by volume growth and operational use so the platform can choose what to collect continuously and what to sample.

Table 12.7: Telemetry Paradigms at Scale: Because logs and traces grow linearly with request volume, they must be aggressively sampled or aggregated at the edge, whereas metrics can be continuously pushed or pulled without overwhelming the network.

Telemetry Type	Definition	Volume	Primary Use Case
Metrics	Aggregated numerical data (counters, gauges, histograms).	Low (constant size)	Alerting, SLA tracking, and high-level dashboarding.
Logs	Discrete, timestamped text records of specific events.	High (scales with requests)	Post-incident root cause analysis and auditing.
Traces	End-to-end request paths across distributed microservices.	Very High	Diagnosing latency bottlenecks and distributed failures.

Notice in table 12.7 that volume grows from constant (metrics) through linear (logs) to super-linear (traces) with request rate, which is why effective platforms present high-level metric dashboards by default and enable investigation into lower levels only when anomalies are detected.

12.2.5 Model-type operations diversity

Beyond scale considerations, different model types require fundamentally different operational patterns. The practices appropriate for deploying a large language model are entirely inappropriate for a fraud detection system, and vice versa. The archetype taxonomy in section 1.6.1 helps interpret the model-type operational requirements in table 12.8: LLMs demand staged rollouts over days to weeks with hours-long rollback windows, while fraud detection requires hourly updates with seconds-fast rollback to address adversarial dynamics.

Table 12.8: Model-Type Operational Requirements: Update frequency, deployment patterns, and rollback speeds vary by model type due to differing risk profiles. LLMs require monthly staged rollouts with hours-to-days rollback due to quality regression risks, while fraud detection demands hourly updates with seconds-fast rollback to counter adversarial dynamics.

Model Type	Update Frequency	Deployment Pattern	Primary Risk	Rollback Speed
Archetype A (GPT-4/Llama-3)	Monthly to quarterly	Staged, careful	Quality regression, safety	Hours to days
Archetype B (DLRM at Scale)	Daily to weekly	Shadow, interleaving	Engagement drop	Minutes
Fraud Detection	Hourly to daily	Rapid with instant rollback	False negatives	Seconds
Vision (Classification)	Weekly to monthly	Canary	Accuracy regression	Minutes
Search Ranking	Daily	A/B with holdout	Relevance degradation	Minutes

The table is a risk-to-cadence map, not a model catalog. Large language models sit at the slow end because size, cost, and subtle quality regressions make every release a high-stakes event. A minor degradation in response quality might not appear in automated metrics but could erode user satisfaction measurably, so LLM updates typically involve extended shadow deployment, human evaluation alongside automated metrics, staged rollouts over days or weeks, and safety evaluation before any production exposure.

The cost of regression becomes concrete when one model must preserve every capability at once:



Lighthouse 12.1: Archetype A (GPT-4/Llama-3): cost of regression

Archetype A (GPT-4/Llama-3), the general-purpose LLM introduced in section 1.6.1, faces the "Generalist's Dilemma." Because the model serves millions of distinct use cases, a fine-tuning update to improve Python coding might silently degrade haiku writing. The release gate therefore combines broad capability benchmarks such as Massive Multitask Language Understanding (MMLU) and HumanEval with policy-based safety checks before any production rollout.

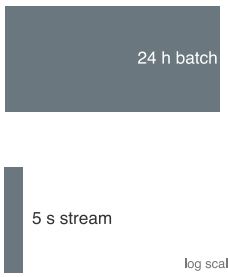
The operational cadence for LLMs is measured in weeks to months, with each update treated as a significant event requiring cross-functional coordination. Recommendation systems operate at the opposite end of the operational spectrum because freshness, not deployment fear, often dominates the risk profile (Steck et al. 2021; Gomez-Uribe and Hunt 2015). User preferences shift continuously, new content arrives constantly, and stale recommendation features risk degrading relevance before the next batch update catches up.

Recommendation operations therefore emphasize four patterns:

- **Continuous training**: Pipelines produce daily or weekly model updates.
- **Interleaving experiments**: Multiple model variants are compared on the same requests.
- **Rapid iteration**: Changes can reach production within hours.

- **Rigorous A/B testing**: Statistical infrastructure separates real engagement changes from noise.

A key metric that captures this operational urgency is feature freshness latency, which measures how quickly user actions propagate into the model's predictions.



Streaming closes the freshness lag that batch leaves open.

Example 12.1: Feature freshness latency

Scenario: A user clicks a "Basketball" video. The time required for their feed to show more basketball content is the feature freshness latency.

Math:

$$T_{\text{freshness}} = T_{\text{available}} - T_{\text{event}}$$

Setup:

- Batch Pipeline (Daily): Events are aggregated at midnight.
  - $T_{\text{freshness}} \approx$ 12–24 hours.
  - **Impact**: User leaves session before recommendations update.
- Streaming Pipeline (Real-time): Events flow through Kafka/Flink to Feature Store.
  - $T_{\text{freshness}} \approx$ 1–5 seconds.
  - **Impact**: Next page load reflects the interest.

Systems insight: For session-based recommendations, moving from Batch ($T_{\text{freshness}} \approx 24$ h) to Streaming ($T_{\text{freshness}} \approx 5$ s) often yields a 10–20 percent lift in engagement, justifying the increased infrastructure cost.

The key insight is that recommendation operations are fundamentally about ensemble management. A single recommendation request might invoke ten to fifty distinct models, each requiring its own update cadence while maintaining coherent behavior as a system.

Fraud detection systems face a distinct set of operational challenges because their inputs are shaped by adaptive adversaries. Fraudsters actively probe systems to find exploits, then rapidly shift tactics once detected. A fraud model that cannot adapt within hours provides a window of vulnerability. These adversarial dynamics impose four operational requirements:

- **Frequent updates**: Models update hourly or more often in response to emerging patterns.
- **Instant rollback**: Serving can revert within seconds when false positive rates spike.
- **Shadow scoring**: All transactions are scored by candidate models for rapid comparison.
- **Feature velocity monitoring**: Sudden distribution shifts are detected before adversaries exploit them.

The risk profile is asymmetric. False negatives (missed fraud) cause direct financial losses, while false positives (legitimate transactions blocked) cause customer friction. Operations must balance these competing concerns in real time.

These diverse operational patterns reflect a single underlying principle: risk profile determines operational cadence. LLMs carry large deployment-risk surfaces, including bias, misuse, and environmental harms that motivate careful risk-benefit analysis before release (Bender et al. 2021). Recommendation systems operate rapidly because stale models lose relevance faster than bad updates can cause damage. Fraud detection operates continuously because adversaries do not wait for scheduled deployments. Understanding this principle enables teams to design appropriate operational practices for new model types by analyzing their risk characteristics rather than copying patterns from superficially similar systems.

The rest of the chapter stays at the enterprise-fleet level: shared infrastructure for many interacting models, not the foundational move from ad-hoc notebooks to automated single-model pipelines. The question is when that shared layer becomes cheaper and safer than allowing each team to build its own operational stack.

12.2.6 Platform team justification

A dedicated ML platform team is justified when fragmentation costs more than the shared infrastructure needed to replace it. The decision therefore combines quantitative factors (cost savings, velocity improvements) with qualitative factors (consistency, governance, talent retention).

The ROI calculation presented earlier provides the primary quantitative argument, and the supporting metrics show where shared ownership converts into fleet-wide savings: infrastructure efficiency, time to production, and incident reduction. Infrastructure efficiency improves through shared GPU clusters, which achieve 70 to 80 percent utilization vs. 30 to 40 percent for dedicated per-team resources. For an organization with 100 GPUs at $2 per GPU-hour, moving from 35 percent to 75 percent effective utilization saves approximately $930,000 annually when the same 35 active GPU-equivalents can be served by fewer provisioned GPUs. Time to production decreases through platform abstractions that reduce the time from trained model to production deployment; if this acceleration enables one additional high-value model to reach production per quarter, the business value typically exceeds platform costs. Incident reduction follows from standardized deployments and monitoring, with mature platforms often reducing ML-related incidents by 60 to 80 percent and translating that reduction into both direct cost savings and improved user experience.

The qualitative case follows from the same fragmentation cost. Platform failures are coordination failures as much as infrastructure failures, and shared ownership improves four coordination surfaces:

- **Consistency**: Standardized practices ensure all models meet baseline quality standards for monitoring, rollback capability, and documentation.
- **Knowledge sharing**: Centralized teams make operational expertise available to all model teams rather than leaving it siloed.
- **Career development**: Platform roles provide career paths for ML engineers interested in infrastructure.
- **Governance readiness**: Platform-level controls provide the foundation for compliance as regulatory requirements for AI grow.

The decision to establish a platform team typically occurs when organizations recognize that the alternative, allowing fragmentation to continue, imposes costs exceeding the platform investment. This recognition often follows a significant production incident that revealed cross-model dependencies or operational gaps. The resulting economics show why platform operations become more valuable as the model fleet grows.

> **Systems Perspective 12.2: Platform returns improve with every model added**
>
> Because equation 12.1 scales the numerator linearly with N_{models} while platform cost stays roughly fixed, the ROI ratio rises with every model the platform serves: the per-model savings are recovered against a denominator that barely moves. The benefit is not a faster-than-linear payoff but a ratio that keeps climbing past break-even, which is why organizations that delay platform investment accumulate operational debt that becomes progressively more expensive to address as the fleet grows.

12.3 Fleet Economics and Utilization

While operational tooling saves engineering hours, the financial ledger of a deployed model fleet is dominated by utilization. A GPU that is idle between bursts, reserved for failed rollouts, or stranded behind a scheduling bottleneck costs the same as a GPU serving useful predictions. Deployment practices therefore cannot scale economically until the platform can see where capacity is used, where it is reserved, and where it is wasted.

Engineering constraints are ultimately economic constraints: every design decision trades cost against performance, and the "right" infrastructure is the one that maximizes useful computation per dollar over the system's lifetime. Using the $350,000 per 8-GPU DGX H100 node assumption, a 10,000-GPU cluster represents about $437.5M in node hardware CapEx before networking, facility, staffing,

and maintenance costs. The purchase price, however, is only the beginning. Over a typical three-year hardware lifecycle, power, cooling, facility, networking, staffing, and maintenance determine whether the fleet is an asset or an idle liability. Total cost of ownership (TCO)[2] matters here because it turns utilization into the central operating invariant behind the broader ML-lifecycle accounting developed in section 12.7.11.

The economics of ML infrastructure differ from traditional IT in three fundamental ways. Accelerators depreciate quickly, power is a first-order operating cost, and utilization sensitivity is extreme: the same fleet can be a brilliant investment at 80 percent utilization or a financial disaster at 20 percent utilization, with no change in hardware or facility costs. For operations at scale, this is the central lesson. The platform must keep expensive capacity doing useful work while still preserving enough headroom for failures, rollbacks, and traffic spikes.

[2] **TCO (Total Cost of Ownership)**: A financial framework, formalized by Gartner in the 1980s for IT procurement, that sums CapEx (one-time acquisition) and OpEx (recurring operation) over a system's lifecycle. For ML clusters, TCO analysis is uniquely consequential because power, cooling, staffing, facility, and network costs materially change the three-year economics, and a 60-percentage-point swing in utilization (20 percent to 80 percent) can flip the build-vs.-buy decision entirely without changing any hardware specification.

12.3.1 Cost centers as operating constraints

The total cost of an ML cluster decomposes into two broad categories. **Capital expenditure (CapEx)** covers the one-time costs of building the infrastructure: accelerators, servers, networking equipment, facility construction, and installation. **Operational expenditure (OpEx)** covers the recurring costs of running it: electricity, cooling, network bandwidth, staffing, maintenance, and software licenses. For a large on-premises cluster, the approximate breakdown is as follows.

Table 12.9 separates the cost stack into capital and operating centers, each with different utilization implications.

Table 12.9: Frontier Training Cost Stack: Capital costs are dominated by accelerators, networking, and facilities, while operating costs are dominated by electricity and specialized staffing.

Cost center	Cost class	Typical share	Included components and implications
Accelerators and servers	CapEx	50–60% of CapEx	GPUs or TPUs, host servers, baseboard management controllers, and local storage; this category dominates upfront purchase and depreciation.
Networking	CapEx	10–15% of CapEx	InfiniBand switches, HCAs, cables, and optical transceivers; a fat-tree fabric for 1,024 GPUs can cost $10–20 million.
Facilities and cooling	CapEx	15–25% of CapEx	Building construction or retrofit, transformers, UPS systems, PDUs, and cooling plant; liquid cooling adds 10–15% to facility costs but reduces long-term OpEx.
Electricity	OpEx	60–70% of OpEx	At $0.07/kWh, a 1,024-GPU H100 cluster consuming 1 MW including cooling at PUE 1.1 costs about $615,000 per year.
Staffing and maintenance	OpEx	20–30% of OpEx	System administrators, hardware technicians, replacement parts, and software license fees.

Consider a 175B-parameter frontier model as the running cost example for this analysis. For this model, a minimum viable training cluster requires approximately 1,024 H100 GPUs spread across 128 nodes to complete a training run in 2–4 weeks. Evaluating the TCO over a three-year lifecycle reveals a stark utilization dependency. The hardware CapEx dominates at $44.80M ($350,000 per node), supported by a $5 million investment in a two-tier InfiniBand fat-tree network and a proportional $10 million facility allocation. Operational costs add approximately $1.5 million annually for electricity (at $0.07/kWh with a PUE of 1.1) and specialized staffing, bringing the three-year total to roughly $64 million. If this dedicated cluster only trains six large models per year, the effective cost per run is $3.5 million. Renting the same 1,024-GPU allocation in the public cloud at $4.00 per GPU-hour costs about $1.3–2.7 million per run (1,024 times 336–672 hours $\times$ $4), or $24–48 million over three years for six runs per year. The cloud is cheaper for this bursty cadence because the organization is not paying for idle weeks. The economic advantage of owning hardware only materializes at *continuous utilization*: if the cluster runs 24/7 (supporting training, inference, fine-tuning, and experimentation), the effective on-premises cost drops to approximately $2.40 per GPU-hour, significantly undercutting the cloud rate. This utilization dependency is the central tension in every build-vs.-buy analysis.

12.3.2 Utilization as the economic invariant

The most consequential economic question is whether the fleet can sustain enough useful load to amortize fixed capacity. Build-vs.-buy is one expression of that question, but the same invariant appears inside a deployed platform: reserved capacity, fallback pools, regional replicas, and canary headroom all become economical only when their reliability value justifies their idle cost. Figure 12.3 makes the utilization dependency visible by plotting cumulative cost over time for owned and rented capacity.

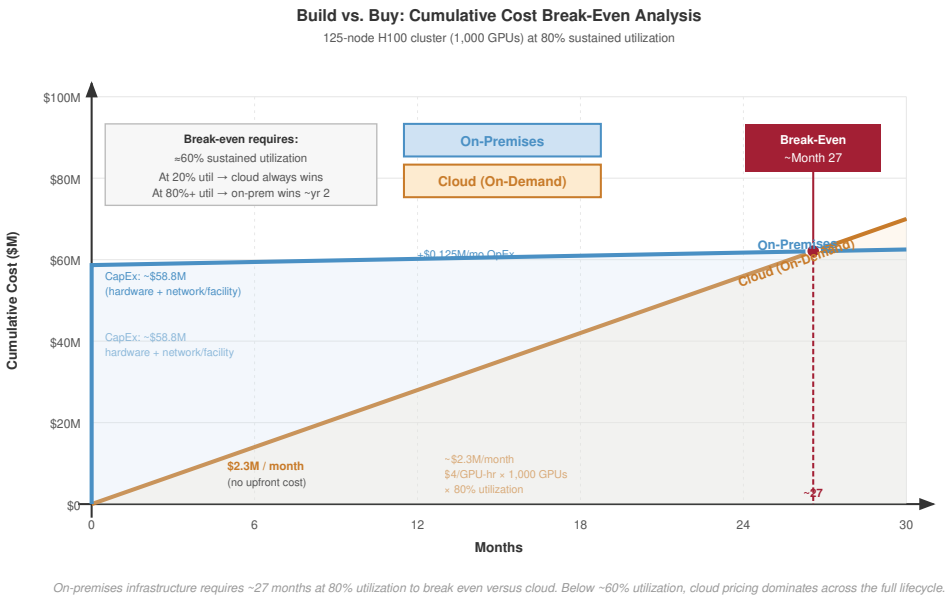


Figure 12.3: TCO: Build vs. Buy: Cumulative cost over a 30-month horizon for a high-utilization H100 reference cluster. The plotted curves compare an upfront on-premises CapEx step plus monthly OpEx against linear cloud rental cost.

The plotted reference uses 1,024 H100 GPUs across 128 nodes at 80 percent sustained utilization. The on-premises curve starts at $59.80M in upfront CapEx and adds about $1.5M per year, while cloud rental at $4/GPU-hour grows by about $2.39M per month. Over 30 months, the cloud line reaches $71.8M, and the on-premises line crosses it around 26.4 months.

Cloud providers charge by the GPU-hour. At approximately $4.00 per H100-hour, a single 8-GPU node running at 80 percent utilization costs $224,256 per year. An on-premises DGX H100 node costs approximately $350,000 to purchase. Amortized over three years and combined with electricity costs of $4,519 per node per year, the total annual on-premises cost is approximately $121,186 per node. Solving the break-even equation with electricity scaling at the same utilization, on-premises infrastructure becomes favorable when sustained utilization exceeds roughly 42.5 percent.

The same calculation also explains why cloud pricing cannot be read as a simple price list. Reserved capacity lowers the per-hour rate by 40–60 percent, but it converts optionality into commitment. Spot or preemptible capacity can be 60–80 percent cheaper, but the discount arrives as interruption risk. Cheap capacity is therefore useful only when the training or serving system already has checkpointing, elasticity, and admission control that keep interruptions from becoming lost work or user-visible failures.

Owned capacity moves a different set of risks into the platform. Facility construction can add $500–1,000 per kW of IT capacity, and networking, staffing, maintenance, and hardware obsolescence continue to accrue even when the GPUs are idle. A three-year-old GPU may be economically stranded if the next generation delivers 3× more performance per watt. The engineering question is therefore not whether one purchasing channel is universally cheaper; it is whether the workload mix can keep fixed capacity useful enough to compensate for its lifecycle and operating risks.

Hybrid designs follow from the same invariant. Owned infrastructure can carry predictable baseline work, while cloud allocations absorb peak demand, early hardware access, or bursty training campaigns. The hybrid split is not a compromise between two vendor categories; it is a scheduling policy over capital risk, utilization, and deadline risk.

For the 175B model, cadence is the variable that determines the answer. A team that trains one large model per year and serves it for the remaining eleven months may use owned training hardware only 15–20 percent of the time. A team running back-to-back experiments, hyperparameter sweeps, fine-tuning jobs, and model variants can sustain 70–80 percent utilization on the same fleet. The hardware has not changed; the workload mix has changed the economics. In the middle regime, the platform owns enough capacity for the continuous baseline and bursts to the cloud for the large training runs that temporarily require 5–10$\times$ the baseline allocation.

12.3.2.1 Operational complexity

TCO also understates the operational complexity of owning the fleet. Hardware maintenance means diagnosing failed GPUs, NVLink cables, InfiniBand HCAs, and cooling components before a local fault becomes a cluster-wide slowdown. A 10,000-GPU cluster experiencing one GPU failure every five hours, as predicted by MTBF analysis at the canonical 50,000-hour per-GPU MTBF, needs replacement paths measured in hours rather than days. Section E.1.1 supplies the per-component MTTF baselines and converts them into the fleet-level failure rate behind this five-hour figure, so the staffing argument traces back to a measured reliability budget rather than an assumed one.

The software side creates the same coupling. CUDA, GPU drivers, InfiniBand drivers, container runtimes, schedulers, monitoring systems, and training frameworks all participate in the achieved utilization number. A driver/toolkit incompatibility can silently degrade performance, while a scheduler or monitoring gap can strand accelerators behind failed jobs. Large training clusters commonly need 5–15 infrastructure engineers per 10,000 GPUs because performance optimization is continuous: as models, frameworks, and hardware configurations change, communication patterns, memory pressure, and parallelism strategies change with them. Section B.6.1 works the underutilized fleet through the computation, communication, and coordination axes, giving the performance engineer a framework to locate where the lost utilization actually goes rather than tuning blindly.

Cloud providers charge a margin above raw hardware and electricity partly because they absorb this operating burden. The point is not that cloud is simpler in every technical sense; it exposes a different interface. The platform either pays for internal expertise in GPU kernel behavior, InfiniBand operations, distributed-systems debugging, liquid cooling, and power infrastructure, or it pays a provider to hide some of that machinery behind a service contract. In both cases, the economics are still governed by useful work per dollar.

12.3.2.2 From TCO to total value of ownership

A more complete framework than TCO is Total Value of Ownership (TVO), which includes the value generated by infrastructure rather than only its cost. Two clusters with identical TCO may create different outcomes if one reaches a result two weeks earlier, sustains higher scaling efficiency, starts jobs faster, checkpoints more cheaply, or serves the resulting model at lower cost per token. The value terms are hard to price exactly, but they are not optional in systems reasoning: time-to-result affects product deadlines, scaling efficiency affects final model quality under a fixed training window, experimentation velocity affects the number of hypotheses the team can test, and inference efficiency can dominate training cost over a multi-year deployment.

This value-oriented perspective changes the infrastructure decision from cost minimization to constraint management. A cheaper cluster that slows the research cycle or produces a model with higher serving cost may be more expensive over the model lifecycle. Conversely, a more expensive training platform can pay for itself if it enables a smaller or more efficient deployed model.

The inference dimension of TVO deserves particular emphasis because it often dominates the total economic picture. Training the 175B model is a one-time cost – even at $5 million per training run, it is a bounded expenditure. Serving the trained model, however, is an ongoing operational cost that accumulates indefinitely. A popular LLM serving 10 million queries per day, with each query generating an average of 500 tokens, processes 5 billion tokens daily. At an inference cost of $2.00 per million tokens on H100 hardware, the daily serving cost is $10,000, or approximately $3.6

million per year. The cumulative inference cost exceeds a $5 million training run after roughly 17 months, and exceeds it several times over during a multi-year deployment. This inversion means that infrastructure decisions optimized for training (maximizing TFLOP/s per dollar) may be suboptimal for the model's total lifecycle cost. An organization that spends an additional $2 million on training infrastructure to produce a model that is 20 percent more efficient at inference (through better architecture search enabled by faster experimentation) can recover that investment over a three-year serving lifetime at this traffic level.

Napkin Math 12.5: The 10,000-GPU cluster

Consider a cluster of 1,250 DGX H100 nodes (10,000 GPUs) for training the 175B model.

On-premises (3-year lifecycle):

- **Hardware CapEx**: 1,250 × $350,000 = $437.5M
- **Network CapEx**: ~$25M (InfiniBand fat-tree fabric)
- **Facility CapEx**: ~$75M (liquid-cooled data center hall)
- **Annual Electricity**: simplified GPU-only estimate: 10,000 GPUs × 700 W × PUE 1.1 × 8,760 h/year × $0.07/kWh = $4.7M/year
- **Annual Staffing**: ~$5M/year
- **3-year total**: $537.5M + 3 years × $9.7M = ~$566.7M

Cloud rental (3-year rental at 80 percent utilization):

- **Annual Cost**: 10,000 GPUs × $4/GPU-hour × 8,760 h/year × 0.80 = $280.3M/year
- **3-year total**: ~$841.0M

Systems insight: Utilization is the binding variable in fleet TCO. On-premises saves approximately $274.3M over three years at 80 percent utilization, but the savings disappear if utilization drops below roughly 53.9 percent because the on-premises hardware still incurs facility and staffing costs regardless of load.

That utilization threshold explains why infrastructure scale can become a competitive advantage.

Systems Perspective 12.3: The infrastructure moat

Building a 10,000-GPU cluster saves hundreds of millions over cloud rental, but only if the organization has enough workloads to keep it busy. Large technology companies with continuous training pipelines, frequent model refreshes, and massive inference workloads achieve 70–90 percent utilization, making on-premises infrastructure highly cost-effective. Smaller organizations with sporadic training needs may achieve only 20–30 percent utilization, making cloud rental cheaper despite the higher per-hour cost. This dynamic creates an infrastructure moat: organizations with scale can afford to build, and building makes scale cheaper, which enables more ambitious models, which require more infrastructure. The gap compounds over time, giving high-utilization organizations a structural advantage when training the largest models.

12.3.3 Depreciation and lifecycle

ML accelerators depreciate faster than any other category of IT equipment. Traditional servers have useful lifetimes of 5–7 years; network switches last even longer. ML accelerators become economically obsolete in 3–4 years because each new generation delivers more performance per watt. Under the low-precision efficiency basis in table 12.10, a V100-era fleet draws similar per-GPU power to an H100-class fleet but receives only about 1/6.8 the throughput per watt. The electricity cost per unit of useful computation is therefore about 6.8× higher than a team using the H100-class reference hardware.

The economic impact of this depreciation is significant. A V100 GPU that cost $10,000–12,000 in 2018 could be purchased on the secondary market for $2,000–3,000 in 2023, a depreciation of 70–80 percent in five years. An A100 GPU that cost $15,000–20,000 in 2021 traded for $8,000–12,000 in 2024, after just three years. These depreciation rates are far steeper than traditional IT equipment, reflecting the rapid pace of accelerator innovation.

Rapid depreciation turns refresh policy into a scheduling problem. Most organizations depreciate ML accelerators over three years for accounting purposes, even if the hardware physically functions longer. Replacing the entire fleet at once creates a large CapEx spike, while staggered refresh lowers peak expenditure but leaves the scheduler with mixed-generation hardware. Training frameworks must then handle different compute capabilities, memory capacities, and communication bandwidths inside one fleet, so the accounting choice becomes a software and placement constraint.

The same lifecycle logic determines where older accelerators remain useful. A GPU that is inefficient for the largest training runs may still serve smaller models, fine-tuning jobs, or memory-bound inference economically because memory bandwidth improves more slowly than peak arithmetic throughput. Resale, repurposing, and capacity-sharing arrangements are all attempts to preserve utilization while the hardware's market value falls. They work only when the platform can route work to the right generation without hiding performance cliffs from users.

Depreciation is most punitive for bursty workloads. If a team trains one large model per year, the GPUs lose value during the idle months even though they perform no useful computation. Cloud capacity amortizes the same depreciation across many customers, while owned capacity must amortize it across the owner's own workload mix. That is why the utilization invariant appears again: the hardware must either stay busy or be cheap enough to strand.

For the 175B model, the depreciation calculus is stark. Under the $350,000-per-8-GPU-node assumption, a 1,000-GPU H100 cluster costs $43.75 million before network and facility costs. If that cluster has a resale value of approximately $7–10 million after a three-year refresh window because newer accelerators deliver 3–4× the performance per watt, its net hardware depreciation is $33.75–36.75 million. If the cluster trained only two large models during its lifetime, each model effectively cost $16.9–18.4 million in depreciated hardware alone – before accounting for electricity, staffing, or facility costs. If the same cluster ran continuously at 80 percent utilization for three years (training, fine-tuning, inference, and experimentation), the depreciated hardware cost per GPU-hour drops to approximately $1.61–1.75 after resale, still well below the cloud rate. The depreciation math reinforces the central lesson of TCO analysis: utilization is the single most important variable in determining whether owned infrastructure is economically viable.

12.3.4 Power efficiency trajectory

The trajectory of power efficiency across accelerator generations provides a quantitative framework for making cluster refresh decisions. Replacing an older cluster with a newer generation can pay for itself through electricity savings alone, particularly at scale.

Table 12.10: Power Efficiency Across GPU Generations: Each generation delivers substantially more computation per watt, meaning that for a fixed power budget, newer hardware provides multiplicatively more throughput. A facility that draws 10 MW can train models roughly 10× faster with B200s than with V100s, without any increase in electricity cost.

Generation	Peak TFLOP/s (best precision)	TDP (W)	TFLOP/s per watt	Relative Efficiency
V100 (2017)	125 TFLOP/s (FP16)	300 W	0.42 TFLOPs/s/W	1×
A100 (2020)	312 TFLOP/s (FP16)	400 W	0.78 TFLOPs/s/W	1.9×
H100 (2022)	1,979 TFLOP/s (FP8)	700 W	2.83 TFLOPs/s/W	6.8×
B200 (2024)	4500 TFLOP/s (FP8)	1000 W	4.50 TFLOPs/s/W	10.8×

As table 12.10 shows, the implication is that hardware refresh cycles are about getting more work per dollar of electricity, not just more FLOP/s. At scale, the electricity savings from upgrading to a

more efficient generation can amortize a significant fraction of the new hardware's purchase cost within the first year. This economic dynamic drives the rapid depreciation of ML accelerators: a three-year-old GPU is not just slower than a newer generation; it is more expensive to operate per unit of useful computation.

Consider a concrete refresh scenario. An organization operating 1,000 V100 GPUs (300 W each, 0.42 TFLOPs/s/W) consumes 300 kW of IT power for 125,000 TFLOP/s of aggregate throughput. Replacing them with 1,000 H100 GPUs (700 W each, 2.83 TFLOPs/s/W) increases power consumption to 700 kW but delivers 1,979,000 TFLOP/s, a 15.8× throughput increase for a 2.3× power increase.

Alternatively, the organization could match the V100 fleet's throughput with roughly 63 H100 GPUs, consuming only 44 kW and freeing 256 kW of power capacity for other workloads. The better strategy depends on whether the organization is throughput-constrained (wants to train larger models) or power-constrained (has a fixed electrical budget).

Both scenarios demonstrate that generational efficiency improvements reshape the economics of the fleet. The power-constrained case is particularly instructive: a data center with a fixed 300 kW power budget can deliver about 6.8× more computation by replacing V100s with H100s, even though it can only install 43 percent as many GPUs. The 15.8× throughput gain applies to a same-GPU-count refresh that raises power from 300 kW to 700 kW. Power, not procurement budget, is increasingly the binding constraint for fleet expansion.

The interplay between CapEx and OpEx also shapes procurement strategy. Cloud providers amortize their hardware over shorter periods (often 18–24 months) because they can sell older-generation instances at lower prices to price-sensitive customers, extracting residual value. On-premises operators typically amortize over 3–5 years, accepting that the hardware's relative performance declines over time.

Some organizations adopt a hybrid approach: running baseline workloads on owned infrastructure for cost efficiency and bursting to the cloud for peak demand or for access to newer hardware before committing to a large purchase. This hybrid model fits mid-sized AI companies that have a steady-state training workload, which justifies owned hardware, but periodically need 2–3× their base capacity for new model training campaigns.

The power efficiency trajectory has a direct implication for the 175B model's training economics, and a V100-versus-H100 comparison makes that implication concrete. Training on 1,000 V100 GPUs would require approximately 300 W each (300 kW IT power for 1,000 GPUs) and take roughly 8 months (given the V100's lower throughput). Training on 1,000 H100 GPUs requires 700 W each (700 kW IT power for 1,000 GPUs) but completes in approximately 2–4 weeks. The H100 cluster consumes 2.3× more power per unit time but finishes 8–16× faster, resulting in a net energy reduction of 3.5–7× for the same training run. When electricity costs $0.07/kWh, the V100 training run costs approximately $120,000 in electricity while the H100 run costs approximately $25,000. The newer hardware is simultaneously faster, cheaper to operate, and more energy-efficient – a rare alignment that makes hardware refresh decisions straightforward for organizations with the capital to invest.

12.3.5 Capacity lead time as operational risk

The economics of ML infrastructure are not purely about hardware specifications and electricity rates. Capacity lead time is itself an operational risk: a platform that cannot acquire, reserve, or free capacity quickly enough may miss model launches, delay rollback-safe migrations, or run without the headroom needed for incident response.

The GPU supply chain is unusually concentrated. NVIDIA holds approximately 80–90 percent of the data center GPU market for ML workloads. TSMC manufactures essentially all high-end GPU dies. A small number of companies (SK Hynix, Samsung, and Micron) produce the HBM stacks. This concentration means that a disruption at any single point in the supply chain, whether a natural disaster at a fab, an equipment failure at an HBM manufacturer, or a geopolitical event affecting chip exports, can delay GPU deliveries across the entire industry.

The practical consequence of this concentration is that capacity changes slowly at large scale. Procurement timelines for large deployments (1,000+ GPUs) typically span 6–12 months from purchase decision to first usable capacity, and demand spikes can stretch that horizon further. For deployed fleets, that delay changes operations: the platform must forecast model growth, reserve

refresh capacity, and maintain fallbacks before demand arrives rather than after dashboards turn red.

For the 175B model, the planning lesson is that hardware access becomes part of the deployment schedule. A minimum viable 1,000-H100 allocation represents approximately $44 million in node hardware under the $350,000-per-8-GPU-node assumption used above, but the operational risk is not only the purchase price. If the allocation arrives a quarter late, the platform may lack capacity for fine-tuning, shadow traffic, canary expansion, or rollback-safe serving during the model launch window. Capacity planning is therefore a reliability control, not just a procurement function.

12.3.6 Cloud capacity as an operations constraint

For organizations that choose the cloud path, the provider absorbs hardware ownership but exposes capacity, topology, and interruption risk as operating constraints. The specific instance types and pricing change frequently, so the durable distinction is not vendor branding; it is how the cloud allocation affects communication, scheduling, and fault tolerance.

First, the interconnect fabric determines whether distributed jobs and multi-node inference paths retain the scaling assumptions used in design. InfiniBand-like fabrics and Ethernet-based fabrics have different fixed-latency and per-byte costs; section D.1 separates those terms so the platform can decide whether the topology penalty matters for a given workload.

Second, provisioning granularity determines how much topology the platform controls. Individual instances maximize flexibility but force the team to assemble placement, networking, and scheduling policy. Pod-level allocations provide a preconnected unit but reduce composition freedom.

Third, custom accelerator options apply the same generality-efficiency trade-off introduced at the silicon level: lower cost per operation for supported workloads, but reduced flexibility for nonstandard architectures. The choice mirrors the on-premises accelerator decision, except that the exit path is a cloud migration rather than a hardware refresh.

The pricing models across providers share a common risk structure. On-demand instances buy flexibility at the highest per-hour cost. Reserved instances buy lower unit cost through commitment. Spot or preemptible instances buy the lowest unit cost by accepting interruption risk. The platform choice is therefore a fault-tolerance decision: cheap capacity is useful only when checkpointing, elasticity, and admission control keep interruptions from becoming user-visible failures.

The economics of spot instances deserve particular attention because they can dramatically reduce training costs for organizations with the engineering sophistication to exploit them. At a 70 percent discount, spot H100 instances cost approximately $1.20 per GPU-hour instead of $4.00. For the 175B model, the 1,000-GPU, 2–4 week workload consumes approximately 336,000–672,000 allocated GPU-hours. The on-demand cost is therefore about $1.3–2.7 million, while spot pricing would reduce the bill to roughly $0.4–0.8 million before accounting for interruptions. However, the stochastic nature of preemption transforms training from a deterministic process into a fault-tolerance engineering problem. If the cloud provider reclaims 5 percent of the nodes mid-training, a standard training job crashes instantly. Capturing spot economics requires an elastic training framework (such as TorchElastic) that can dynamically rebalance the computation graph when nodes are added or removed. The economic viability hinges on the **checkpoint tax**: if the system must checkpoint every 10 minutes to limit data loss from preemption, and each checkpoint takes 30 seconds, approximately 5 percent of the "cheap" compute is consumed by I/O overhead. There exists a break-even point where the frequency of preemption events combined with checkpoint overhead makes spot instances more expensive in wall-clock time than reserved instances, despite the lower hourly rate.

A critical consideration for cloud-based ML infrastructure is **networking between instances**. Unlike on-premises clusters where the network topology is custom-designed, cloud instances share the provider's network fabric with other tenants. Cloud providers may offer placement groups or dedicated networking fabrics that reserve InfiniBand-like or equivalent bandwidth between instances within the same group.

However, cross-group or cross-zone communication may traverse shared infrastructure with lower bandwidth and higher latency. Training frameworks that span multiple placement groups or availability zones must account for this heterogeneous bandwidth topology, a challenge that does not arise in dedicated on-premises clusters. The practical consequence is that cloud-based training

jobs must be sized to fit within a single placement group whenever possible, as crossing group boundaries can reduce scaling efficiency by 20–40 percent.

For the 175B model, the cloud path presents a specific challenge: securing 1,000+ GPUs in a single placement group for a multi-week training run. Cloud providers typically limit placement group sizes to 256–512 GPUs, meaning that a large training run must either negotiate a custom allocation, often requiring a substantial commitment, or accept the performance penalty of spanning multiple groups. The availability of large contiguous GPU allocations varies by region and time of day, and organizations can wait weeks for a sufficiently large allocation during periods of peak demand. This availability uncertainty is a hidden cost of the cloud path that does not appear in the per-GPU-hour pricing but can delay training timelines significantly.

Checkpoint 12.2: TCO decision framework

Your organization needs to train 10 models/year, each requiring 1,000 GPU-hours on H100s. You are evaluating whether to purchase an on-premises cluster with 8 GPUs or use cloud instances at $4/GPU-hour.

1. Calculate the annual cloud cost.
2. Calculate the annualized on-premises cost, assuming $350,000 per 8-GPU node, $0.07/kWh electricity at PUE 1.1, and 700 W per GPU.
3. Solve for the number of annual training runs at which on-premises becomes cheaper.

12.3.7 Infrastructure planning methodology

Infrastructure planning begins when a workload requirement becomes a facility requirement. A large training run does not merely ask for more GPUs: the model size fixes the memory pressure, the token budget fixes the compute budget, the desired calendar time fixes aggregate throughput, and the communication pattern fixes the fabric that can make the cluster useful. Once an accelerator is chosen, its TDP constrains cooling, rack density, pod layout, facility power, and ultimately total cost of ownership. Planning is therefore a causal chain, not a purchasing checklist.

The first step is workload characterization. Target model size determines the minimum accelerator count and memory strategy. Dataset size, target throughput, and acceptable training time turn the training objective into a FLOP-hour requirement. Communication pattern then changes the network answer: dense models stress AllReduce, mixture-of-experts models stress AllToAll, and pipeline-parallel models introduce more point-to-point traffic. Inference requirements add another constraint because the hardware that trains a model efficiently may not be the hardware that serves it economically.

With those constraints explicit, bottom-up sizing works from the accelerator outward. Roofline analysis distinguishes compute-bound training from memory-bound serving. Compute-bound training optimizes for sustained TFLOP/s per dollar, while memory-bound inference often optimizes for bandwidth per dollar and latency per watt.

At that point, planning becomes a sequence of coupled design constraints rather than a shopping list. Node sizing decides how many accelerators share a host and which memory tier carries optimizer state, activations, and data staging. For the 175B model, that points toward eight GPUs per node with tensor parallelism and roughly 2 TB of host DRAM for optimizer-state offload. Cluster sizing then divides the total compute budget by sustained per-node throughput after MFU losses and adds 5–10 percent for a maintenance pool and spares.

Network design follows the primary collective and validates expected scaling with equation 5.1. The fabric is also a cost line, not just a performance one: section 3.5 derives the leaf and spine switch counts a fat-tree requires, but each InfiniBand NDR switch costs $15,000–30,000 and active optical cables add $500–1,000 per link, so the network for a 1,024-GPU cluster runs $5–15 million depending on oversubscription and optical mix. A RoCE (RDMA over Converged Ethernet) fabric lowers per-port cost but is lossy under the bursty, synchronized traffic of distributed training: even a 1 percent packet-retransmission rate can cut effective AllReduce throughput by 10–20 percent because every GPU waits for the slowest participant. When the GPU fleet is a $35+ million investment, the InfiniBand premium can pay for itself by keeping the network from becoming the bottleneck.

Power and cooling translate IT load through PUE, then sanity-check rack density locally: a four-node DGX H100 rack is roughly a 30–40 kW design point once GPUs, host systems, networking, power conversion, and cooling overhead are included. Higher density changes the cooling architecture, not just the electric bill. TCO and schedule remain design variables because the cluster must arrive in time to matter, and GPU supply or electrical work often dominates lead time.

The same chain becomes concrete for the 175B training plan. Large-batch training is compute bound, so the design optimizes for sustained TFLOP/s per dollar and selects H100s rather than a bandwidth-optimized inference accelerator. The 2.2 TB training state cannot live on one device, which forces tensor parallelism inside each node plus optimizer sharding, activation checkpointing, and offload across the wider data-parallel group. That memory constraint produces the DGX H100 node shape before the facility planner ever reaches the power spreadsheet.

Cluster size then follows from the compute budget. At 6 FLOPs per parameter-token over 175 billion parameters and 300 billion tokens, the run requires 3.15×10^{23} FLOPs. Using a conservative BF16/FP16 planning basis rather than the FP8 headline peak, each H100 delivers 989 TFLOP/s peak and 445.1 TFLOP/s sustained at 45 percent MFU. With 1,024 GPUs, the cluster reaches roughly 455.7 PFLOP/s sustained and completes the run in about 8 days of idealized compute time, or 2–4 weeks with operational overhead. The TP-8, PP-4, DP-32 configuration generates structured AllReduce traffic suited to a rail-optimized InfiniBand fabric.

The 128 nodes draw 33.5 kW per 4-node rack across 32 racks at approximately 1.1 MW of facility-relevant load, necessitating liquid cooling. The 3-year TCO comes to approximately $63M on-premises; renting the six-runs-per-year workload in the cloud is roughly $24–48M depending on whether each run lasts 2 or 4 weeks, while continuous cloud use is more expensive. The break-even depends on sustained utilization beyond the initial training run. GPU procurement (6–12 months) and facility preparation must begin immediately, with phased deployment targeting initial capacity within 3 months.

12.3.7.1 Site selection and physical constraints

The planning chain has so far assumed a facility exists, but for a fleet at this scale the site itself is a planning variable that fixes several constraints before any accelerator is purchased. Power availability is the first and usually most binding. A 10,000-GPU pod requires 10–15 MW of continuous power, equivalent to a medium-sized factory, and at megawatt scale the gap between a favorable and an unfavorable electricity rate can amount to tens of millions of dollars over a hardware lifecycle. Locations near hydroelectric dams (the Pacific Northwest, Scandinavia, Quebec) offer abundant low-carbon power but are often remote from talent pools and existing fiber, so the trade-off between cheap power and proximity to engineers becomes one of the most consequential siting decisions.

Cooling environment is the second constraint. Sites in temperate or cold climates achieve lower PUE through **free cooling** (using outside air directly, without mechanical refrigeration) for much of the year; Meta's Lulea, Sweden facility, where the average annual temperature is 1 degree Celsius, approaches a PUE of 1.03 on year-round free cooling. Hot, arid regions face both higher cooling costs and water scarcity for evaporative towers. A 10 MW facility using standard evaporative cooling can consume well over 100 million liters of water annually, which in water-stressed regions places a data center in direct competition with municipal and agricultural demand and can trigger permitting limits; closed-loop liquid or dry-cooler systems cut water use to near zero but raise capital cost and, in hot climates, PUE. Network connectivity is the third constraint: training clusters that ingest geographically distributed data need WAN bandwidth, while serving clusters need proximity to internet exchange points for end-user latency, which is why some organizations separate training pods in power-rich remote sites from serving pods near population centers.

These constraints are frequently in tension, and regulatory boundaries can override all of them. Export controls act as a hard filter on which accelerator classes a given geography may host, and data-sovereignty mandates such as the European Union's General Data Protection Regulation (GDPR) can compel facilities into expensive, power-constrained regions to satisfy compliance rather than efficiency. The result is a **power alley** phenomenon that concentrates capacity in a few zones: Northern Virginia for connectivity, the Pacific Northwest for cheap hydroelectric power, the Nordics for free cooling. As those zones saturate, local grids face multi-year waits for new substation capacity, pushing operators toward unconventional "brownfield" sites (retired aluminum smelters, defunct coal plants) where high-voltage transmission exists but fiber and cooling must be built from scratch.

12.3.7.2 Construction and deployment timelines

Site choice feeds directly into schedule, and the timeline from decision to first computation is the planning constraint most organizations underestimate. Building a facility from scratch takes 18–30 months, with the electrical substation (18–24 months) and building shell (12–18 months) the longest-lead items; GPU lead times during high demand reach 6–12 months and run concurrently. The practical consequence is that infrastructure decisions must be made 18–30 months before the infrastructure is needed, often selecting a facility design for hardware that does not yet exist. To absorb that mismatch, experienced teams design for **headroom**: oversizing electrical infrastructure by 30–50 percent, designing the cooling plant for higher heat densities than the initial deployment requires, and specifying flexible rack layouts, at a typical premium of 10–20 percent of facility CapEx that buys the ability to upgrade compute without reconstructing the building.

The construction timeline is governed by a **critical path** that must align two disparate workstreams: the facility track (site, power, shell, cooling) and the hardware track (silicon allocation, server assembly, network integration). GPU procurement captures headlines, but the true bottleneck is frequently the electrical substation, whose permitting and transformer delivery cannot be compressed by spending more. This creates a high-stakes synchronization problem: a 20,000-GPU allocation ($500M+) secured before the facility is ready depreciates in a warehouse, while a $200M shell completed before silicon arrives leaves capital assets idle. To hedge it, teams use **phased deployment**, commissioning the facility in waves rather than targeting a single go-live date, often bringing up an initial 10–20 percent of capacity (2,000 of 10,000 GPUs) while construction continues. The early phase lets the software team validate the distributed-training stack and tune collective-communication kernels on the real topology while simultaneously stress-testing power distribution and cooling, surfacing hotspots and cabling defects at small scale.

What justifies these contortions is the **time value of compute**. For large-model development, the cost of delay can exceed the interest on capital, because delayed compute postpones experiments, product launches, and downstream capabilities. A model projected to generate $10 million per month in value loses $30 million to a three-month construction delay, often exceeding the cost of the facility's electrical infrastructure entirely. That calculus can justify paying premiums for prefabricated modular data centers or leasing temporary colocation space to bridge the gap between silicon delivery and facility readiness, and it is why capacity planning is a reliability and product-velocity control, not only a procurement function.

> Checkpoint 12.3: Infrastructure planning exercise
>
> Your team needs to train a 70B-parameter model on 1T tokens within 4 weeks. Using the following specifications:
>
> - H100 GPU: 1979 TFLOP/s FP8 peak, assume 45 percent MFU
> - Compute budget: $6 \times 70 \times 10^9 \times 10^{12} = 4.2 \times 10^{23}$ FLOPs
> - Available power: 2 MW
>
> 1. Determine the minimum number of GPUs needed.
> 2. Evaluate sufficiency of the 2 MW power budget, assuming PUE 1.1, 700 W per GPU, and 50 percent overhead for non-GPU components.
> 3. Estimate the training-run cost at $4/GPU-hour in the cloud and $350,000 per 8-GPU node on premises.

While the economic justification for platform operations becomes clear at scale, the technical implementation begins with a deceptively difficult problem: preventing independent models from colliding. Multi-model management requires untangling the hidden dependencies that emerge when hundreds of models share the same data, infrastructure, and user experiences.

12.4 Multi-Model Management

Imagine an e-commerce platform where the search ranking model uses outputs from a user embedding model. If the embedding team silently pushes an updated model with a different dimensionality

or scale, the search model will immediately begin producing garbage predictions. The maturity progression from single to multi-model operations hinges on managing precisely this kind of invisible entanglement.

Managing multiple machine learning models in production introduces coordination challenges absent from single-model operations. When models share features, feed predictions into one another, or compete for shared infrastructure resources, their individual behaviors become interdependent. Effective portfolio management therefore starts with registries, dependency tracking, and ensemble-aware deployment practices that make those relationships explicit.

12.4.1 Model registries at scale

Effective multi-model management begins by turning artifacts into governed interfaces. A model registry serves as the central catalog for all machine learning artifacts in an organization, but at enterprise scale the catalog must also expose the dependency graph that determines which downstream systems an update can break.

12.4.1.1 Core registry requirements

The registry can enforce dependency control only if four ordinary catalog capabilities are reliable. Table 12.11 summarizes the minimum interface: version identifiers let downstream consumers pin the exact artifact they validated; metadata exposes the training and deployment context needed for review; artifact storage turns registry entries into durable deployable assets; and access control keeps ownership boundaries explicit.

Table 12.11: Core Registry Requirements: Enterprise model registries need versioning, metadata, artifact storage, and access control before dependency-aware validation can be reliable. These catalog capabilities make model artifacts behave like governed production interfaces rather than files on shared storage.

Requirement	What it stores or enforces	Why it matters at scale
Version management	Unique artifact identifiers plus lineage for training data, hyperparameters, code commits, and evaluation metrics.	Downstream consumers can pin the exact artifact they validated.
Metadata storage	Training configuration, evaluation results, hardware requirements, serving configuration, and ownership.	Deployment and review decisions can be made without reverse-engineering the artifact.
Artifact storage	Durable model binaries with efficient retrieval and caching near serving locations.	Large models such as LLMs can exceed 100 GB and cannot rely on ad-hoc file distribution.
Access control	Read-write, administrative, and read-only permissions at team and dependency boundaries.	Model developers, platform operators, and dependent teams interact through governed APIs.

12.4.1.2 Dependency tracking

Beyond these core requirements, the distinguishing feature of enterprise registries is explicit dependency tracking. Figure 12.4 illustrates how updates to an upstream model (such as a user embedding model) automatically trigger alerts and validation for all downstream consumers, including ranking and retrieval models. When Model A consumes features computed by Model B, this relationship must be recorded and enforced.

The key takeaway from figure 12.4 is that a single upstream update can silently invalidate every downstream consumer unless the registry enforces explicit dependency edges. The necessity of this tracking becomes clear when considering a recommendation system with four interdependent models:

- Embedding Model E produces user and item embeddings
- Retrieval Model R uses embeddings from E to generate candidates
- Ranking Models R1, R2, R3 score candidates using embeddings from E
- Ensemble Model M combines outputs from R1, R2, R3

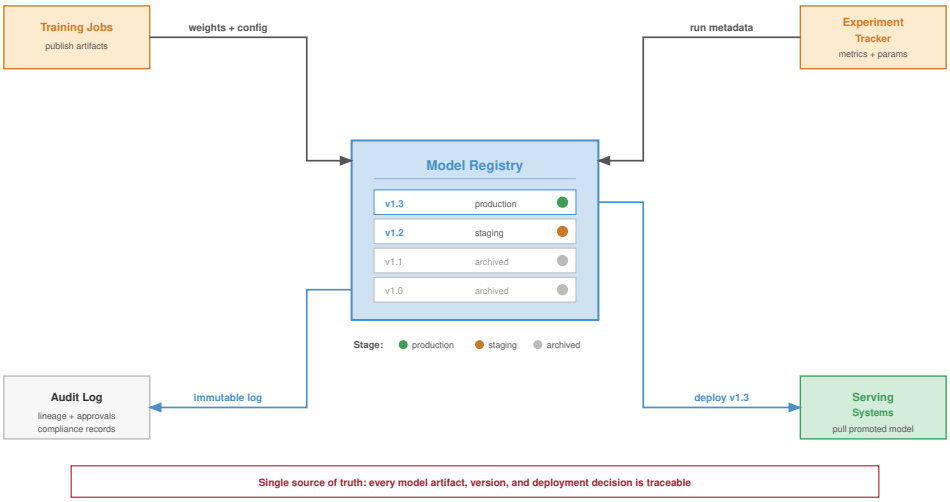


Figure 12.4: Dependency-Aware Model Registry: Diagram showing a registry that tracks not just *artifacts* but the *graph* of dependencies between models. An update to the "User Embedding Model" triggers alerts or automated retraining for dependent "Ranking" and "Retrieval" models, preventing silent downstream failures.

The dependency graph must be explicit in the registry because the update approval process is graph traversal. When Embedding Model E is updated, the registry executes four graph-traversal steps:

1. Identify all dependent models (R, R1, R2, R3, M)
2. Trigger re-evaluation of dependent models with new embeddings
3. Block deployment of the new E until compatibility is verified
4. Coordinate deployment order if updates proceed

Without explicit dependency tracking, organizations discover dependencies through production failures when an upstream model change breaks downstream consumers.

Listing 12.1 illustrates how a single YAML entry captures artifact location, training provenance, and evaluation thresholds, giving the dependency tracker enough metadata to identify which downstream models must be re-evaluated when the upstream embedding changes.

Listing 12.1: Model Registry Schema: A YAML entry capturing model metadata, artifact location, training provenance, and evaluation results for dependency tracking and reproducibility.

```yaml
model:
  name: user_embedding_v3
  version: "3.2.1"
  type: embedding_model
  domain: recommendation

artifact:
  path: gs://models/user_embedding_v3/3.2.1/
  format: tensorflow_savedmodel
  size_bytes: 4294967296

training:
  data_version: user_interaction_2024_01
  code_commit: abc123def
  started_at: 2024-01-15T10:00:00Z
  duration_hours: 48
  hardware: 8xA100-80 GB

evaluation:
  metrics:
    recall_at_100: 0.342
    embedding_quality: 0.891
  evaluation_set: eval_2024_01

dependencies:
  upstream:
    - feature_store/user_features_v2
    - feature_store/interaction_features_v1
  downstream:
    - models/candidate_retrieval_v4
    - models/ranking_ensemble_v2

serving:
  min_replicas: 10
  max_replicas: 100
  latency_p99_target_ms: 5
  memory_gb: 16

ownership:
  team: recommendation-core
  oncall: recsys-oncall-team
```

12.4.2 Ensemble management

Recommendation systems exemplify the multi-model management challenge because they operate as ensembles of 10–50 models per request. The platform must manage the ensemble as one production interface even when different teams own its components.

12.4.2.1 Why ensembles dominate recommendation

Modern recommendation systems use ensemble architectures because one model cannot simultaneously satisfy every latency, quality, and business constraint. Diverse objectives demand specialized models: engagement, diversity, freshness, inventory, and business constraints often pull in different directions, so separate models specialize in each objective and the ensemble combines their outputs. Staged filtering is the systems counterpart to that specialization. Processing enormous candidate sets with a single expensive model is computationally infeasible, so multi-stage architectures progressively filter candidates: retrieval produces a manageable candidate set, ranking scores that set with richer features, and later re-ranking or business-rule stages produce the final ordering (Covington et al. 2016; Liu et al. 2022).

The same decomposition improves experimentation velocity because teams can update individual components without retraining the entire recommendation stack, but that independence creates

version-compatibility obligations for the platform. It also changes risk management. Other ensemble components can sometimes compensate when one model fails or produces poor results, yet the platform must detect when compensation hides a regression rather than resolving it.

12.4.2.2 Ensemble deployment patterns

Deploying ensemble updates requires coordination that single-model deployments do not. Consider updating the fine ranking model within a recommendation ensemble. Table 12.12 breaks down the staged ensemble deployment pattern into four phases: shadow deployment (24–48 hours logging without serving), canary (1 percent traffic for 4–8 hours), staged rollout (5 percent to 100 percent over 24–72 hours), and soak (7–14 days monitoring for delayed effects).

Table 12.12: Staged Ensemble Deployment Pattern: Four-phase rollout for updating recommendation ensemble components. Shadow deployment (24–48 hours) validates behavior without user impact, canary (1 percent traffic, 4–8 hours) enables statistical regression detection, staged rollout progressively increases exposure, and soak period (7–14 days) catches delayed interaction effects.

Deployment Stage	Actions	Duration	Rollback Trigger
Shadow	New model scores alongside production, results logged but not served	24–48 hours	Quality metrics below threshold
Canary	1% traffic receives new model results	4–8 hours	Statistical significance of regression
Staged Rollout	5% → 25% → 50% → 100%	24–72 hours	Business metric degradation
Soak	Full traffic, extended monitoring	7–14 days	Delayed effects emerge

The extended timeline reflects the difficulty of detecting regressions in ensemble systems. A component change that improves its local metrics might degrade system-level performance through subtle interactions with other components.

12.4.2.3 Interaction effects

Ensemble components interact in ways that make local validation insufficient; three recurring patterns explain why the platform needs holdouts, intermediate logging, and long soak periods:

- **Compensation effects**: The retrieval model starts returning lower-quality candidates and the ranking model learns to compensate by upweighting quality signals; once retrieval is fixed, ranking can over-compensate and degrade results.
- **Distribution shift propagation**: An upstream model improves locally but changes the input distribution seen by downstream models trained on the old representation.
- **Feedback loops**: Ranking decisions affect which items users interact with, and those interactions become training data for future models over days to weeks.

Managing these interactions requires holdout groups that experience no changes and provide stable baselines, extensive logging of intermediate model outputs beyond final recommendations, long-term monitoring for feedback loop effects, and periodic ensemble reset experiments that retrain all components together.

12.4.3 Model lifecycle management

Lifecycle management exists to keep model dependencies from becoming permanent obligations: every promotion widens a model's blast radius, and every retirement must migrate its consumers before the platform can reclaim resources. The stages below trace that progression from development to archive, each defined by the gates it imposes rather than by a status label.

```
Development → Staging → Canary → Production → Deprecation → Archive
```

In development, models exist as experimental artifacts. Operations requirements are intentionally light, but they are not optional: successful experiments must preserve results, version history,

and enough reproducibility metadata to be challenged later. The operational concern is therefore transition readiness. A promising model should not move to staging until it has clear production-readiness criteria, automated evaluation against production-equivalent data, and documentation sufficient for review.

Staging turns that candidate into a deployable service without exposing users. A staged model should process production traffic in shadow mode, run against production feature pipelines, execute on production-equivalent hardware, and meet latency and throughput requirements. The promotion gate then combines automated checks, such as metric thresholds and latency requirements, with human review of model behavior and risk.

Production widens the blast radius from validation traffic to live traffic, so the model now requires continuous monitoring with alerting, capacity for traffic fluctuations, rollback procedures, and on-call support. Production is not a terminal state. Models still need retraining as data distributions shift, feature pipeline updates as upstream data changes, infrastructure updates as serving systems evolve, and periodic re-evaluation against newer baselines.

The lifecycle closes only when a model can be retired without breaking its consumers. Deprecation identifies dependent systems, provides migration paths and timelines, maintains the old model until migration completes, and archives artifacts for reproducibility and audit. Organizations often underinvest in this final stage, leading to accumulation of zombie models[3] that consume resources but provide questionable value. Platform-level lifecycle enforcement helps address this pattern.

3 **Zombie Models**: Production models that continue serving despite being obsolete or superseded. Industry surveys estimate 20–40 percent of production models at mature organizations qualify as zombies, each consuming GPU memory, on-call attention, and monitoring budget while delivering negligible business value. The operational cost extends beyond wasted compute: zombie models inflate the dependency graph, making platform-wide upgrades and security patches slower and riskier.

12.4.4 Deployment patterns by model count

The appropriate deployment pattern depends on the number and interdependence of models being updated because each additional dependency widens the rollback unit. Table 12.13 categorizes deployment patterns by scale: single model deployments (monthly updates for isolated vision classifiers), pipeline deployments (weekly updates for 3–5 sequential NLP models), ensemble deployments (daily updates for 10–50 recommendation components), and platform deployments (continuous updates across hundreds of enterprise models).

Table 12.13: Deployment Patterns by Scale: Four patterns addressing different model counts and update frequencies. Single model deployments (1 model, monthly updates) use standard canary rollouts, while platform deployments (100+ models, continuous updates) require automated policy enforcement, cross-model impact analysis, and global rate limiting to prevent simultaneous high-risk deployments.

Pattern	Model Count	Update Frequency	Example
Single Model	1	Monthly	Vision classifier
Pipeline	3-5	Weekly	NLP processing pipeline
Ensemble	10-50	Daily	Recommendation system
Platform	100s	Continuous	Enterprise ML platform

For isolated models with no dependencies, standard deployment patterns suffice. Canary deployments, blue-green switches[4], and gradual rollouts all work effectively because rollback means returning one model artifact to its previous version.

Pipelines introduce ordering constraints because models execute in sequence and each model's output feeds the next. The deployment rule is compatibility before exposure: deploy upstream models before downstream consumers, validate each stage before proceeding, maintain version compatibility between stages, and roll back the pipeline as a unit if any stage fails.

Ensembles widen the coordination problem because multiple models may execute in parallel or in a graph. Different teams can update components on different schedules, partial updates may change only part of the ensemble, and the system behavior emerges from component interactions. Testing in isolation is therefore insufficient; integration testing becomes the deployment gate.

At platform scale, continuous deployment means some model is always being updated somewhere, so the platform must prevent individually safe changes from combining into an unsafe fleet state. Platform deployment requires automated rollout policies based on model risk classification, cross-model impact analysis before deployment approval, global rate limiting to prevent simultaneous high-risk deployments, and automated correlation of incidents with recent deployments.

4 **Blue-Green Deployment**: A release pattern that uses two identical production environments ("Blue" and "Green"). Only one environment is live at a time. This enables zero-downtime rollouts and near-instant rollback if the new model fails, at the cost of doubling the required serving infrastructure (C_{infer}).

12.4.5 Cross-model dependencies in practice

12.4.5.1 Example: E-commerce model ecosystem

The dependency-control argument becomes concrete in an e-commerce model graph, where a single embedding interface fans out into retrieval, ranking, and business logic. An e-commerce platform typically starts the graph with embedding models. A user embedding model generates user representations from behavior history, while a product embedding model represents products from attributes and interactions. Those embeddings are not final predictions; they are shared interfaces consumed by downstream services.

The next layer turns shared representations into candidate sets and business signals. A candidate retrieval model uses the embeddings to find relevant products, and a price sensitivity model estimates how strongly a user will respond to price changes. A ranking model then scores the candidate products using both embedding features and auxiliary signals, while a business rules model applies promotional and inventory constraints before results reach the user.

Figure 12.5 maps this dependency structure, showing how user and product embeddings flow through retrieval and price sensitivity models before converging in the ranking and business rules layers. This graph reveals critical operational implications.

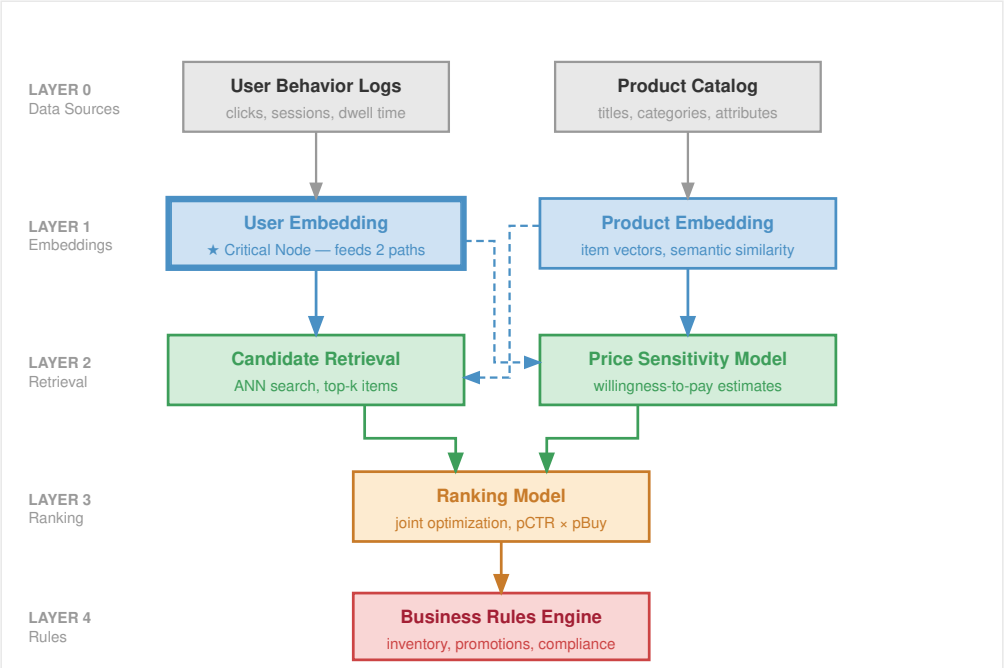


Figure 12.5: E-Commerce Model Ecosystem: A complex dependency graph where upstream models (Embeddings) feed into mid-tier models (Retrieval, Price Sensitivity) which feed into final ranking and logic layers. Changes to identifying "User Embedding" require coordinated updates to all downstream consumers.

As figure 12.5 makes visible, a single change to the User Embedding model propagates through two direct consumers and four transitive downstream nodes, including the final business rules layer. Operational procedures must therefore address four coordination requirements:

1. Re-evaluate all downstream models with new embeddings before deployment
2. Consider simultaneous deployment of related components
3. Monitor both direct metrics (embedding quality) and downstream metrics (ranking performance)
4. Maintain embedding version compatibility or coordinate synchronized updates

The example illustrates why multi-model management requires explicit dependency tracking and coordinated deployment procedures. Dependency graphs and registries, however, are static artifacts. Moving entangled models from a repository into a live production environment without causing cascading failures requires automated, verifiable deployment pipelines.

12.5 CI/CD for ML at Scale

The dependency graphs and ensemble architectures examined in multi-model management do not deploy themselves. Each model update must navigate the dependency web: an embedding model update might require re-evaluation of four transitive downstream components, including the business rules layer, before any component can safely reach production. This coordination challenge transforms CI/CD from a per-model concern into a platform orchestration problem. Where software CI/CD validates code in isolation, ML CI/CD at scale must validate models within their operational context, ensuring upstream changes do not break downstream consumers and that deployment order respects the dependency graph.

Chapter 5 detailed how data, tensor, and pipeline parallelism enable training models too large for single machines, producing artifacts that require validation and deployment at scale. Continuous integration and continuous deployment practices for machine learning differ from traditional software CI/CD along one axis: while software CI/CD focuses on code correctness and deployment reliability, ML CI/CD must additionally validate data, verify model performance, and manage the interactions between code, data, and learned parameters. At platform scale, these challenges multiply as pipelines must coordinate across hundreds of models with varying requirements.

12.5.1 Training pipeline automation

CI/CD for machine learning begins by making training a reproducible producer of deployable artifacts, not an artisanal job run. A well-designed training pipeline executes reproducibly, handles failures gracefully, and produces artifacts suitable for deployment validation.

12.5.1.1 Pipeline stages

A complete training pipeline includes data validation, training execution, model evaluation, registry registration, and canary deployment, each separated by quality gates that prevent defective artifacts from advancing (figure 12.6). The sequence matters because each stage protects a different fleet invariant.

1. Data validation pins the data version and rejects schema, freshness, or distribution failures before they consume training capacity.
2. Feature engineering preserves the feature contract between training and serving, so preprocessing changes do not become silent production regressions.
3. Training records code, data, hyperparameters, hardware envelope, and random seeds so a promising artifact can be reproduced or challenged later.
4. Evaluation compares the candidate with baselines, slice metrics, latency budgets, and task-specific gates before any user traffic is exposed.
5. Artifact generation packages the model with its serving configuration, dependency versions, and rollback target.
6. Registration records the artifact in the model registry with lineage, approvals, and deployment eligibility.

Each stage should be independently executable and idempotent. If the pipeline fails at evaluation, restarting should not re-execute data validation and feature engineering unless their inputs have changed.

12.5.1.2 Pipeline orchestration

Training pipelines require orchestration systems that make dependency order, resource allocation, and failure recovery explicit. The orchestrator represents the workflow as a directed acyclic graph (DAG), tracks dependencies between stages, retries transient failures without rerunning unaffected work, schedules scarce resources such as GPUs and memory, caches intermediate results when inputs are unchanged, and records logs and artifacts for later debugging.

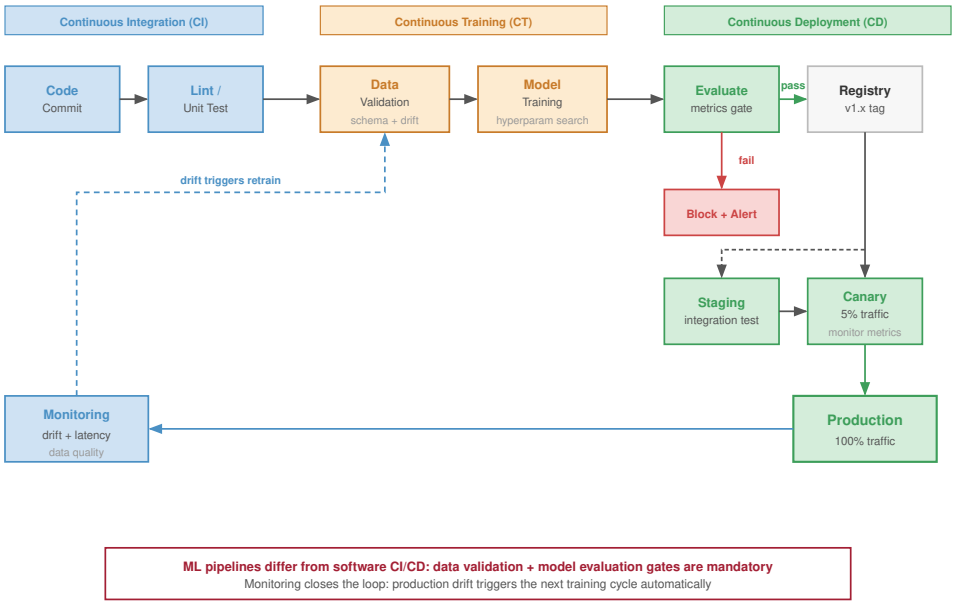


Figure 12.6: ML CI/CD Pipeline: The automated workflow transforming code and data into a deployed service. Stages include Data Validation (schema/drift checks), Training, Evaluation (metric gates), Artifact Registration, and Staged Deployment (canary rollout). Feedback loops automatically trigger retrains or alerts if gates fail.

Common orchestration choices include Kubeflow Pipelines[5] (Bisong 2019), Airflow with ML extensions, and cloud-native solutions like Vertex AI Pipelines or SageMaker Pipelines. The choice depends on which dependency and resource contracts the platform must enforce, as well as existing infrastructure, team expertise, and scale requirements. Orchestration fixes the execution contract; parameterization fixes the variation contract, so the same graph can run against different data, models, and hardware envelopes without becoming a new program each time.

12.5.1.3 Pipeline parameterization

Effective pipelines separate configuration from code, as illustrated in listing 12.2.

5 **Kubeflow**: An open-source ML platform released by Google in 2018 that couples pipeline orchestration to Kubernetes resource management. The key systems consequence: because Kubeflow DAGs express both computational dependencies and resource requests, the orchestrator can schedule GPU-intensive training steps only when accelerators are available, preventing the resource contention that makes manual scheduling of multi-model training fleets untenable.

Listing 12.2: Pipeline Parameterization: A YAML configuration separating data paths, feature sets, training hyperparameters, and evaluation criteria from pipeline code.

```yaml
training_pipeline:
  model_type: transformer_ranking
  data:
    train_path: gs://data/train/2024-01-*
    eval_path: gs://data/eval/2024-01-15
    schema_version: v3.2
  features:
    user_features: [embedding, history, demographics]
    item_features: [embedding, attributes, popularity]
  training:
    epochs: 10
    batch_size: 4096
    learning_rate: 0.001
    optimizer: adam
    hardware: 4xA100
  evaluation:
    metrics: [ndcg_10, mrr, coverage]
    baseline_model: ranking_v2.1.0
```

The separation enables configuration-driven training because the platform can vary data versions, hyperparameter sweeps, reproducibility records, and environment-specific resource overrides without rewriting the pipeline code. A parameter file is also an audit artifact. When a candidate model later reaches a release gate, the platform can reconstruct exactly which data slice, feature set, hardware envelope, and threshold configuration produced it. That lineage turns validation from a one-time judgment into a reproducible contract.

12.5.2 Validation gates

Validation gates determine whether a trained model should proceed toward production. Effective gates balance thoroughness against deployment velocity. A release gate is an evidence bundle rather than a single score: model performance, latency, policy and fairness constraints, data quality, and operational readiness each answer a different failure mode before the model can safely receive traffic.

12.5.2.1 Performance gates

Performance validation compares the candidate model against absolute thresholds where the model must exceed minimum acceptable performance, relative baselines where the model must match or exceed current production performance, and historical trends where the model should not regress from recent performance trajectory. Listing 12.3 demonstrates this multi-criteria evaluation.

Listing 12.3: Performance Gate Evaluation: A validation function that checks absolute thresholds, relative improvement over the production model, and regression bounds on secondary metrics.

```
def evaluate_performance_gate(
    candidate_metrics, production_metrics, thresholds
):
    """
    Evaluate whether candidate model passes performance gates.

    Returns tuple of (passed: bool, reasons: list)
    """
    reasons = []

    # Absolute threshold check
    if candidate_metrics["ndcg_10"] < thresholds["min_ndcg"]:
        reasons.append(
            f"NDCG_10 {candidate_metrics['ndcg_10']:.4f} below minimum {thresholds['min_ndcg']}"
        )

    # Relative improvement check
    relative_improvement = (
        candidate_metrics["ndcg_10"] - production_metrics["ndcg_10"]
    ) / production_metrics["ndcg_10"]
    if relative_improvement < thresholds["min_improvement"]:
        reasons.append(
            f"Improvement {relative_improvement:.2%} below minimum {thresholds['min_improvement']:.2%}"
        )

    # Regression check on secondary metrics
    for metric in ["mrr", "coverage"]:
        if candidate_metrics[metric] < production_metrics[metric] * (
            1 - thresholds["max_regression"]
        ):
            reasons.append(
                f"{metric} regression exceeds {thresholds['max_regression']:.2%} tolerance"
            )

    return (len(reasons) == 0, reasons)
```

12.5.2.2 Latency gates

Production models must meet latency requirements. Validation should measure inference latency on representative hardware, test at expected throughput levels, and account for batching effects if applicable. Table 12.14 specifies latency gate thresholds by model type: fraud detection demands the strictest requirements (5 ms p50, 20 ms p99 with instant blocking on violation), while LLMs accept broader bounds (500 ms p50, 2000 ms p99) reflecting their different operational constraints.

Table 12.14: Latency Gate Thresholds by Model Type: Production latency requirements (p50 and p99) and gate actions when thresholds are exceeded. Fraud detection enforces the strictest requirements (5 ms p50, 20 ms p99) with high-priority blocking, reflecting the real-time nature of transaction processing. LLMs accept broader bounds (500 ms p50, 2000 ms p99) while requiring optimization before deployment approval.

Model Type	p50 Target	p99 Target	Gate Action if Exceeded
LLM	500 ms	2000 ms	Block deployment, require optimization
Recommendation	10 ms	50 ms	Block deployment
Fraud Detection	5 ms	20 ms	Block deployment, high priority
Vision	50 ms	200 ms	Warning, conditional approval

Latency gates catch overt performance violations, but some regressions pass every gate and still degrade the product. The most dangerous failures are semantically silent: valid outputs built on corrupted inputs.

Example 12.2: Silent model regression

Scenario: A product ranking model passes offline validation gates but causes a measurable business regression after deployment.

Failure mode: The culprit is a silent failure in an upstream feature pipeline. A schema change causes a key behavioral feature to return `null` for a small slice of users. The model serving infrastructure, designed for robustness, automatically imputes these nulls as `0.0`.

Consequence: Since `0.0` is a valid value in the feature space, no errors are logged. The model simply makes worse predictions for those users until a business metric monitor catches the regression.

Systems insight: Semantic silence is more dangerous than loud failure: syntactically valid feature values can hide production regressions until business metrics move.

12.5.2.3 Fairness gates

Fairness concepts become operational at release time when policy choices are encoded as thresholds. For models affecting users, fairness validation[6] turns a policy commitment into a release gate. The platform cannot simply ask whether "the model is fair" because different contexts encode fairness differently. In operations, those definitions first appear as executable thresholds. Equation 12.3 expresses one operational choice: the probability of a positive prediction must differ by less than threshold ϵ between protected groups a and b. Equation 12.4 encodes a stricter choice from the equalized-odds family: prediction behavior must be similar across groups after conditioning on the true outcome (Hardt et al. 2016).

$$\text{Demographic Parity: } |\Pr(\hat{Y} = 1 \mid A = a) - \Pr(\hat{Y} = 1 \mid A = b)| < \epsilon \tag{12.3}$$

$$\text{Equalized Odds: } |\Pr(\hat{Y} = 1 \mid Y = y, A = a) - \Pr(\hat{Y} = 1 \mid Y = y, A = b)| < \epsilon \tag{12.4}$$

where A represents the protected attribute, $\hat{Y}$ is the model prediction, and Y is the true outcome. The operational point is not that every gate should enforce every definition. The gate must record which definition the organization chose, compare the measured value against historical baselines as well as absolute thresholds, and route borderline cases to human review because a near-threshold fairness result is also a governance decision.

6 **Fairness Validation in ML**: Multiple fairness definitions exist – demographic parity, equalized odds, calibration – and satisfying them simultaneously is generally impossible under realistic conditions (Chouldechova 2017; Kleinberg et al. 2016). This forces a systems design choice: automated validation gates must encode which fairness definition the organization prioritizes, making the gate configuration itself a policy decision that cannot be delegated to a generic threshold.

12.5.2.4 Data quality gates

Before training or deployment, data quality validation ensures that data meets expected properties (Caveness et al. 2020). Schema conformance verifies all required fields are present with correct types. Statistical properties ensure feature distributions remain within expected bounds. Freshness checks confirm data is not stale beyond acceptable thresholds. Completeness verification ensures missing data rates stay within tolerance.

These gates catch issues that would otherwise manifest as mysterious model degradation, completing the pre-release evidence bundle before deployment risk shifts from artifact validity to live-traffic exposure. The next control is no longer another offline score; it is the amount of production traffic allowed to test the artifact under real conditions.

12.5.3 Staged rollout strategies

Staged rollout is the mechanism that turns deployment risk into a controllable exposure budget. Traffic increases only after the model earns each larger blast radius through continued acceptable performance.

Napkin Math 12.6: The safety of staged rollouts

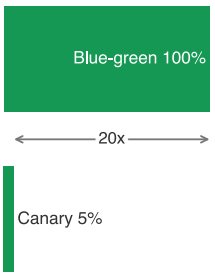
Problem: A team is deploying a new ranking model. A "Blue-Green" deployment (100 percent cutover) exposes all users to any potential bugs. A "Canary" deployment starts at 5 percent traffic. By how much does the canary approach reduce the deployment's "Risk Exposure"?

Math: Risk exposure is proportional to the traffic percentage affected during the detection window.

1. **Blue-Green Exposure**: 100 percent of users affected until rollback.
2. **Canary Exposure**: 5 percent of users affected until rollback.
3. **Risk Mitigation**: $100/5 = 20\times$.

Systems insight: Staged rollouts are an insurance policy for model quality. Limiting initial exposure to 5 percent reduces the blast radius of a catastrophic failure by 20×. In the Machine Learning Fleet, where model behavior is probabilistic and hard to unit-test, gradual exposure is the only reliable way to ensure that "SOTA on paper" does not become "Broken in Production."

12.5.3.1 Blue-green deployment



Canary rollout cuts initial exposure by 20x.

Blue-green deployment maintains two identical production environments. The current version (blue) serves traffic while the new version (green) is prepared. Once ready, traffic switches instantaneously to green.

The infrastructure cost of this duplication is qualitatively different for ML systems than for stateless web services. A recommendation or language model that requires multiple GPUs to hold its weights in VRAM cannot share that memory across environments: blue-green deployment for a large model means holding two complete copies of model weights across parallel accelerator allocations simultaneously. Loading model weights into GPU memory also takes measurable time (tens of seconds to minutes for multi-gigabyte checkpoints), so the "instant rollback" benefit assumes those weights remain warm in the standby environment throughout the deployment window. The effective cost is therefore not merely doubled serving infrastructure but doubled GPU memory reservation for the duration of the transition. Many organizations running large models treat blue-green deployment as a theoretical option and rely on canary or shadow deployment in practice because hardware capacity and cost make the duplicate allocation impractical.

Blue-green deployment offers a simple mental model and full testing in a production-equivalent environment before exposure. It also requires duplicate GPU and memory allocation during transition, provides no gradual exposure to detect subtle quality regressions, and relies on a binary switch that may miss issues emerging only at scale. The pattern fits low-risk changes to lightweight models where gradual rollout provides limited additional safety and the duplicate capacity cost is acceptable.

12.5.3.2 Canary deployment

Canary deployment[7] routes a small percentage of traffic to the new version while monitoring for regressions. If metrics remain acceptable, traffic percentage increases until the new version serves all traffic.

A typical progression moves from 1 percent to 5 percent to 25 percent to 50 percent to 100 percent as evidence accumulates. Determining the duration of each stage is critical. Equation 12.5 relates stage duration to sample requirements, request rate, and traffic percentage, enabling precise calculation of minimum canary durations for statistical validity:

$$T_{\text{stage}} = \frac{n_{\text{samples needed}}}{r_{\text{requests}} \times p_{\text{stage}}} \tag{12.5}$$

where T_{stage} is the duration required at a given percentage, $n_{\text{samples needed}}$ is the number of observations needed for statistical significance, r_{requests} is the request rate, and p_{stage} is the traffic percentage.

7 **Canary Deployment**: Named after the coal-mining practice of using canaries to detect toxic gases – a small sacrifice that protects the whole. For ML systems, the analogy is precise: model regressions typically manifest as gradual accuracy degradation rather than crashes, making them invisible to health checks but detectable through statistical comparison of canary vs. control traffic. Without canary stages, a subtly degraded model can reach 100 percent traffic before anyone notices the loss.

12.5.3.3 Worked example: Canary duration calculation

A model serves 1 million requests per hour. To detect a 1 percent change in click-through rate with 95 percent confidence requires approximately 10,000 samples per variant.

At 1 percent canary traffic: $T_{1\%} = \frac{10{,}000}{1{,}000{,}000 \times 0.01} = 1$ hour. At 5 percent canary traffic: $T_{5\%} = \frac{10{,}000}{1{,}000{,}000 \times 0.05} = 0.2$ hours $= 12$ minutes.

The organization might configure five rollout stages:

- 1 percent for 2 hours (2× minimum for buffer)
- 5 percent for 30 minutes
- 25 percent for 30 minutes
- 50 percent for 1 hour
- 100 percent deployment

Total rollout: approximately 4 hours for a confident deployment.

> Napkin Math 12.7: The cost of a delayed alert
>
> **Problem**: A team deploys a recommendation model that has a silent bug: it reduces Click-Through Rate (CTR) by 0.5 percentage points (10 percent relative), from 5 percent to 4.5 percent. If the service handles 5,000 requests/s and each click is worth \$0.50/click, how much revenue is lost if detection and remediation take 24 hours?
>
> **Math**:
>
> 1. **Total Requests**: 5,000 requests/s × 86,400 s/day = 432,000,000 requests.
> 2. **Lost Clicks**: 432,000,000 requests × 0.005 = 2,160,000 lost clicks.
> 3. **Total Financial Loss**: 2,160,000 lost clicks × \$0.50/click = \$1,080,000.
>
> **Systems insight**: A "minor" 0.5 percentage-point regression in a high-traffic model is a \$1,080,000 daily disaster. MLOps is fundamentally economic risk management. Every hour shaved from "Time-to-Detection" via canary deployments and automated drift monitoring translates directly into saved revenue. In the fleet stack, monitoring is the "Insurance Policy" that protects model business value.

12.5.4 Multi-region deployment coordination

Chapter 7 developed checkpointing, elastic training, and recovery mechanisms for handling failures within distributed training jobs. Multi-region deployment uses the same fault-tolerance idea [preserve useful work and state across failures] in the inference plane, where coordination across geographic regions introduces challenges absent from single-region operations. Model version consistency, traffic routing during transitions, and coordinated rollback require explicit protocol design to prevent mixed-version serving that can corrupt A/B test validity and user experience.

12.5.4.1 Coordination challenges

Multi-region ML deployment introduces two coordination burdens absent from stateless service rollouts. First, model artifacts are large: a production recommendation system's ensemble of sparse and dense models may require tens to hundreds of gigabytes of checkpoint data, and a large language model serving inference may require hundreds of gigabytes per replica. Distributing those artifacts across wide-area networks before a region can serve traffic is a bandwidth-constrained scheduling problem, not a metadata-only configuration push. Second, most ML models do not carry their context in the request; they pull it from a feature store. If Region B has received the new model binary but its regional feature store has not yet synced the latest user embeddings, that region will serve degraded predictions despite a technically successful deployment. The readiness check for an ML region must therefore verify the model binary, feature-store consistency, and serving health as one coherent state.

Mixed-version serving is the failure mode that binds multi-region deployment. A rollout protocol must preserve a coherent version boundary even when clocks, traffic volume, routing, and rollback behavior differ by region.

Clock skew and timing coordination create ambiguity about canary phase boundaries. When a deployment starts at 2:00 PM UTC, Region A may begin its 1 percent canary while Region B, due to network delays or operational variation, still serves the old version. Defining deployment phases using wall-clock time leads to inconsistent user experiences as users crossing region boundaries encounter different model versions, so production systems need logical rollout states in addition to timestamps.

Regional traffic variation turns the same percentage into different statistical evidence. A 1 percent global canary might represent 5 percent of traffic in a low-volume region, enough for statistical significance, but only 0.3 percent in a high-volume region, potentially insufficient. Per-region sample sizes must be validated independently before the platform treats a canary as evidence.

Cross-region request routing then turns regional skew into user-visible inconsistency. Users may be routed to different regions based on latency, load balancing, or failover. A user whose requests span multiple regions during a deployment window may receive predictions from different model versions, violating the consistency assumptions underlying A/B test analysis.

Rollback is the final consistency test. Rolling back one region while others continue serving the new version creates the same mixed-version problem that careful deployment coordination was meant to prevent, so rollback must be part of the rollout protocol rather than an ad hoc emergency action.

12.5.4.2 Deployment strategies

Multi-region coordination chooses where to spend the rollout tax: longer deployment duration, stronger synchronization, or bounded regional skew. Three architectural approaches make different trade-offs:

- **Sequential regional rollout**: Regions deploy one at a time, completing the full canary progression in each region before the next begins. This maximizes safety by limiting blast radius to a single region, but extends total deployment duration proportionally to region count.
- **Synchronized global rollout**: All regions maintain identical deployment state simultaneously. A global coordination service transitions regions between canary phases at the same logical timestamp, giving users a consistent experience but making every regional issue part of the global deployment decision.
- **Hybrid rollout**: A deployment coordinator enforces minimum and maximum phase boundaries while allowing regions to progress independently within those bounds. Regions can accelerate through phases if local metrics are strong or pause if issues emerge, while global constraints prevent excessive version skew.

The right choice depends on whether the deployment values minimum blast radius, global consistency, or bounded regional independence most.

For a conservative sequential rollout across 5 regions, the typical progression is:

1. Canary region (lowest traffic): Full canary cycle, 24 to 48 hours
2. Early adopter regions (2 regions, 20 percent global traffic): Parallel deployment, 24 to 48 hours
3. Majority regions (2 regions, 70 percent global traffic): Parallel deployment, 24 to 48 hours
4. Final validation: Cross-region consistency check, 12 to 24 hours

Total deployment duration: 4 to 8 days for a conservative rollout.

12.5.4.3 Traffic management during transitions

Maintaining request consistency during deployment transitions requires three routing controls that keep the experiment and user experience coherent:

- **Sticky routing**: A user's requests consistently route to the same region throughout the deployment window, typically through consistent hashing on user identifier. Users experience either the old version or new version consistently, never mixing within a session.

- **Version pinning**: Clients include a model version hash in the request, and the serving infrastructure routes to replicas serving that version. This supports gradual client migration independent of server-side deployment state.
- **Request isolation**: Cross-region traffic is disabled during critical deployment phases. During canary evaluation, this ensures metrics reflect single-region behavior rather than mixed routing patterns.

These controls preserve the statistical meaning of deployment metrics while users move through a changing fleet.

12.5.4.4 Consistency models for deployment

The choice of consistency model affects both deployment complexity and validity of deployment metrics. Table 12.15 compares deployment consistency models for multi-region systems: strong consistency guarantees identical versions across regions (essential for financial predictions) but requires high coordination overhead, while eventual consistency allows independent progression suitable for content recommendations at the cost of temporary version divergence:

Table 12.15: Consistency Models for Multi-Region Deployment: Three consistency guarantees with their use cases and coordination overhead. Strong consistency (all regions serve identical versions) is essential for financial predictions but requires high synchronization overhead. Eventual consistency enables independent regional progression suitable for content recommendations but may produce temporary version divergence.

Model	Guarantee	Use Case	Coordination Overhead
Strong	All regions serve identical version	Financial predictions, safety-critical	High (global synchronization)
Eventual	Regions converge to same version	Content recommendations	Low (independent progression)
Bounded staleness	Regions within k versions of each other	Real-time ranking	Medium (version monitoring)

For A/B testing validity, model serving typically requires strong consistency within treatment groups. If some users assigned to treatment receive old-version predictions due to deployment timing, the measured treatment effect is diluted. Eventual consistency across treatment groups is acceptable since each group is analyzed independently.

12.5.4.5 Rollback coordination

Rolling back a multi-region deployment requires careful coordination to prevent oscillation and mixed-version serving. The rollback protocol has two ordered phases:

1. **Stop traffic to the new version globally**: The deployment coordinator broadcasts rollback intent, waits for every region to acknowledge that it has stopped routing new traffic to the new version, and lets in-flight requests complete. Regions that do not acknowledge before the timeout are marked unhealthy.
2. **Restore the old version globally**: The coordinator confirms that all regions serve only the old version, re-enables normal traffic routing, clears deployment state, and prepares the system for a later re-attempt.

The protocol ensures that at no point do some regions serve the new version while others have rolled back, which would create inconsistent user experiences.

Partial rollback allows rolling back individual regions while others continue. This is appropriate when issues are region-specific (infrastructure problems, regional traffic patterns) rather than model-inherent. The deployment coordinator tracks per-region state and prevents inconsistent global decisions based on partial information.

12.5.4.6 Worked example: Multi-region coordination overhead

A recommendation system deploys across 5 regions with average inter-region latency of 80 ms. The coordination protocol requires five steps:

1. Announce deployment intent (broadcast to all regions): 80 ms
2. Receive acknowledgments (wait for slowest region): 80 ms
3. Execute deployment phase (region-local): variable
4. Confirm completion (broadcast): 80 ms
5. Receive confirmations (wait for slowest region): 80 ms

Minimum coordination overhead per phase transition: 320 ms for the synchronization protocol itself. For a deployment with 5 canary phases, coordination adds 1.6 seconds to total deployment time, negligible compared to the hours spent in each phase.

However, the coordination service becomes a critical dependency. If the coordinator fails during a deployment, three behaviors are possible depending on the consistency model:

- With strong consistency: All regions freeze in current state until coordinator recovers
- With eventual consistency: Regions continue independent progression, potentially diverging
- With bounded staleness: Regions continue but coordinator failure triggers alerts if staleness exceeds bounds

Organizations deploying safety-critical models typically implement coordinator redundancy through consensus protocols (Raft, Paxos) that survive single-node failures while maintaining consistency guarantees.

12.5.5 Production experiment validation

Once rollout coordination preserves version boundaries, the next question is whether the candidate model is actually safe to expose. Shadow traffic, interleaving, A/B tests, and network-effect checks form an experiment ladder: each step buys stronger production evidence at higher operational cost.

12.5.5.1 Shadow deployment and traffic replay

Shadow deployment runs the new model in parallel with production, receiving the same inputs and logging outputs, but not affecting user-visible results (figure 12.7). This provides the highest fidelity testing environment short of actual production exposure, enabling detection of issues that escape offline validation.

Shadow deployment earns its duplicate infrastructure cost when it answers validation questions offline tests cannot. Operational load testing proves the new model can handle full production traffic volume without crashing, leaking memory, or violating latency service-level objectives (SLOs)[8]. A model that passes offline validation with small datasets may exhibit memory leaks, performance regressions, or resource contention when processing millions of requests per hour. Shadow deployment catches these operational issues before user impact.

Once the shadow path is isolated, the validation signal comes from comparing behavior under real traffic. Output comparison reveals distribution shifts, outlier behavior, and edge cases that aggregate offline metrics hide; for classification models, this might expose systematic shifts in confidence scores, while for recommendation systems it might expose changes in diversity or category distribution. Behavioral validation catches failures on long-tail inputs that appear infrequently in validation sets but thousands of times daily in production traffic. Performance characterization measures actual latency, throughput, and resource consumption, validating capacity assumptions before the model receives user-visible traffic.

Shadow deployment requires capturing and replaying production traffic, and the replay choice determines the trade-off among fidelity, cost, and freshness. Three architectural patterns address different operational requirements.

Live mirroring duplicates every production request to the shadow model in real time. The production model serves the response while the shadow model processes the same input in parallel, providing immediate validation at full production scale but requiring shadow infrastructure capable of handling 100 percent traffic load. To keep this validation path from becoming a production dependency, the serving system must invoke shadow requests asynchronously, enforce timeouts, shed shadow load when it falls behind, and isolate resources so shadow work cannot affect production responses.

[8] **SLO (Service Level Objective)**: A target reliability level. For a model-serving API, "three nines" (99.9 percent) availability allows only 8.7 hours of downtime per year. For latency, a P99 SLO of 200 ms means 99 percent of requests must complete faster than 200 ms. Meeting these targets at scale requires automated scaling and circuit breaking to handle load spikes.

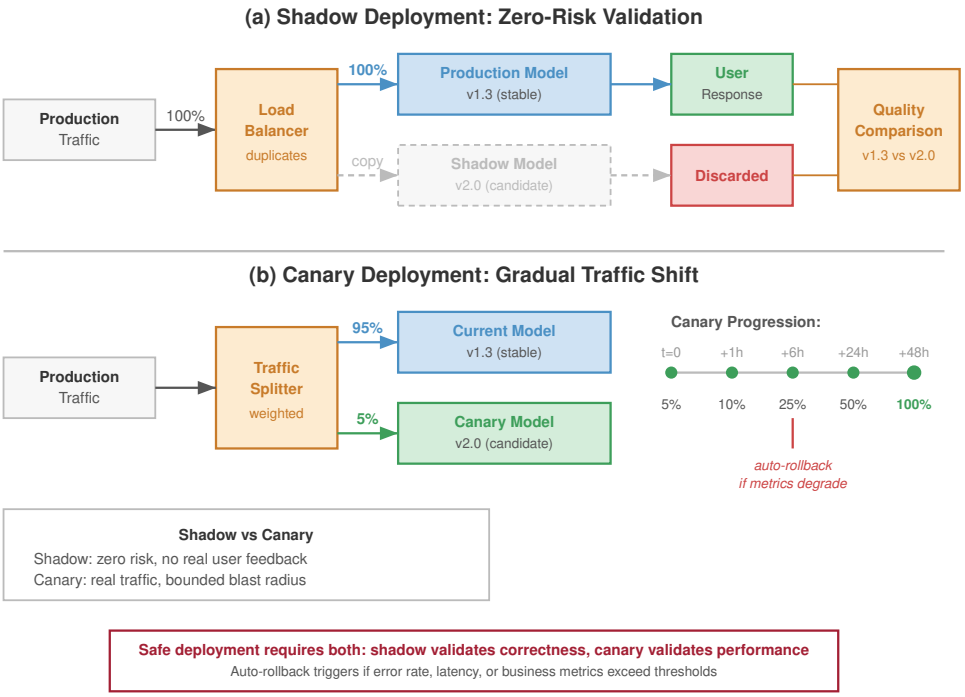


Figure 12.7: Shadow Deployment Architecture: Production traffic is mirrored to the shadow model asynchronously. The router returns the production response to the user immediately, while both responses are logged for offline quality comparison and operational validation.

Sampled replay mirrors a configurable percentage of production traffic to shadow models. This reduces infrastructure costs while maintaining statistical power for validation: a shadow model receiving 10 percent of traffic still processes hundreds of thousands of requests daily at scale, sufficient for detecting most issues. The sampling policy determines which failures remain visible at lower cost. Random sampling is simple and unbiased, stratified sampling preserves representation across user segments and request types, and adaptive sampling increases the rate for patterns where shadow and production outputs diverge.

Batch replay captures production traffic logs and replays them asynchronously against shadow models. This decouples shadow validation from production latency constraints, supports faster-than-real-time regression tests against historical traffic, and allows off-peak cost optimization. The same decoupling weakens the signal for freshness-sensitive systems: validation is delayed by hours or days, requires persistent logging infrastructure, may replay time-dependent features with stale context, and cannot validate real-time operational characteristics.

Effective shadow deployment requires quantitative comparison metrics beyond simple accuracy because the decision is whether the candidate behaves like production under real load. Output divergence measures how shadow predictions differ from production predictions: classification systems track disagreement rates, probability shifts, and class-specific concentration, while regression systems compute root mean square error (RMSE) between shadow and production outputs. Performance metrics compare latency distributions, throughput capability, and resource consumption; a shadow model with equivalent accuracy but 50 percent higher p99 latency requires infrastructure capacity adjustments before deployment.

The metric set must also separate operational failures from genuine model changes. Error-mode metrics count timeouts, exceptions, malformed outputs, and null predictions. A shadow model that times out on 0.1 percent of requests encounters 1,000 failures per day at 1M requests/day scale, and the request patterns that trigger those failures guide remediation. Statistical validation then

determines whether observed differences represent genuine model changes or random variation. For a shadow model processing 100K requests with 1 percent disagreement rate and production model at 1.5 percent disagreement, a two-proportion z-test determines statistical significance:

$$z = \frac{0.015 - 0.010}{\sqrt{0.0125 \times 0.9875 \times (2/100000)}} = \frac{0.005}{0.000497} \approx 10.1$$

With $z > 1.96$, this difference is statistically significant at $\alpha_{\text{sig}} = 0.05$, indicating a genuine shift rather than sampling noise.

12.5.5.2 Worked example: Shadow deployment workflow

A fraud detection model processes 5 million transactions daily. The team develops a new model architecture expected to improve precision while maintaining recall. The shadow deployment workflow begins with a sampled-shadow phase that mirrors 10 percent of traffic for three days, so shadow infrastructure handles 500K requests per day. The observed recall is 94.2 percent for the shadow model vs. 94.5 percent in production, which is not statistically different, while precision improves from 82.3 percent to 87.1 percent with statistical significance. Shadow p99 latency reaches 45 ms vs. 38 ms in production, still acceptable under the 50 ms SLO, so the team proceeds to full shadow.

The full-shadow phase mirrors 100 percent of traffic for five days. This confirms that the precision improvement holds at full scale, but it also exposes an edge case: the shadow model flags 0.02 percent of transactions as errors due to an unexpected feature distribution, which is 1,000 transactions per day. The root cause is heightened sensitivity to outliers in transaction amount. After adjusting feature clipping thresholds and redeploying shadow, the error rate falls to 0.001 percent, clearing the gate for canary deployment.

Postdeployment validation then compares actual production metrics with the shadow predictions. Precision materializes at 87.3 percent in production vs. 87.1 percent in shadow, within expected variation. In this case, shadow deployment successfully predicted production behavior.

12.5.5.3 Shadow deployment infrastructure

Operating shadow deployments at scale requires infrastructure that preserves the validation signal without perturbing production. The traffic mirroring layer intercepts production requests and duplicates them to shadow environments, handling routing logic, sampling decisions, timeout enforcement, and error isolation so shadow failures cannot affect production. Logging and comparison infrastructure captures outputs from both production and shadow models, computes divergence metrics, and stores results for analysis; high-throughput systems can generate terabytes of comparison data, so storage and query design become part of the validation architecture.

The operational surface completes the loop. Alerting and dashboards surface statistically significant divergences, performance regressions, and elevated error rates to deployment decision makers, while drill-down views expose the request patterns that explain divergence. Resource isolation keeps shadow workloads from stealing production capacity through separate compute pools, network bandwidth allocation, and database capacity. Cloud deployments often achieve that isolation with separate clusters; on-premises deployments require explicit resource partitioning.

12.5.5.4 When shadow deployment is essential

Shadow deployment is most valuable when the cost of duplicate infrastructure is smaller than the cost of an undetected production regression:

- New model architectures where offline validation may miss production-specific failure modes
- High-stakes models (financial, medical, safety-critical) where production issues have severe consequences
- Models with complex dependencies on real-time features where offline replay cannot fully validate behavior
- Performance-sensitive deployments where latency or throughput regressions must be detected before user impact
- Regulatory environments requiring preproduction validation evidence

It is less critical when the deployment risk is bounded and rollback is cheap:

- Minor model updates (retraining with same architecture) where production behavior is well-understood
- Low-risk models where rapid rollback is acceptable
- Resource-constrained environments where shadow infrastructure costs exceed validation benefits

The decision is therefore economic as much as technical: pay for shadow validation when duplicate capacity buys risk information that offline tests and cheap rollback cannot provide.

12.5.5.5 Interleaving experiments

Interleaving experiments[9] were developed for ranking and search evaluation (Chapelle et al. 2012) and are also used in recommender personalization experiments (Blog 2017). Rather than splitting users between variants, interleaving presents items from both variants to each user, then measures which items users engage with.

The key insight is statistical efficiency. An interleaving experiment can require far fewer samples than A/B testing, and Netflix reports more than 100× fewer users for some ranking experiments (Chapelle et al. 2012; Blog 2017), because each user provides direct comparison signals rather than contributing only to aggregate statistics.

Interleaving implementation:

1. Both model variants score all candidates
2. Results are interleaved using team draft or probabilistic interleaving
3. User interactions attribute credit to the originating variant
4. Statistical tests determine winning variant

Interleaving is essential for recommendation systems where detecting small engagement changes quickly enables rapid iteration, as figure 12.8 contrasts with traditional A/B testing.

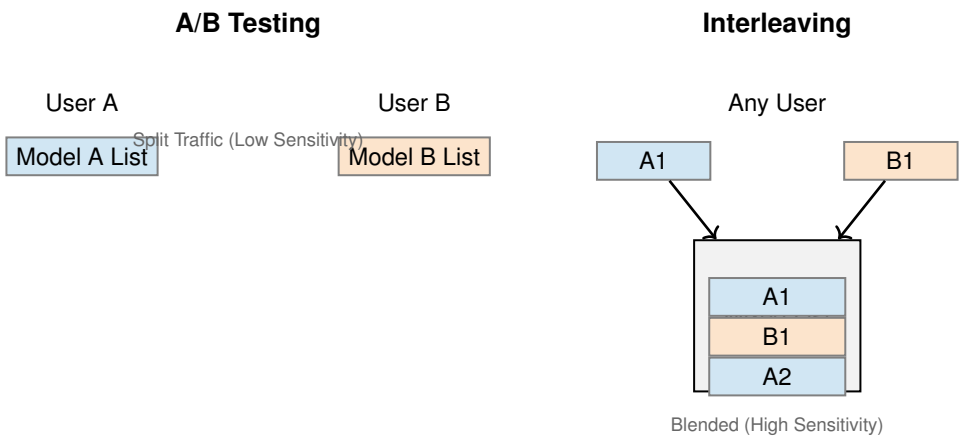


Figure 12.8: Interleaving vs. A/B Testing: In traditional A/B testing (left), users see only one variant. In interleaving (right), users see a blended list. Clicks on items are attributed to the source ranker, providing a higher-sensitivity signal that controls for user-specific variance.

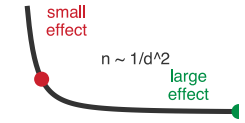
12.5.5.6 A/B testing statistical foundations

The statistical challenges of experimentation multiply at platform scale. A/B testing provides rigorous frameworks for comparing model variants (Kohavi et al. 2009, 2020), but requires careful attention to statistical power[10], significance thresholds, and multiple testing correction.

[9] **Interleaving Experiments**: Originally developed for search engine evaluation, interleaving merges results from two rankers into one list shown to each user, then credits clicks to the originating ranker (Chapelle et al. 2012). It can require far fewer samples than A/B testing because each user provides a direct within-subject comparison rather than only a between-population signal. For fleet-scale recommendation systems iterating on many model variants, that sample-efficiency gain translates directly into faster experiment cycles and lower opportunity cost from suboptimal models in production.

[10] **A/B Testing Power**: The probability that an experiment correctly detects a true effect ($1 - \beta_{\text{stat}}$, typically set to 0.8). At fleet scale, the systems challenge is Statistical Power: if the expected improvement is small (for example, 0.1 percent), the experiment may require millions of samples to reach statistical significance, consuming substantial serving resources (C_{infer}) for weeks.

12.5.5.7 Sample size calculation



Detecting smaller effects explodes the required sample size.

Improper statistical practices lead to false positives that waste engineering resources or false negatives that miss genuine improvements, so the required sample size for detecting an effect must be chosen from four parameters: significance level (α_{sig}), statistical power ($1-\beta_{\text{stat}}$), baseline conversion rate (p), and minimum detectable effect (δ). Equation 12.6 formalizes this relationship for comparing two proportions, showing that required samples scale inversely with the square of the minimum detectable effect:

$$n_{\text{sample}} = \frac{(Z_{\alpha_{\text{sig}}} + Z_{\beta_{\text{stat}}})^2 \times 2p(1-p)}{\delta^2} \tag{12.6}$$

where $Z_{\alpha_{\text{sig}}}$ is the critical value for significance level α_{sig} (typically 1.96 for $\alpha_{\text{sig}} = 0.05$), $Z_{\beta_{\text{stat}}}$ is the critical value for power (typically 0.84 for 80 percent power), p is the baseline rate, and δ is the minimum detectable effect as an absolute difference.

12.5.5.8 Worked example: Sample size for recommendation model

A recommendation system has baseline CTR of 5 percent. The team wants to detect a 10 percent relative improvement (0.5 percentage points absolute) with 95 percent confidence and 80 percent power.

Parameters:

- $Z_{\alpha_{\text{sig}}} = 1.96$ (95 percent confidence, two-tailed)
- $Z_{\beta_{\text{stat}}} = 0.84$ (80 percent power)
- $p = 0.05$ (baseline CTR)
- $\delta = 0.005$ (0.5 percentage point improvement)

Calculation:

$$n_{\text{sample}} = \frac{(1.96 + 0.84)^2 \times 2 \times 0.05 \times 0.95}{0.005^2}$$

$$n_{\text{sample}} = \frac{7.84 \times 0.095}{0.000025} = \frac{0.7448}{0.000025} = 29,792$$

Each variant requires approximately 30,000 samples, totaling 60,000 observations. At 1 million requests per day, this experiment requires less than 2 hours. However, for a model with 1 percent baseline CTR detecting a 5 percent relative improvement (0.05 percentage points), the calculation yields:

$$n_{\text{sample}} = \frac{7.84 \times 2 \times 0.01 \times 0.99}{0.0005^2} = \frac{0.1552}{0.00000025} = 620,800$$

Now each variant needs approximately 621K samples. At 1M total requests/day split evenly between the two variants, the experiment requires about 30 hours. The lower the baseline rate and smaller the effect, the longer the experiment must run.

12.5.5.9 Statistical significance testing

Once data is collected, a two-proportion z-test determines if the observed difference is statistically significant. Equation 12.7 computes the test statistic as the difference in observed rates normalized by the pooled standard error:

$$z = \frac{\hat{p}_B - \hat{p}_A}{\sqrt{\hat{p}(1-\hat{p})(\frac{1}{m_A} + \frac{1}{m_B})}} \tag{12.7}$$

where $\hat{p}_A$ and $\hat{p}_B$ are the observed conversion rates for control and treatment, m_A and m_B are sample sizes, and $\hat{p} = \frac{m_A\hat{p}_A + m_B\hat{p}_B}{m_A + m_B}$ is the pooled proportion. If $|z| > Z_{\alpha_{\text{sig}}}$, reject the null hypothesis and conclude the variants differ significantly.

12.5.5.10 Multiple testing correction

Running multiple A/B tests simultaneously or sequentially without correction inflates the familywise error rate. With 20 independent tests at $\alpha_{\text{sig}} = 0.05$, the probability of at least one false positive is:

$$\Pr(\text{at least one false positive}) = 1 - (1 - \alpha_{\text{sig}})^k = 1 - 0.95^{20} = 0.642$$

The result is a 64 percent chance of falsely detecting an improvement. Three correction approaches address this problem.

Bonferroni correction adjusts the significance threshold to $\alpha'_{\text{sig}} = \frac{\alpha_{\text{sig}}}{k}$ for k tests. This is conservative but simple. For 20 tests with $\alpha_{\text{sig}} = 0.05$, use $\alpha'_{\text{sig}} = 0.0025$ for each test. This controls the familywise error rate but reduces statistical power.

Šidák correction provides a less conservative adjustment. Equation 12.8 computes the per-test threshold that maintains the desired familywise error rate exactly, yielding slightly more statistical power than Bonferroni:

$$\alpha'_{\text{sig}} = 1 - (1 - \alpha_{\text{sig}})^{1/k} \tag{12.8}$$

For 20 tests: $\alpha'_{\text{sig}} = 1 - 0.95^{1/20} = 0.00256$, slightly more lenient than Bonferroni.

False Discovery Rate (FDR) control using Benjamini-Hochberg procedure allows a specified proportion of false positives among all rejections. This is appropriate when some false positives are acceptable. Order p-values from smallest to largest: $p_{(1)} \leq p_{(2)} \leq \ldots \leq p_{(k)}$. Find the largest i such that:

$$p_{(i)} \leq \frac{i}{k} \times \alpha_{\text{sig}}$$

Reject all hypotheses $H_{(1)}, \ldots, H_{(i)}$. This procedure yields higher statistical power than Bonferroni when running many tests.

12.5.5.11 Sequential testing and early stopping

Traditional A/B tests fix sample size in advance and evaluate once. Sequential testing allows monitoring results during data collection with principled early stopping rules. This can reduce experiment duration by 50 percent or more while controlling error rates.

The sequential probability ratio test (SPRT) evaluates the likelihood ratio after each observation:

$$\Lambda_m = \frac{p(X_1, \ldots, X_m \mid H_1)}{p(X_1, \ldots, X_m \mid H_0)}$$

Stop and reject H_0 if $\Lambda_m \geq \frac{1-\beta_{\text{stat}}}{\alpha_{\text{sig}}}$, stop and accept H_0 if $\Lambda_m \leq \frac{\beta_{\text{stat}}}{1-\alpha_{\text{sig}}}$, otherwise continue collecting data.

For large-scale A/B testing, group sequential methods divide the experiment into planned analysis stages. At each stage, compare test statistic to adjusted thresholds (computed using methods such as O'Brien-Fleming or Pocock boundaries) that maintain overall α_{sig}.

12.5.5.12 Practical implementation considerations

Real-world A/B testing faces complications beyond textbook statistics. Carryover effects occur when users exposed to treatment retain behavior changes after returning to control. This violates independence assumptions, so mitigation requires sufficient washout periods or cookie-based consistent assignment.

Network effects occur when treating user A affects user B's behavior through interaction. This violates the stable unit treatment value assumption (SUTVA), so mitigation uses cluster randomization at network community level, though this reduces statistical power (Rubin 1980; Hudgens and Halloran 2008; Eckles et al. 2017).

Novelty effects occur when new model variants show artificial improvement because users respond to novelty, not genuine superiority. Mitigation extends experiment duration (typically 2–4 weeks) to observe steady-state behavior.

Metric selection can mislead when surrogate metrics (clicks, engagement) do not align with long-term objectives (retention, revenue). Mitigation tracks both short-term surrogate metrics and long-term guardrail metrics, even if the latter require longer observation periods.

12.5.5.13 Worked example: Multiple testing scenario

A platform team runs 30 A/B tests per quarter comparing candidate models. Using $\alpha_{\text{sig}} = 0.05$ without correction, expect $30 \times 0.05 = 1.5$ false positives per quarter. Over a year, expect approximately 6 models falsely identified as improvements, wasting engineering effort on deployments that provide no actual value.

Applying Bonferroni correction: $\alpha'_{\text{sig}} = \frac{0.05}{30} = 0.00167$ per test. This requires larger sample sizes. For the recommendation model example above (5 percent baseline, 0.5pp effect), original requirement was 30K samples per variant. With a two-sided Bonferroni threshold, the critical value rises to about 3.14, increasing the requirement to roughly 60K samples per variant.

Using FDR control at $q = 0.05$: Across repeated use, the Benjamini-Hochberg procedure controls the expected proportion of false discoveries among rejected hypotheses. If 10 of 30 tests are declared significant, the target is an expected false discovery proportion of 5 percent among those discoveries, not a hard maximum count in that single quarter. This provides better power than Bonferroni when running many tests.

The choice of correction method depends on consequences of false positives. For high-stakes decisions (financial models, safety-critical systems), use conservative Bonferroni correction. For exploratory analysis where missing true effects is costly, use FDR control.

> Checkpoint 12.4: Family-wise error rate under multiple testing
>
> The correction methods are Bonferroni, Šidák, and Benjamini-Hochberg. Apply the family-wise reasoning to a smaller platform.
>
> - ☐ **Family-wise error**: For 20 simultaneous A/B tests per quarter at $\alpha_{\text{sig}} = 0.05$ with no correction, calculate the expected number of false positives and the family-wise error rate under independence.
> - ☐ **Bonferroni correction**: Apply Bonferroni correction, calculate the per-test threshold α'_{sig}, compute the family-wise error rate after correction, and compare it to the uncorrected rate.
> - ☐ **Bonferroni vs. FDR**: For 20 tests where the team expects most candidates to genuinely improve the metric, choose Bonferroni or Benjamini-Hochberg FDR control and justify the statistical-power trade-off.

12.5.6 SUTVA violations and network effects

The statistical results above rest on a fundamental assumption: independence between users. A treatment assigned to one user must not change another user's outcome. If User A receives an improved recommendation algorithm, User B's behavior should remain unaffected because User B never saw the new algorithm. This independence assumption underpins all the sample size calculations and significance tests presented so far.

Social platforms systematically violate this assumption. When User A receives better content recommendations, they share that content with their network, including User B in the control group. User B's engagement changes despite never seeing the treatment. The control condition becomes contaminated, not through experimental error, but through the natural mechanics of networked products.

Formally, standard A/B testing relies on the Stable Unit Treatment Value Assumption (SUTVA): user i's outcome $Y_i(w)$ depends only on their own treatment assignment w, not on the treatments assigned to other users (Rubin 1980). This assumption fails systematically in networked products and distributed ML systems, leading to biased effect estimates that can mislead deployment decisions (Hudgens and Halloran 2008; Eckles et al. 2017).

12.5.6.1 Network effect categories

Network effects manifest in three primary forms, each requiring different detection and mitigation strategies:

- **Direct network effects**: User A's treatment directly influences user B's outcome through platform interactions. In a social feed ranking experiment, treatment users may share algorithmically optimized content with control users, biasing measured effects toward zero because control users partially receive the treatment through network propagation.
- **Indirect network effects**: Market-level mechanisms affect all users regardless of their individual treatment assignment. A ride-sharing pricing experiment that increases driver compensation in the treatment group can attract more drivers overall, causing control users to experience shorter wait times and making the measured treatment effect underestimate the true benefit.
- **Spillover effects**: Treatment effects spread across geographic or temporal boundaries. A local recommendation experiment can shift restaurant foot traffic and popularity signals in adjacent neighborhoods, changing recommendation quality for nearby control users.

In all three cases, the experiment no longer measures independent user-level treatment effects.

12.5.6.2 Quantifying SUTVA violations

The severity of network effect bias depends on network structure and outcome correlation. For cluster-randomized experiments, equation 12.9 gives a common design-effect approximation that quantifies how within-cluster correlation inflates the variance of treatment effect estimates, directly determining the sample size increase required for equivalent statistical power:

$$\text{DEFF} \approx 1 + (m-1)\rho \tag{12.9}$$

where m is the average cluster size and ρ is the intra-cluster correlation of outcomes (how similar outcomes are within connected user groups). This factor indicates how much larger sample sizes must be to achieve equivalent statistical power. Network interference can also bias the estimated treatment effect itself, so variance inflation is only part of the correction.

12.5.6.3 Worked example: Network effect bias in social recommendation

A social platform tests a new feed ranking algorithm. Individual user randomization assigns 50 percent of users to treatment.

Experimental setup:

- 10 million users, average 150 connections each
- Clustering coefficient $C = 0.4$ (typical for social networks)
- Outcome: daily engagement minutes

Naive analysis results:

- Treatment group: 45.2 minutes average
- Control group: 43.8 minutes average
- Measured effect: +1.4 minutes (+3.2 percent)

However, network analysis reveals that control group users have on average about half of their connections in treatment, as expected under independent individual randomization. These treatment connections share algorithmically-boosted content that control users see, inflating control group engagement.

Corrected analysis using inverse probability weighting for network exposure:

- Adjusted control baseline: 42.3 minutes (what control would show without spillover)
- True treatment effect: +2.9 minutes (+6.9 percent)
- SUTVA violation inflated control by 1.5 minutes, halving the measured effect

With intra-cluster correlation $\rho = 0.15$ and an average experiment cluster size of $m = 1.4$ effective users:

$$\text{DEFF} = 1 + (1.4 - 1) \times 0.15 = 1.06$$

This toy calculation gives 6 percent variance inflation and would require 6 percent larger sample sizes for equivalent power, but the bias correction is far more impactful than the variance adjustment.

12.5.6.4 Detection strategies

Detecting SUTVA violations requires explicit measurement of network exposure through three complementary checks:

- **Ego-network analysis**: Measure each control user's exposure to treatment through their connections using $e_i = |\{j \in \mathcal{N}_i : W_j = 1\}|/|\mathcal{N}_i|$, where $\mathcal{N}_i$ is the connection set for control user i and W_j indicates treatment assignment for user j. If control group outcomes correlate with e_i, network effects are present; regression of control outcomes on exposure quantifies spillover magnitude.
- **Interference tests**: Compare outcomes for control users with high versus low treatment exposure. Under SUTVA, these groups should show identical outcomes, so significant differences indicate network contamination.
- **Temporal analysis**: Examine whether treatment effects propagate over time. If day-over-day control group metrics trend toward treatment group metrics, spillover is accumulating through the network.

Together, these checks turn spillover from a hidden assumption violation into a measurable experimental condition.

12.5.6.5 Mitigation approaches

When SUTVA violations are detected, four experimental design modifications can recover valid causal estimates:

- **Graph cluster randomization**: Treatment is assigned at the community level rather than the individual level. Graph partitioning algorithms such as Louvain or spectral clustering divide the user graph into clusters with dense internal connections and sparse cross-cluster edges, then randomize clusters so users primarily interact with others in the same condition. The trade-off is reduced statistical power: with k clusters, effective sample size becomes k rather than the total individual-user count.
- **Ego-exclusion designs**: Users whose network exposure exceeds a threshold are excluded from analysis. By analyzing only control users with minimal treatment connections, for example $e_i < 0.05$, the control condition remains uncontaminated at the cost of sample size.
- **Switchback experiments**: All users alternate between treatment and control over time periods such as hours or days. Since all users receive both conditions, there is no cross-user contamination within periods, and analysis compares outcomes across time periods rather than across users.
- **Geo-based experiments**: Geographic boundaries act as natural barriers to network effects. For location-dependent services, city-level or region-level randomization eliminates most spillover pathways because users in different cities rarely interact directly.

The right design is the one that removes the dominant interference path without destroying statistical power.

12.5.6.6 Practical implementation

Implementing network-aware A/B testing requires infrastructure investment:

- Graph analysis pipelines that compute network statistics and cluster assignments
- Exposure calculation for every user based on their connections' treatment status
- Modified statistical tests that account for clustered randomization
- Monitoring dashboards showing spillover indicators

For recommendation systems at scale, the engineering cost is justified by the magnitude of bias that network effects introduce. A system measuring +3 percent improvement when the true effect is +6 percent may incorrectly reject valuable model changes or incorrectly prioritize inferior alternatives.

12.5.7 Managing the edge fleet

The operational challenges examined thus far assume a controlled data center environment. However, Chapter 11 demonstrated that federated learning and on-device ML extend the "Machine Learning Fleet" to millions of heterogeneous edge devices with constrained connectivity, power, and compute. Managing this distributed population introduces three distinct MLOps challenges.

The three constraints interact, so they should be handled as one operational design problem rather than as independent notes. Table 12.16 maps each constraint to the platform control it requires, including Hardware-in-the-Loop validation and Federated Analytics.

Table 12.16: Edge Fleet Controls: Edge operations couple rollout latency, device heterogeneity, and privacy-limited observability. The management layer must reconstruct global fleet health from partial signals while preserving compatibility with many active model versions.

Edge fleet constraint	Why it appears	Platform control
Extreme version skew	Rollouts can take weeks or months because devices are offline, on low battery, or on restricted networks. At any time, 50 model versions may remain active.	Maintain backward-compatible data pipelines and serving contracts for models deployed months earlier.
Device-aware validation	Accuracy gates do not reveal whether a model exceeds a 1 MB microcontroller memory budget or triggers thermal throttling on a smartphone system on chip (SoC).	Add Hardware-in-the-Loop (HIL) validation on physical or emulated target devices before promotion.
Privacy-limited telemetry	Raw predictions cannot stream back continuously because privacy rules and bandwidth costs constrain observability.	Use Federated Analytics: devices compute local statistics such as error rates or drift, then transmit only aggregated, anonymized health signals.

12.5.8 Rollout risk management

Not all deployments carry equal risk. Effective CI/CD systems classify and handle deployments based on their risk profile. Table 12.17 provides risk-based rollout strategy selection, mapping four risk categories to appropriate rollout strategies: low-risk minor fixes proceed through fast canary, while critical core model changes require the full shadow deployment, human review, and staged rollout sequence.

12.5.8.1 Risk classification

Equation 12.10 formalizes deployment risk as the product of regression probability, impact severity, and exposure level, providing a quantitative foundation for risk-based rollout decisions:

$$R_{\text{rollout}} = p_{\text{regression}} \times I_{\text{regression}} \times E_{\text{exposure}} \tag{12.10}$$

where $p_{\text{regression}}$ is the probability that the change causes a regression, $I_{\text{regression}}$ is the impact severity if regression occurs, and E_{exposure} is the exposure level during the rollout period.

The rollout risk framework suggests three mitigation strategies:

- Reduce $p_{\text{regression}}$: More thorough testing before deployment
- Reduce $I_{\text{regression}}$: Architectural patterns that limit blast radius
- Reduce E_{exposure}: Slower rollouts with lower initial traffic percentages

These levers turn the scalar risk formula into a rollout policy: each risk category chooses how much evidence to collect, how small the blast radius must remain, and how slowly exposure should grow.

12.5.8.2 Risk categories

The risk equation becomes actionable only when the platform maps measured risk to a rollout policy. Table 12.17 turns probability, impact, and exposure into deployment posture: low-risk changes can move through a fast canary, while safety-critical changes need shadow validation, human review, and slower traffic expansion.

Table 12.17: Risk-Based Rollout Strategy Selection: Four qualitative risk categories mapped to deployment strategies. Teams compute numeric rollout risk using the probability, impact, and exposure formula above, then choose the rollout pattern that matches the resulting risk profile.

Category	$p_{\text{regression}}$	$I_{\text{regression}}$	Rollout Strategy
Low	Minor code fix	Limited user impact	Fast canary
Medium	Retrained model	Engagement effects	Standard canary
High	New architecture	Revenue impact	Extended shadow + slow canary
Critical	Core model change	Safety implications	Shadow + human review + staged

12.5.8.3 Automated rollback triggers

Risk categories decide how cautiously traffic should expand, but rollback triggers decide when expansion must stop. The trigger configuration in listing 12.4 makes that stop condition executable by binding each monitored metric to a threshold, observation window, and minimum sample count.

Listing 12.4: Automated Rollback Configuration: Metric-specific thresholds, observation windows, and minimum sample sizes that balance sensitivity against false triggers.

```
rollback_config = {
    "metrics": {
        "engagement_rate": {
            "threshold": -0.02,  # 2% relative decline triggers rollback
            "window_minutes": 15,
            "min_samples": 1000,
        },
        "error_rate": {
            "threshold": 0.01,  # 1% absolute increase triggers rollback
            "window_minutes": 5,
            "min_samples": 500,
        },
        "latency_p99": {
            "threshold": 1.5,  # 50% relative increase triggers rollback
            "window_minutes": 5,
            "min_samples": 100,
        },
    },
    "rollback_action": "immediate",  # or 'gradual' for less severe issues
    "notification": ["oncall", "model-owner"],
}
```

Automated rollback must balance sensitivity against false triggers. The statistical significance requirements (minimum samples, window duration) prevent premature rollback from random fluctuation while enabling rapid response to genuine regressions.

12.5.9 CI/CD patterns by model type

Model type determines which CI/CD constraint binds: semantic quality, engagement power, adversarial urgency, or classification accuracy. Table 12.18 contrasts CI/CD patterns by model type: LLMs require quality-gated pipelines with human evaluation taking days to weeks, while fraud detection uses threshold-gated pipelines enabling hours-fast deployment with seconds-fast rollback to counter adversarial dynamics.

Table 12.18: CI/CD Patterns by Model Type: Validation focus, rollout speed, and rollback capabilities vary by model type. LLMs require quality-gated pipelines with human evaluation taking days to weeks for deployment, while fraud detection uses threshold-gated pipelines enabling hours-fast deployment with seconds-fast automated rollback to counter adversarial dynamics.

Pattern	Model Type	Validation Focus	Rollout Speed	Rollback Speed
Quality-gated	LLM	Human eval, safety	Days to weeks	Hours
Metric-driven	Recommendation	Engagement metrics	Hours to days	Minutes
Threshold-gated	Fraud	Precision/recall	Hours	Seconds
Accuracy-focused	Vision	Classification metrics	Days	Minutes

For LLMs, the binding constraint is semantic and safety quality, so the pipeline is deliberately slow. Automated benchmark evaluation on MMLU, HumanEval, and similar tasks narrows the candidate set; human evaluation checks sample outputs across capability categories; safety evaluation exercises red-teaming and toxicity cases; shadow deployment measures user-satisfaction signals; and a slow staged rollout gives the release an extended soak period. The full cycle may take 2–4 weeks from candidate model to full deployment because the main risk is a subtle regression that ordinary automated metrics miss.

Recommendation systems bind on freshness and engagement power, so the pipeline prioritizes rapid but statistically defensible comparison. Offline NDCG and recall screen candidate models, interleaving compares the candidate against the production baseline on the same requests, significance tests decide whether engagement moved, and a rapid canary promotes or rolls back automatically. Routine updates may complete in 24–48 hours because the cost of stale recommendations can exceed the cost of a carefully bounded experiment.

Fraud models bind on adversarial urgency. The pipeline still evaluates labeled fraud cases, validates the false-positive rate on legitimate traffic, and shadow-scores transactions to compare precision and recall, but it is designed around rapid deployment with instant rollback capability because attackers adapt while the rollout is still in progress. Routine updates may complete in 4–12 hours, and emergency updates may deploy in under 1 hour when new fraud patterns emerge.

A mature CI/CD pipeline ensures that only healthy, verified models reach production, completing the deployment cycle in hours rather than weeks. Deployment is not the *finish line*; it is the *starting line*. Once a model is safely deployed, the primary operational question is whether it continues to perform as expected under shifting conditions. Answering that question requires a monitoring architecture that scales alongside automated deployment.

12.6 Monitoring at Scale

Application-layer drift detection depends on a healthy physical substrate. At fleet scale, a degraded NVLink connection or a thermally throttling GPU can masquerade as a software timeout. Monitoring therefore starts bottom-up with fleet telemetry, then climbs through alert aggregation, model-quality signals, dashboards, and cost observability so that operators can route a symptom to the layer that can actually repair it.

12.6.1 Fleet telemetry and hardware observability

Because failure is routine at pod scale, fleet monitoring and hardware observability are not operational conveniences but prerequisites for productive training. A 10,000-GPU cluster experiences hardware failures roughly once every five hours, and the difference between a 10-minute recovery and a multi-hour disruption depends on whether the operations team can detect degradation before it becomes failure. The monitoring system is itself a distributed system, collecting telemetry from every GPU, switch, and cooling component in the fleet.

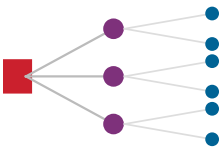
The telemetry spans three physical levels. At the GPU level, the most diagnostic signals are junction temperature (which reveals cooling degradation), ECC error counts in HBM and SRAM (where a rising rate of correctable errors often precedes a crash-inducing uncorrectable error), and SM utilization (which distinguishes hardware faults from software inefficiencies). At the network level, packet retry rates on individual ports reveal failing cables or switches. At the facility level,

coolant temperature deviations trigger automated load shedding before thermal damage occurs. The challenge is not collecting these signals individually but correlating them across spatial and temporal dimensions.

The most operationally valuable monitoring capability is **correlation analysis** across these telemetry streams. A single GPU showing elevated temperature might indicate a failing fan or a coolant flow restriction. If 8 GPUs in the same node simultaneously show elevated temperatures, however, the cause is more likely a node-level cooling issue. If all GPUs in a rack show elevated temperatures, the cause is probably a rack-level CDU problem. If GPUs across multiple racks show coordinated temperature changes, the cause is likely a facility-level event such as a chiller failure. The monitoring system must detect and classify these patterns at the correct spatial scale to direct operators to the actual root cause, rather than generating hundreds of independent alerts for what is a single underlying problem.

The scale of this telemetry is nontrivial. A 10,000-GPU cluster generates approximately 1 TB of metric data per day – junction temperatures sampled every second across 10,000 GPUs alone produce 864 million data points daily, before accounting for ECC counters, power readings, NVLink error rates, and network port statistics. Ingesting, indexing, and querying this firehose of time-series data with sub-second latency requires its own dedicated infrastructure. Organizations typically dedicate 1–2 percent of the cluster's total compute and storage capacity to the monitoring stack itself: time-series databases (Prometheus, InfluxDB, or custom solutions), alerting engines, and dashboard systems. This operational overhead is the cost of visibility at scale; without it, the fleet is flying blind.

Before a training job is allocated to a set of nodes, the scheduler employs automated preflight checks to verify hardware health. The control plane runs a battery of short, intensive diagnostics: GEMM benchmarks to verify Tensor Core throughput, NCCL AllReduce tests to validate NVLink and InfiniBand bandwidth, and memory stress tests to catch weak HBM bit cells that might produce uncorrectable errors under sustained load. A node that underperforms on any diagnostic is automatically quarantined for repair, and a healthy replacement is substituted before the job launches. This validation process adds 5–10 minutes to job startup time, a negligible cost compared to the hours of wasted computation when a training run crashes mid-flight due to a degraded GPU that passed a simple power-on self-test but fails under sustained arithmetic load.



One slow accelerator stalls every peer at the barrier.

The most insidious adversary in a large fleet is not the hard failure but the gray failure: a component that continues to function but at degraded performance. A single GPU with a partially failed HBM stack might operate at only 75 percent of its peak bandwidth. An NVLink with marginal signal integrity might force frequent link retraining, causing microsecond stalls that accumulate into seconds of lost time per training step. In a synchronous data-parallel workload, a single straggler slows the entire cluster, because every other GPU must wait for the slowest participant to complete its AllReduce contribution. These gray failures are invisible to simple "up/down" health checks and require continuous, fine-grained performance benchmarking to detect. The most effective approach is to run periodic micro-benchmarks on idle nodes (or during scheduled maintenance windows) and compare each node's performance against the fleet baseline. A node whose GEMM throughput drops below 90 percent of the fleet median, or whose NVLink bandwidth drops below 85 percent, is flagged for investigation even though it has not experienced any hard error.

> **Systems Perspective 12.4: Proactive vs. reactive maintenance**
>
> Fleet operators have learned, often through costly experience, that proactive maintenance dramatically reduces the impact of hardware failures on training productivity. The three pillars of proactive maintenance are: predictive diagnostics (using models trained on historical telemetry to predict component failures 24–72 hours before they occur), scheduled burn-in testing (running benchmark workloads on newly installed nodes before assigning production work), and rolling maintenance windows (cycling 2–5 percent of nodes through health checks without reducing available capacity). Organizations that invest in proactive maintenance typically achieve 95–98 percent effective fleet utilization, compared to 80–90 percent for organizations that rely on reactive maintenance.

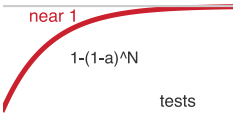
In multi-tenant clusters where multiple training jobs share the same physical infrastructure, the noisy neighbor problem introduces a performance hazard that is invisible to individual job metrics. While containerization strictly limits CPU and memory usage, the network fabric is often a shared resource susceptible to interference. If Job A initiates a massive AllReduce operation across the spine switches just as Job B attempts to fetch training data from networked storage, the resulting micro-bursts of packet contention can throttle Job B's throughput by 30–40 percent. This interference is particularly pernicious in RDMA-enabled clusters where traffic bypasses the host CPU, rendering standard OS-level packet scheduling ineffective. Modern orchestration mitigates this via static rail alignment – physically dedicating specific InfiniBand subnets to specific jobs – or by deploying congestion notification protocols that throttle aggressive flows at the switch hardware level. For organizations running the 175B model training alongside smaller research experiments, the safest approach is to physically partition the cluster into isolated "islands" with dedicated network fabrics, accepting the utilization penalty of fragmentation in exchange for performance predictability.

Checkpoint cadence is the point where hardware telemetry becomes visible to ML operators rather than only to data-center technicians. A degraded node that forces frequent checkpoint recovery, or a storage path that lengthens checkpoint writes, turns directly into lost training throughput. Section E.2.1 derives the checkpoint-compute trade-off and the Young/Daly interval, which section 7.1.2 applies to checkpointing strategies; the operations responsibility here is to expose the telemetry those formulas depend on: write time, failure rate, straggler behavior, and recovery duration. Without that visibility, a platform may report that a job is "running" while useful learning has quietly collapsed behind retries and slow checkpoints.

With the physical substrate accounted for, monitoring returns to the deployed model fleet. Models that pass validation gates and survive canary deployment enter a production environment where gradual degradation, data drift, and emergent interactions can erode performance over weeks or months. The staged rollout strategies and rollback triggers examined in CI/CD detect acute failures during deployment; monitoring systems must detect chronic degradation during operation. At platform scale, where hundreds of models operate simultaneously, the naive approach of applying single-model monitoring practices to each model independently leads to alert fatigue, missed correlations, and operational chaos. Monitoring strategies appropriate for enterprise-scale ML platforms require hierarchical aggregation and systemic governance.

12.6.2 The alert fatigue problem

The mathematical reality of monitoring at scale exposes the limitations of per-model alerting. Consider the mathematics of monitoring 100 models with independent alerting. If each model has 10 monitored metrics, and each metric generates alerts at a 0.3 percent false positive rate (3-sigma thresholding), the expected number of false alerts is still substantial.



As tests multiply, false alerts become mathematically inevitable.

Napkin Math 12.8: The false alarm tax

Problem: Consider a system monitoring 100 models. Each model has 10 metrics (latency, accuracy, drift, and others). Alert thresholds are set at 3-sigma (99.7 percent specificity), and a control script re-evaluates every metric on a fixed interval. This configuration generates a massive volume of false alarms that the on-call engineer must address each day.

Math:

1. **Total Monitors**: 100 models $\times$ 10 metrics = 1,000 monitors.
2. **False Positive Rate**: $1 - 0.997 = 0.003$ (0.3 percent).
3. **Checks per day**: Assume checks every 5 minutes (288 checks/day).
4. **Daily false alarms**: $1{,}000 \times 288 \times 0.003 \approx 864$ alerts/day.

Systems insight: Even with high-specificity (3-sigma) alerts, scale becomes problematic. It is not feasible to alert on raw metrics. One must use *hierarchical aggregation* (for example, "Cluster Health" instead of "Node Health") to survive the false alarm tax.

Equation 12.11 reveals the mathematical inevitability of alert fatigue at scale: for a single metric with false positive rate α_{fp}, the probability of at least one false alert grows exponentially with the number of independent tests N_{tests}:

$$\Pr(\text{at least one false alert}) = 1 - (1 - \alpha_{\text{fp}})^{N_{\text{tests}}} \tag{12.11}$$

With $\alpha_{\text{fp}} = 0.05$ and $N_{\text{tests}} = 1000$ (100 models $\times$ 10 metrics):

$$\Pr(\text{false alert}) = 1 - (1 - 0.05)^{1000} = 1 - 0.95^{1000} \approx 1.0$$

The probability is essentially 100 percent. At this scale, the monitoring system will generate false alerts continuously. This creates a destructive dynamic: operators learn to ignore alerts because most are false, genuine issues get lost in the noise, and the monitoring system provides negative rather than positive value.

12.6.2.1 Worked example: Alert volume calculation

The 3-sigma analysis above already strains a team; loosening to the more common 2-sigma threshold (a 5 percent false positive rate) makes the load untenable. An ML platform monitors 100 models with the following configuration:

- 10 metrics per model (accuracy, latency p50, latency p99, throughput, error rate, data freshness, feature drift, memory usage, GPU utilization, request volume)
- Alert threshold at 2 standard deviations (approximately 5 percent false positive rate per metric)
- Metrics checked every 5 minutes

Expected daily false alerts: Daily false alerts $= 100 \times 10 \times 0.05 \times \frac{24 \times 60}{5} = 14,400$.

Even if 99 percent of these are deduplicated or auto-resolved, the remaining 144 alerts daily overwhelm any on-call team. The monitoring system becomes useless despite (or rather, because of) comprehensive coverage.

12.6.3 Hierarchical monitoring architecture

The alert fatigue problem demands a fundamentally different approach. The solution is hierarchical monitoring (figure 12.9) that turns monitoring into an alert-routing system: broad signals decide whether the fleet is impaired, portfolio signals decide which model family or domain owns the incident, model signals support local diagnosis, and infrastructure signals route failures to the platform team. The hierarchy reduces alert volume while preserving the path from symptom to owner.

The hierarchy works because each level answers a different routing question. Table 12.19 maps each level to its signal, owner, and operational role.

Table 12.19: Hierarchical Monitoring Levels: Monitoring levels separate alerting from diagnosis. Higher levels decide whether and where to route an incident; lower levels preserve the evidence needed to identify whether the cause is model behavior, data drift, serving saturation, or infrastructure failure.

Level	Representative signals	Primary owner	Operational role
Business	Revenue or conversion attributed to recommendations, engagement indicators, automation rate, and human-review volume.	Incident lead	Few high-confidence alerts that indicate product or business impairment.
Portfolio	Engagement lift and diversity for recommendation, fraud caught and false-positive rate, or violation detection and appeal rate.	Domain team	Routes investigation to the model family or product area that owns the shared objective.
Model	Task quality, latency distributions, throughput, error rates, resource utilization, serving saturation, and recent deployment state.	Model owner	Supports local diagnosis after business or portfolio signals move.
Infrastructure	GPU cluster utilization and availability, feature store latency, training pipeline time, serving health, networking, and storage.	Platform team	Routes failures that require capacity, placement, networking, storage, or control-plane repair.

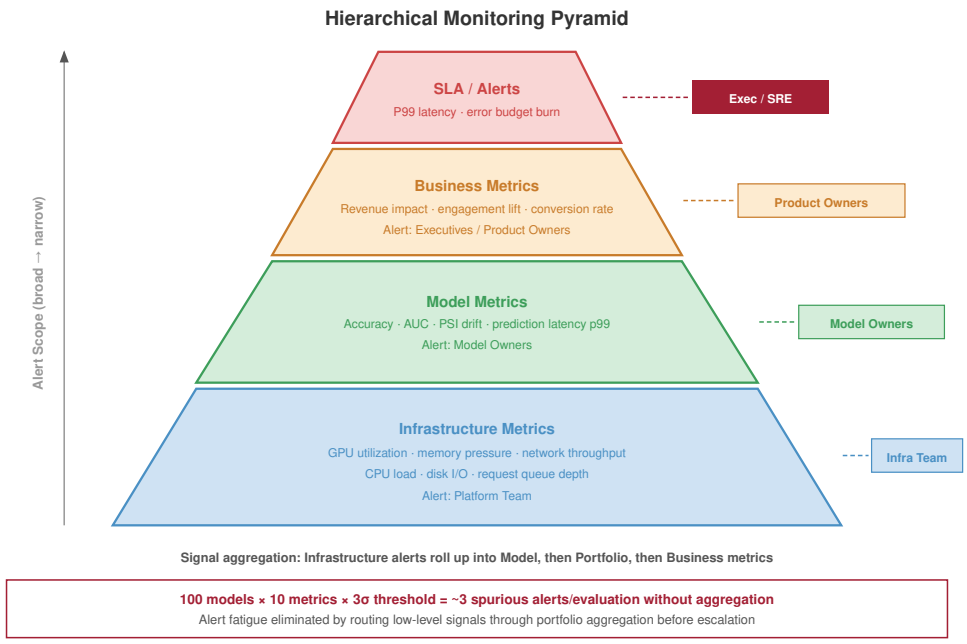


Figure 12.9: Hierarchical Monitoring Architecture: To prevent alert fatigue, monitoring operates at four abstraction levels. High-level business metrics trigger alarms for broad issues, while lower-level metrics are used primarily for investigation and root cause analysis.

12.6.4 Anomaly detection across the fleet

Rather than alerting on individual metric thresholds, fleet-wide anomaly detection identifies unusual patterns across the model portfolio. The detection stack proceeds from local statistical process control, to correlation across model families, to drift scores that explain why a metric moved. That ordering keeps alert volume low while preserving enough evidence to route the incident to the right owner.

12.6.4.1 Statistical process control

Statistical process control originated in industrial quality control (Shewhart 1931). When adapted for ML monitoring, control charts[11] track whether metric distributions remain stable over time and complement drift-detection methods that watch for changing data distributions (Gama et al. 2014). The core idea is distinguishing common cause variation (normal fluctuation) from special cause variation (genuine anomalies).

For a metric X with established mean μ and standard deviation σ, the upper control limit is $\text{UCL} = \mu + 3\sigma$ and the lower control limit is $\text{LCL} = \mu - 3\sigma$. Points outside control limits or systematic patterns, such as 7 consecutive points above or below the mean, trigger investigation.

12.6.4.2 Fleet-wide correlation

When multiple models exhibit similar anomalies simultaneously, the root cause is likely shared infrastructure or data rather than individual model issues. Correlation analysis across models enables three operational efficiencies:

- Automatic attribution of anomalies to likely causes (deployment, data issue, infrastructure)
- Deduplication of alerts that have common causes
- Prioritization based on breadth of impact

Listing 12.5 turns correlation into a triage rule. When the fraction of anomalous models crosses the fleet threshold, the incident changes from many local debugging tasks into one shared-cause investigation.

[11] **Statistical Process Control (SPC)**: Shewhart (1931) established the control-chart framework for manufacturing quality control. Under a normal approximation, a 3-sigma control limit produces a false-alarm probability of only 0.27 percent for a stable process. For ML fleet monitoring, even small per-test false-alarm rates compound across many models and metrics, making alert-fatigue management an inherent constraint of applying SPC at platform scale.

Listing 12.5: Fleet Anomaly Attribution: Detecting correlated anomalies across a model fleet and attributing them to shared infrastructure or data causes.

```
def detect_fleet_anomaly(model_metrics, threshold=0.6):
    """
    Detect correlated anomalies across model fleet.

    Returns list of (timestamp, affected_models, likely_cause) tuples.
    """
    anomalies = []

    for timestamp in model_metrics.timestamps:
        # Identify models with anomalous metrics at this time
        anomalous_models = []
        for model in model_metrics.models:
            if is_anomalous(model_metrics[model][timestamp]):
                anomalous_models.append(model)

        # Check if anomaly fraction exceeds correlation threshold
        if (
            len(anomalous_models) / len(model_metrics.models)
            > threshold
        ):
            # Many models affected -> likely shared cause
            cause = attribute_to_shared_cause(
                timestamp, anomalous_models
            )
            anomalies.append((timestamp, anomalous_models, cause))

    return anomalies
```

The threshold prevents the monitor from overreacting to isolated model noise while still catching platform-wide failures. Above the threshold, the right owner is usually the shared dependency, such as a deployment, feature pipeline, or infrastructure service, rather than each model team independently.

12.6.4.3 Drift detection

Data drift represents gradual shifts in input distributions that degrade model performance over time. Detecting drift requires distinguishing between two fundamental types.

Covariate shift occurs when the distribution of input features $p(x)$ changes, but the relationship between inputs and outputs $p(y \mid x)$ remains constant. This is detectable in real-time by monitoring input statistics such as mean, variance, and null rates without needing labels.

Concept drift occurs when the relationship $p(y \mid x)$ changes, such as when users change their definition of spam or relevant content. This requires ground truth labels to detect, which are often delayed by minutes, days, or weeks.

Because labels are often delayed, most real-time monitoring systems use covariate shift as a leading indicator of possible performance degradation. A cheap drift score is valuable even when it is not a complete theory of robustness because it tells the operator where to investigate before outcome labels arrive.

For continuous features, the Population Stability Index (PSI)[12] quantifies distribution shift (Yurdakul and Naranjo 2020). Equation 12.12 computes PSI as the sum of log-ratio weighted differences between actual and expected bucket proportions, yielding actionable thresholds: values below 0.1 indicate stability, while values at or above 0.25 demand immediate investigation. In this chapter, PSI plays a narrow operational role as an inexpensive alerting signal.

$$\text{PSI} = \sum_{i=1}^{K_{\text{bucket}}} (A_i - E_i) \times \ln\left(\frac{A_i}{E_i}\right) \tag{12.12}$$

where A_i is the proportion in bucket i of the actual (current) distribution, E_i is the proportion in bucket i of the expected (reference) distribution, and K_{bucket} is the number of buckets.

[12] **Population Stability Index (PSI)**: Originally developed in credit scoring to detect shifts in loan-applicant populations, where regulators mandated quantitative drift monitoring. The standard thresholds (PSI < 0.1 stable, PSI ≥ 0.25 action required) were established empirically in financial services. For ML fleet monitoring, PSI's advantage is computational cheapness – a single pass over bucketed histograms – making it feasible to track hundreds of features across hundreds of models without saturating the monitoring budget.

The interpretation is deliberately coarse. PSI below 0.1 usually leaves the model in the normal monitoring path. Values between 0.1 and 0.25 create an investigation ticket. Values at or above 0.25 indicate enough shift that the platform should treat the model as potentially degraded even before labels confirm the outcome.

Napkin Math 12.9: Time-to-detection: The monitoring lag

Problem: A production model has a baseline accuracy of 95 percent. A data drift event causes accuracy to drop by 2 percent. If labeled outcomes arrive at 1000 samples/hour, how long is needed to statistically prove the model has degraded?

Math: Detection requires enough samples to distinguish the signal (the 2 percent drop) from the noise (random variance). A two-sample proportion test sets that sample count from the baseline and degraded accuracies (p_1, p_2) and the confidence and power targets ($z_{\alpha/2}$, z_β):

$$n \approx \frac{(z_{\alpha/2} + z_\beta)^2 \left[p_1(1-p_1) + p_2(1-p_2)\right]}{(p_1 - p_2)^2}$$

1. **Samples Required**: At 95 percent confidence and 80 percent power, the standard normal quantiles are $z_{\alpha/2} = 1.96$, $z_\beta = 0.84$, and plugging in gives $n \approx (1.96 + 0.84)^2 \times [0.95(1-0.95) + 0.93(1-0.93)]/(0.95-0.93)^2 \approx 2,207$ samples.
2. **Detection Latency**: 2,207 samples / 1000 samples/hour $\approx$ 2.2 hours.

Systems insight: Monitoring is not instantaneous. For a 2 percent degradation, the model operates in a broken state for approximately 2.2 hours before there is enough data to trigger an alert. This is the *monitoring lag*. Reducing this lag requires either increasing the volume of labeled data (expensive) or monitoring "proxy" metrics like PSI (faster but noisier). In a high-stakes fleet, the alert triggers on *input drift* as a leading indicator rather than waiting for accuracy to drop.

Fleet-wide drift monitoring extends that signal across the portfolio. A PSI spike in one noncritical feature may be a local investigation, but simultaneous drift across shared features or multiple dependent models points to a data pipeline failure with a larger blast radius.

12.6.5 Model-type specific monitoring

Monitoring cadence follows failure cost. Compare the model-type monitoring parameters in table 12.20: recommendation systems demand real-time CTR monitoring with 5 percent degradation thresholds, while vision classifiers tolerate daily accuracy checks with dataset-specific thresholds reflecting their lower update frequency.

Table 12.20: Model-Type Monitoring Parameters: Primary metrics, alert thresholds, and monitoring frequencies tailored to model operational requirements. Recommendation systems demand real-time CTR monitoring with 5 percent degradation thresholds, while vision classifiers tolerate daily accuracy checks at dataset-specific thresholds reflecting their lower update frequency and more stable input distributions.

Model Type	Primary Metrics	Alert Thresholds	Monitoring Frequency
Recommendation	CTR, engagement lift	5% relative drop	Real-time
Fraud Detection	Precision, recall, fraud rate	1% degradation	Real-time
LLM	Quality scores, safety metrics	Per-model calibration	Hourly
Vision	Accuracy by class	Dataset-specific	Daily
Search Ranking	NDCG, click position	2% degradation	Real-time

Recommendation monitoring is real-time because the product consequence is real-time. Click-through rate, dwell time, and conversion rate should be compared against time-matched historical baselines, control traffic when available, and the previous model version after a release. Those

fast engagement metrics are not sufficient by themselves: diversity, catalog coverage, and filter-bubble indicators protect the long-term user experience, while revenue attribution and promotional-inventory metrics connect recommendation quality to business outcomes.

Fraud monitoring binds on adversarial urgency. Missed fraud creates immediate financial loss, but excessive false positives create customer friction and manual-review load. The monitoring surface therefore has to pair detection rate, prevented dollar amount, and detection latency with false-positive rate, blocked-legitimate-transaction events, manual-review volume, and adversarial indicators such as probing patterns or sudden shifts in fraudulent behavior.

LLM monitoring binds on semantic quality, which is difficult to measure from service metrics alone. Latency, token generation rate, error rate, and safety-classifier scores establish operational health; user satisfaction signals such as thumbs-up/thumbs-down rates, regeneration rate, and task-completion proxies add delayed quality evidence. Safety metrics such as toxicity detections, refusal rate, and hallucination indicators remain partial, so they should trigger review rather than claim complete coverage.

Standard monitoring cannot detect semantic safety failures such as persuasive misinformation or skilled manipulation. Here, red teaming is a production monitoring channel rather than a full adversarial-evaluation program: human evaluators or automated probes attempt jailbreaks before deployment, and a smaller continuous probe set verifies that production safety filters remain active. Sampling outputs for delayed human review closes the loop for failures that automated metrics do not observe.

12.6.6 Observability architecture

Effective monitoring requires observability infrastructure that preserves enough context to route an alert from symptom to owner. Each evidence channel answers a different operational question. Metrics show that a service property moved outside its envelope, traces show where a request spent time, logs explain which component emitted the abnormal event, and prediction logs connect system health back to model behavior. In multi-model systems, a single user request may traverse multiple models, so distributed tracing[13], pioneered by Google's Dapper system (Sigelman et al. 2010), becomes the only reliable way to decompose end-to-end latency across inference services. Table 12.21 summarizes the architecture as an evidence map rather than as a set of disconnected telemetry tools.

13 **Distributed Tracing**: Google's 2010 Dapper paper established the pattern of propagating unique trace IDs across service boundaries. For multi-model ML pipelines, tracing is the only reliable way to attribute end-to-end latency to a specific model stage – without it, a 50 ms tail-latency spike in a 10-model pipeline requires investigating all 10 models independently. Dapper achieved this with less than 0.3 percent CPU overhead per host, setting the performance bar that makes always-on tracing feasible at fleet scale.

Table 12.21: Observability Signals by Operational Question: Observability architecture is an evidence-routing system. Metrics trigger attention, traces localize the request path, logs explain component behavior, and prediction logs connect operational symptoms to model quality.

Evidence Channel	Primary Question	What It Preserves	Typical Failure Revealed
Metrics	Which service property moved out of bounds?	Real-time streams for alerting and aggregated time series for trends	Drift, saturation, error-rate spikes, cost anomalies
Traces	Where did this request spend time?	Request IDs propagated across model invocations, timing, and resources	Cross-model latency, fan-out failures, hot stages
Logs	Which component emitted the abnormal event?	Structured events with consistent schemas and searchable indexes	New error classes, dependency failures, bad releases
Prediction logs	Did system health preserve model behavior?	Sampled inputs, outputs, features, and delayed labels	Accuracy regressions, bad cohorts, data gaps

Prediction logging is the costliest evidence channel because it records model-facing data rather than compact service counters. Production systems therefore sample it deliberately, for example logging 1 percent of predictions or logging all predictions for specific users, so the fleet retains enough examples for offline accuracy assessment, training-data generation, and failure debugging without turning monitoring into a storage workload.

12.6.7 Dashboard design

A dashboard is the human attention surface of the monitoring hierarchy. It should answer the same routing questions without forcing every reader into the same debugging console: executives need to

know whether the business is impaired, domain owners need to know which portfolio is failing, model owners need the evidence for their model, and incident responders need the trace from symptom to root cause. Table 12.22 shows that progression.

Table 12.22: Dashboard Views by Operational Question: Dashboard hierarchy should route attention from business impact to root cause. Each view reveals the amount of detail needed for the decision at that level, preventing executive views from becoming debugging consoles and investigation views from becoming portfolio summaries.

View	Question Answered	Signals to Surface
Executive	Is the business impaired?	Platform health, business impact, active incidents, key trends
Portfolio	Which domain or model family is driving the impairment?	Model inventory, portfolio metrics, recent deployments, resource utilization and cost
Model	What changed for the responsible model?	Current metrics vs. baselines, deployment history, drift indicators, cost attribution
Investigation	Why did the change occur?	Cross-model correlations, time-series overlays, log search, request-level trace detail

12.6.8 Cost monitoring and anomaly detection

Three structural properties of ML infrastructure make cost anomaly detection qualitatively harder than for general web services. First, GPU autoscaling is lumpy: adding one inference replica for a large model means provisioning an entire multi-GPU node, so a small traffic spillover triggers a step-function cost increase rather than smooth linear scaling. Second, distributed training jobs can misfire catastrophically: a misconfigured hyperparameter sweep, a DAG that does not honor spot-instance preemption, or an unbounded retry loop on a failed checkpoint can consume an entire cluster's worth of GPU-hours before any alert fires. Third, zombie models (section 12.5) continue drawing GPU memory and serving capacity even when they carry negligible traffic, contributing a chronic baseline cost that compounds across the fleet. Cost anomalies in a model platform are therefore not usually caused by traffic spikes; they are caused by training job misconfigurations, autoscaling behavior on GPU-granularity boundaries, or model lifecycle failures. Effective anomaly detection must distinguish between these causes, not just flag that costs moved.

At scale, undetected cost anomalies can accumulate millions of dollars in unexpected charges before manual review catches them. A quantitative framework for cost anomaly detection balances sensitivity against false positive rates.

12.6.8.1 Cost anomaly detection metrics

The foundation of cost monitoring is statistical anomaly detection. For a cost time series with historical mean μ and standard deviation σ, the **Z-score** quantifies how unusual a current observation is:

$$Z = \frac{C_{\text{current}} - \mu}{\sigma}$$

where C_{current} is the observed cost for the current period. A Z-score of 3 indicates the current cost is 3 standard deviations above the historical mean, an event expected less than 0.3 percent of the time under normal operations.

Complementing Z-score analysis, **percentage change** detection captures sudden shifts regardless of historical variance, which makes it useful for catching step-function increases such as a misconfigured autoscaler that doubles instance count overnight:

$$\Delta\% = \frac{C_{\text{current}} - C_{\text{previous}}}{C_{\text{previous}}} \times 100$$

12.6.8.2 Alerting thresholds and false positive analysis

Effective alerting requires calibrating thresholds to balance detection sensitivity against operational burden. Two common configurations are a **Z-score threshold**, which alerts when $|Z| > 3$ under the 3-sigma rule, and a **percentage change threshold**, which alerts when $|\Delta\%| > 50\%$ day over day. The

choice of thresholds determines false positive rates. For a normally distributed cost metric checked daily, a 3-sigma threshold produces:

$$\Pr(\text{false positive per day}) = 2 \times (1 - \Phi(3)) \approx 0.0027$$

Over a year of daily monitoring:

$$E[\text{false alerts per year}] = 365 \times 0.0027 \approx 1$$

This rate is operationally acceptable. Lowering the threshold to 2-sigma would increase annual false alerts to approximately 16, likely causing alert fatigue without meaningfully improving detection.

For percentage-based alerts, false positive rates depend on the underlying volatility of costs. Services with naturally variable demand may require higher thresholds (75 percent or 100 percent) to avoid excessive alerts, while stable baseline services can use tighter thresholds (25 percent or 30 percent).

12.6.8.3 Worked example: Detecting an inference cost spike

A recommendation service typically costs $100 per day for inference compute. The operations team receives an alert: today's cost has reached $250 by end of day.

Historical data shows mean daily cost $\mu = \$100$ with standard deviation $\sigma = \$15$, so the first check is whether the alert is statistically plausible under normal operations: $Z = \frac{\$250 - \$100}{\$15} = \frac{\$150}{\$15} = 10$.

A Z-score of 10 is extraordinarily unlikely under normal operations, but the team still confirms the billing data before treating the alert as real. After ruling out billing delays, double-counting, and pipeline errors, the investigation turns to the operating signal. Query volume is unchanged, so traffic did not cause the spike. P99 latency increased from 50 ms to 125 ms, which means each request now consumes roughly 2.5× the GPU-seconds. A new model version deployed at 2:00 AM used a larger ranking model and additional features for quality improvements, while GPU utilization stayed near 95 percent and throughput per GPU dropped 60 percent. The root cause was therefore the model update, which increased computational cost per prediction: with unchanged traffic and a 2.5× per-request cost, total cost rose from $100/day to $250/day. The platform decision is whether the quality improvement justifies the cost increase or whether optimization through quantization, smaller batches, or model distillation is required.

12.6.8.4 Root cause analysis framework

Table 12.23 groups cost anomaly root-cause categories into five classes, each with distinct investigation paths:

Table 12.23: Cost Anomaly Root Cause Categories: Five primary categories of cost anomalies with their characteristic indicators and investigation approaches. Traffic increases show proportional QPS growth, while efficiency regressions exhibit rising latency with stable traffic.

Category	Indicators	Investigation Path
Traffic increase	QPS proportional to cost	Check upstream services, marketing campaigns, viral events
Efficiency regression	Cost up, QPS unchanged, latency up	Review recent deployments, model updates, infrastructure changes
Resource leak	Gradual cost growth, utilization stable	Check for orphaned resources, failed cleanup jobs, zombie processes
Pricing change	Cost up, all metrics stable	Verify cloud provider pricing, reserved instance expiration
Configuration error	Step-function cost increase	Audit autoscaling rules, instance types, replica counts

12.6.8.5 Cost attribution by service and team

Effective cost management requires attributing costs to organizational units. Tag-based allocation assigns costs based on resource metadata, as shown in listing 12.6.

Listing 12.6: Cost Attribution Schema: Resource tagging dimensions and shared-cost distribution policies for allocating ML infrastructure expenses to teams and services.

```yaml
# Resource tagging schema for cost attribution
cost_allocation:
  dimensions:
    - team: "recommendation"      # Organizational owner
    - service: "ranking-model"    # Specific service
    - environment: "production"   # prod/staging/dev
    - model_type: "inference"     # training/inference
    - cost_center: "CC-4521"      # Finance tracking

  shared_cost_distribution:
    # Platform infrastructure costs distributed by usage
    - resource: "shared-gpu-cluster"
      method: "proportional_gpu_hours"
    - resource: "feature-store"
      method: "proportional_query_volume"
    - resource: "monitoring-infrastructure"
      method: "equal_split"
```

Shared infrastructure costs require allocation policies. Three common methods address this need:

- **Proportional allocation**: Distribute shared costs based on usage metrics (GPU-hours, storage bytes, API calls)
- **Equal split**: Divide costs equally among consuming teams (appropriate for fixed infrastructure)
- **Marginal cost**: Charge teams for the incremental cost their usage adds

The allocation policy determines whether a dashboard can assign action, not just describe spend.

12.6.8.6 Cost dashboards

Effective cost monitoring dashboards use the same attention-routing hierarchy as reliability dashboards. The executive view tracks total ML infrastructure cost, month-over-month trend, budget vs. actual, and cost efficiency metrics such as cost per prediction and cost per active user. The portfolio view then breaks cost down by domain and model type so the platform team can see whether recommendations, fraud, search, or another domain is driving the change. The service view exposes per-service cost, cost per inference, GPU utilization efficiency, and budget comparison. The investigation view adds deployment markers, operational metric correlations, and attribution detail so an anomaly can be tied to a specific change rather than treated as accounting noise.

Four key metrics govern ongoing cost monitoring:

- **Cost per inference**: The serving-unit economic metric formalized in the FinOps treatment below; track its trend here to detect efficiency regressions.
- **Cost per active user**: Infrastructure cost normalized by user base. Enables comparison across services with different scales.
- **GPU utilization efficiency**: Revenue or value generated per GPU-hour. Connects infrastructure cost to business outcomes.
- **Budget burn rate**: Current spending velocity relative to allocated budget. Enables proactive intervention before overruns.

Integrating cost monitoring with the hierarchical monitoring architecture ensures that cost anomalies receive appropriate attention alongside performance and quality metrics. Yet building these CI/CD pipelines and hierarchical monitoring systems for every individual product team is prohibitively expensive. To make these capabilities universally available without duplicating effort, organizations must elevate them into a unified internal product, a discipline known as platform engineering.

12.7 Platform Engineering

Organizations where every data science team independently provisions GPU clusters, configures model registries, and wires up alerting dashboards pay a massive duplication tax on undifferentiated

infrastructure work. Platform engineering solves this by treating the ML infrastructure itself as a product, providing paved roads that allow product teams to focus entirely on modeling rather than infrastructure plumbing.

Platform engineering for machine learning creates shared infrastructure that enables model teams to develop, deploy, and operate models without managing underlying complexity. Effective platforms balance self-service capabilities that accelerate development against governance requirements that ensure consistency and reliability.

12.7.1 Abstraction levels

The abstraction-level decision is how much operational judgment the platform should centralize on behalf of model teams. Too little abstraction leaves every team paying the same infrastructure tax; too much hides the controls that unusual workloads need. The four levels in table 12.24 sit on that flexibility-versus-convenience curve, distinguished by which concerns the platform owns and which it leaves to the model team.

Table 12.24: Platform abstraction levels: Platform engineering centralizes different parts of the ML lifecycle at each abstraction level. The right level depends on whether an organization values workload-specific control or shared operational invariants more.

Level	Platform owns	Model team still owns	Best fit and risk
Level 1	GPU capacity, storage volumes, network connectivity, and basic orchestration such as Kubernetes namespaces	Training code, serving stack, monitoring, and deployment path	Fits unusual workloads with strong infrastructure teams; becomes costly when many teams repeat the same operational work
Level 2	Standardized containers, ML-aware Kubernetes integration, persistent volumes, and basic service-mesh support	Experiment structure, training jobs, serving releases, and monitoring workflows	Reduces packaging duplication; operational correctness still depends heavily on each team
Level 3	ML-specific control loops such as topology-aware scheduling, hyperparameter tuning, distributed training, serving	Modeling judgment and workload-specific trade-offs	Absorbs failure modes that recur across teams; systems such as Kubeflow and Ray often sit near this level
Level 4	Full lifecycle product: IDEs, feature stores, experiment tracking, registries, CI/CD, monitoring, cost, governance	Higher-level product and modeling intent, with less direct low-level control	Maximizes speed and consistency; the platform team accepts responsibility for policy, reliability, and economic visibility

Managed and lifecycle platforms such as Vertex AI, SageMaker, TFX[14] at Google (Baylor et al. 2017), and MLflow (Zaharia et al. 2018) represent the full-platform end of this spectrum. The trade-off is explicit: teams gain speed and consistency by giving up some low-level control, while the platform team accepts responsibility for policy, reliability, and economic visibility.

12.7.2 Self-service model deployment

Self-service deployment works only when the platform can let model teams move quickly while keeping risky choices inside governed boundaries. The design question is therefore which deployment decisions should remain team-owned and which invariants the platform must enforce before a model receives production traffic.

12.7.2.1 Deployment API design

A well-designed deployment API abstracts operational complexity by making the model team's intent declarative. The platform lesson is not the shape of a particular YAML file; it is which invariants the platform must capture before it turns a model version into fleet traffic. The invariants in table 12.25 are the platform contract behind that declarative interface. Prometheus[15] is one common operational sink for the telemetry side of that contract.

14 **TensorFlow Extended (TFX)**: Google's production ML platform. TFX's key architectural insight is that data validation via TensorFlow Data Validation (TFDV) gates the pipeline before training begins, catching schema violations and distribution drift that would otherwise produce silently degraded models. This "fail before training" philosophy prevents the most expensive class of ML waste: GPU-hours spent training on corrupted data.

15 **Prometheus**: An open-source monitoring system now maintained under the CNCF umbrella, using a pull-based model to scrape metrics from exporters. For ML fleets, Prometheus is the standard for tracking operational health (CPU/GPU utilization, request rates), but its time-series model is poorly suited for tracking high-dimensional distribution drift, necessitating a two-tier monitoring architecture.

Table 12.25: Self-Service Deployment Invariants: A deployment API should encode the controls that keep local team autonomy from creating fleet-wide risk. The concrete syntax can vary, but the invariants must survive across platforms.

Declared intent	Platform invariant
Model artifact and version	The serving system loads exactly the validated model from the registry.
Resource envelope	GPU, memory, and replica requests stay inside quota and scheduling constraints.
Traffic policy	Canary, shadow, and rollback controls bound blast radius before full promotion.
Quality gates	Error, latency, drift, and task-quality thresholds block unsafe promotion.
Telemetry sink	Operational metrics flow into operational monitoring, while model-quality signals flow into drift and slice monitors.
Approval boundary	Sensitive data access, budget overruns, and high-risk release windows require review.

TensorFlow Serving provides model-serving lifecycle capabilities for loading, versioning, canarying, and rolling back models (Olston et al. 2017). A self-service platform adds the policy layer around those capabilities: model teams specify intent, and the platform translates it into scheduling, traffic routing, telemetry, and approval controls.

12.7.3 Resource management

Efficient resource utilization is essential for platform economics because shared capacity pays off only when training and serving workloads receive different controls. Training can trade time for utilization; serving trades capacity for tail-latency protection, so a single scheduler policy cannot optimize both.

12.7.3.1 Training resource management

Training workloads are batch-oriented, so their flexibility can be converted into higher utilization. Jobs have defined start and end times, GPU memory requirements are often known in advance, and many runs can be preempted and restarted if checkpointing limits lost work. A training scheduler uses priority queues, fair sharing, and deadline-aware placement to decide which jobs should wait, which should move, and which low-priority jobs can be preempted[16] for urgent work. Spot/preemptible instances fit this control loop because automatic retry on preemption can preserve progress while reducing cost.

[16] **Job Preemption**: The ability to terminate lower-priority workloads to free resources for urgent work. Cloud providers offer 60–90 percent discounts on preemptible/spot instances, but the trade-off is that ML training must support checkpoint-and-resume: without periodic checkpointing, a preempted 72-hour training run restarts from scratch, converting a 70 percent cost savings into a net loss. Checkpoint frequency itself is a trade-off between I/O overhead and wasted-compute risk.

12.7.3.2 Serving resource management

Serving workloads bind on latency rather than batch flexibility. Demand fluctuates with time of day, events, and seasonality, but a live request cannot be preempted without user impact. The serving control loop therefore starts from the latency SLO and asks how much capacity must be online before requests arrive.

Autoscaling adds horizontal capacity from request-rate, latency, or model-specific queue metrics, but it must account for model load time and GPU memory granularity. Resource isolation prevents one model from consuming another model's budget, while cost optimization combines right-sized instances, reserved capacity for baseline demand, and spot instances only where overflow can tolerate interruption. The operational goal is not maximum utilization in isolation; it is the cheapest capacity plan that still protects tail latency.

12.7.3.3 Platform utilization metrics

Once training and serving controls are separated, the shared platform still needs an aggregate measure of whether capacity is being converted into useful work. Equation 12.13 defines platform efficiency as the capacity-weighted average utilization across all resources:

$$U_{\text{platform}} = \frac{\sum_i U_i \times \text{Cap}_i}{\sum_i \text{Cap}_i} \tag{12.13}$$

where U_i is the utilization of resource i and Cap_i is the capacity of resource i. Weighting by capacity keeps the metric honest about where stranded work lives: an idle single GPU and an

idle eight-GPU node are not equally wasteful, so a small underused resource should not drag the platform average down as hard as a large idle pool.

However, raw utilization is incomplete. Effective utilization must also consider utilization quality to determine whether GPUs are doing productive work or waiting on data, utilization fairness to assess whether utilization is distributed appropriately across teams, and utilization cost to evaluate efficiency in terms of cost per unit of ML output.

12.7.3.4 Worked example: GPU cluster efficiency

A platform operates a 100-GPU training cluster with average GPU utilization of 65 percent, memory utilization of 80 percent, four-hour average queue waits, and a cost of $2.50 per GPU-hour. The high memory utilization suggests jobs are generally sized correctly, while the lower compute utilization points to I/O-bound training steps. Queue waits show that demand already exceeds supply, so the platform decision is not simply whether to buy more GPUs. Daily cluster spend is fixed by provisioned capacity at $100 \times 24 \times \$2.50 = \$6,000/day$, but only $100 \times 24 \times 0.65 \times \$2.50 = \$3,900/day$ is currently converted into useful GPU work. If data-loading optimization raises compute utilization to 80 percent, useful work rises to $100 \times 24 \times 0.80 \times \$2.50 = \$4,800/day$ before adding any capacity. The same measurements therefore support two different actions: remove the I/O bottleneck to recover stranded capacity, then use scheduling or expansion only for the remaining queue pressure.

Checkpoint 12.5: The annual cost of GPU cluster underutilization

Use daily cluster spend ($6,000/day) and productive GPU-hours ($3,900/day at 65 percent utilization) to extend the analysis to annual scale and compare against the platform ROI argument.

- ☐ **Underutilization cost**: Calculate the annual cost of underutilization for the 100-GPU cluster at 65 percent utilization, the cost after optimization to 80 percent, and the annual savings from the data-loading optimization.
- ☐ **Queue dynamics**: Using Little's Law $L = \lambda W$, calculate the arrival rate for a 4-hour average wait with 100 jobs in queue and infer whether expansion or utilization optimization recovers more annual value.
- ☐ **Savings vs. platform**: Compare the annual savings from cluster optimization to the $2M/year platform investment scenario and determine the fleet size where cluster-efficiency optimization alone justifies the platform investment, holding all other per-model savings constant.

12.7.4 Advanced fleet metrics: ML productivity goodput (MPG)

While utilization metrics capture resource busyness, they often fail to reflect true engineering value. A GPU spinning at 100 percent utilization on a hyperparameter tuning job that eventually fails due to a configuration error is "efficient" in terms of hardware but "wasteful" in terms of productivity. To address this, ML Productivity Goodput (MPG) provides a comprehensive metric for fleet efficiency (Wongpanich et al. 2025).

MPG extends the fleet law introduced in section 1.5.2 into the **Iron Law of Machine Learning Fleet Efficiency**. Just as the classic Iron Law of Processor Performance (Hennessy and Patterson 2011) decomposes CPU execution time, this formulation decomposes ML fleet efficiency into three orthogonal components:

$$\text{MPG} = \text{Scheduling Efficiency} \times \text{Runtime Efficiency} \times \text{Program Efficiency}$$

- **Scheduling efficiency**: This measures the platform's ability to place jobs on available resources. It penalizes queuing delays and fragmentation where resources exist but cannot be assigned.
- **Runtime efficiency**: This captures the hardware utilization quality during execution. It penalizes "bad put" such as restart overheads from preemption, idle time due to data loading bottlenecks, and straggler effects in distributed training.

- **Program efficiency**: This assesses whether the code running on the hardware is optimized. It penalizes suboptimal kernels, excessive precision (FP32 where BF16 suffices), and redundant computations.

By tracking the iron law components, platform teams move beyond simple "utilization" to measuring "goodput"—the rate at which valid, useful model training work is completed. This shift often reveals that high-utilization clusters may have low MPG due to frequent failures or inefficient code, guiding optimization efforts toward the highest-impact bottlenecks.

12.7.5 Multi-tenancy and isolation

Enterprise platforms earn their sharing dividend only if multiple teams can share infrastructure without sharing failures. Multi-tenancy is therefore an isolation problem before it is an efficiency problem: the platform must bound performance interference, data exposure, and cost spillover at the same time.

12.7.5.1 Isolation requirements

Tenants need isolation at every boundary where one team's workload, data, or cost can affect another team:

- **Performance isolation**: Resource limits, scheduling fairness, and network quality of service prevent one workload from degrading another during training spikes or serving bursts.
- **Security isolation**: Access controls, network segmentation, and encryption keep teams with different sensitivity levels from sharing data paths accidentally.
- **Cost isolation**: Metering and chargeback make each team's usage attributable.

Together, the three boundaries make shared infrastructure safe enough for teams to use without negotiating every deployment by hand.

12.7.5.2 Namespace architecture

A typical multi-tenant architecture uses hierarchical namespaces:

```
Platform
   Team A
      Development
      Staging
      Production
   Team B
      Development
      Staging
      Production
   Shared
       Feature Store
       Model Registry
       Monitoring
```

Each team receives dedicated namespaces with resource quotas, while shared services operate in common namespaces with appropriate access controls. Without controls, one team's demanding workload can degrade performance for others, a failure mode known as the noisy neighbor problem[17]. Prevention requires controls at request, rate, priority, and budget boundaries.

Four controls prevent one tenant from overwhelming shared services:

- **Request limits**: Each request has a cap on the resources it can consume.
- **Rate limiting**: Each tenant has bounded request rates so shared services are not overwhelmed.
- **Priority classes**: Critical workloads receive resources even under contention.
- **Burst budgets**: Temporary resource overages are allowed while long-term fairness is preserved.

Together, these controls make fairness enforceable in the request path instead of relying on after-the-fact negotiation.

[17] **Noisy Neighbor Problem**: A multi-tenancy failure mode where one tenant's workload degrades performance for co-located tenants. ML workloads are particularly prone to this because training jobs allocate GPU memory greedily – a single job requesting all available HBM on a shared node can starve co-located inference services, causing latency spikes that violate SLOs. Unlike CPU workloads, GPU memory cannot be overcommitted, making physical resource isolation the only reliable prevention.

12.7.6 FinOps for ML platforms

Financial operations (FinOps) practices in traditional IT rarely account for the cost shape of ML workloads. GPU compute costs dominate ML budgets, with a single training run potentially costing tens of thousands of dollars. Serving infrastructure scales with traffic, creating variable costs that can swing by an order of magnitude between peak and trough. The experimental nature of ML development means many training runs produce no production value. Effective FinOps for ML requires specialized practices that account for these realities.

Definition 12.1: FinOps for ML

FinOps for ML is the practice of treating compute cost as a first-class engineering constraint (measured in real-time per experiment and model, and optimized jointly with model accuracy and latency) rather than accounting for it retrospectively through annual budget reconciliation.

1. **Significance**: ML compute costs scale steeply with experimentation volume. A team running 1,000 GPU-hours/day at $3/GPU-hour spends $3,000/day ($1.1M/year) on training alone. With per-experiment cost visibility, an engineer can identify that 30 percent of runs are terminated early due to divergence (recoverable with better hyperparameter selection) and that spot instances provide 60–70 percent cost reduction for fault-tolerant workloads, reducing effective spend by 40–50 percent without changing output quality.
2. **Distinction**: Unlike traditional IT budgeting (which allocates fixed annual compute budgets across departments), FinOps for ML operates at the per-run and per-model level with real-time feedback—enabling decisions like early stopping a $50K training run at step 1,000 when loss curves signal divergence, rather than discovering the waste after the full run completes.
3. **Common pitfall**: A frequent misconception is that FinOps means minimizing compute spend. The goal is maximizing accuracy-per-dollar across the full lifecycle: a $500K training run producing a model serving 10B inferences at $0.0001/query may be far more cost-efficient than a $50K run producing a model that requires 10× the inference compute to achieve the same accuracy at scale.

12.7.6.1 Cost components

ML platform cost breakdown spans multiple categories with different optimization strategies. Table 12.26 reveals that training compute dominates (40–60 percent of costs), driven by GPU hours and experiment volume, with spot instances and early stopping as primary optimization levers:

Table 12.26: ML Platform Cost Breakdown: Five cost categories with typical budget share and optimization levers. Training compute dominates (40–60 percent) driven by GPU hours and experiment volume; spot instances and early stopping provide primary savings. Serving compute (20–40 percent) scales with traffic; autoscaling and model optimization reduce costs while maintaining latency SLOs.

Cost Category	Typical Share	Primary Drivers	Optimization Lever
Training compute	40–60%	GPU hours, experiment volume	Spot instances, early stopping
Serving compute	20–40%	Traffic volume, latency SLOs	Autoscaling, model optimization
Storage	10–20%	Dataset size, checkpoint frequency	Tiered storage, retention policies
Network	5–15%	Multi-region, data transfer	Caching, compression
Platform overhead	5–10%	Team size, tooling	Automation, self-service

12.7.7 Cost optimization strategies

Cost optimization is a risk-allocation decision, not a catalog of discounts. Interruptible compute pays off only when checkpointing bounds the work lost to preemption, serving autoscaling only within the latency budget, and instance right-sizing only against measured cost per step or per inference. Each lever below trades a specific risk for a specific saving.

12.7.7.1 Spot and preemptible instances

Cloud providers offer significant discounts (60–90 percent) for interruptible compute capacity. ML training workloads are well suited for spot instances because checkpointing enables recovery from interruptions, training jobs tolerate delays better than serving, and large batch jobs amortize instance acquisition overhead.

Effective spot usage requires three practices:

- **Checkpoint frequency tuning**: Balance checkpoint overhead against potential lost work. For a job costing $10/hour on spot instances, hourly checkpoints losing at most one hour of work ($10) far outweigh checkpoint storage costs.
- **Instance diversification**: Request capacity across multiple instance types and availability zones to reduce interruption probability.
- **Fallback strategies**: Automatically fall back to on-demand instances for time-sensitive jobs or when spot availability is low.

Together, these practices turn a cheap but unreliable resource into a bounded-risk training option.

```
Training Cost Comparison (100 GPU-hours):
  On-demand:     100 × $3.00 = $300
  Spot (70% discount): 100 × $0.90 = $90 (+ potential reruns)
  Reserved (40% discount): 100 × $1.80 = $180 (requires commitment)
  Actual spot with interruptions: ~$110 (accounting for 20% rerun overhead)
```

The serving side allocates risk differently. Serving costs scale with traffic, but ML serving cannot autoscale like a stateless web application. Large models may take seconds or minutes to load, so purely reactive scaling arrives after the latency spike; the platform needs predictive scale-up before anticipated demand.

The capacity steps are also lumpy. Accelerator memory makes serving scale in jumps of 0, 1, 2, or 4 accelerators rather than smooth CPU fractions, and batching adds another coupling because waiting for a larger batch improves utilization but consumes latency budget. An autoscaling policy must therefore choose the least expensive capacity step that still protects the p99 latency SLO.

Instance selection then starts from the binding constraint rather than the newest instance type. Memory-bound training with large embedding tables prioritizes GPU memory capacity and bandwidth. Compute-bound dense training maximizes sustained FLOP/s per dollar.

Serving splits the decision again. Latency-sensitive serving minimizes cold start and single-request latency, while throughput-oriented serving maximizes requests per dollar through batching. Instance selection should therefore be benchmark-driven: compare cost per training step or cost per inference across instance types instead of assuming larger hardware is automatically more economical.

12.7.8 Cost visibility and attribution

Cost optimization requires granular visibility into spending because teams cannot optimize costs they cannot see or own. Attribution policy is an incentive design problem, not only an accounting problem. Direct metering charges teams exactly for resources consumed; it is the most accurate model but can encourage under-provisioning when teams optimize their bill instead of service health. Allocation-based attribution charges based on reserved capacity rather than actual usage, making budgets predictable while potentially hiding waste. Hybrid attribution combines a base charge for allocation with a variable charge for excess usage, balancing predictability with efficiency incentives.

For serving workloads, cost per inference provides the key unit economic metric. Equation 12.14 expresses this as total serving cost divided by inference count, enabling direct comparison of model efficiency and capacity planning:

$$\text{Cost per inference} = \frac{\text{Total serving cost}}{\text{Total inferences served}} \tag{12.14}$$

Cost per inference then becomes the bridge from infrastructure telemetry to product decisions. Teams can compare model versions by asking whether an accuracy gain justifies a 2× inference cost,

evaluate optimization work by measuring whether quantization reduced cost per inference by 40 percent, and forecast monthly serving cost under projected traffic. Tracking the metric by model, customer segment, and request type reveals which workload actually deserves optimization effort.

Effective chargeback closes the loop by making those costs actionable. It requires fine-grained resource metering, attribution rules that map resources to teams, dashboards that show cost by team, project, and model, forecasting tools that help teams plan budgets, and anomaly detection for unexpected cost increases.

12.7.9 Budget-aware development

FinOps extends beyond infrastructure optimization because cost feedback changes which experiments teams run and which models they ship. Unconstrained experimentation leads to runaway costs, but budget controls should make trade-offs visible before they block useful work: per-experiment limits cap individual training runs at a cost threshold, team budgets allocate monthly compute budgets with visibility into consumption, and approval workflows require review for experiments that exceed cost thresholds. These controls should inform rather than block. The goal is cost awareness, not prevention of valuable experiments.

12.7.9.1 Cost-quality trade-offs

Model selection should explicitly consider cost alongside accuracy. Table 12.27 illustrates cost-quality trade-off analysis: moving from small to medium model yields a 3 percentage-point accuracy gain for 10× training cost increase, while medium to large yields only 1 additional percentage point for another 10× cost, a pattern that should inform deployment decisions:

Table 12.27: Cost-Quality Tradeoff Analysis: Diminishing returns in model scaling. Small-to-medium model transition yields a 3 percentage-point accuracy gain for 10× cost increase; medium-to-large yields only 1 additional percentage point for another 10× cost. This pattern demonstrates why explicit cost-quality analysis should inform model selection rather than defaulting to larger architectures.

Model	Accuracy	Training Cost	Serving Cost/1K	Value Judgment
Small	92%	$500	$0.10	Baseline
Medium	95%	$5,000	$0.50	3 percentage points for 10× cost
Large	96%	$50,000	$2.00	Additional 1 percentage point for 10× more

For many applications, the marginal accuracy gain does not justify the cost increase. Making these trade-offs explicit prevents defaulting to the largest available model.

Efficiency metrics should be reviewed alongside model quality so the platform can identify waste that accuracy alone hides. Table 12.28 connects each metric to the operational question it answers.

Table 12.28: Efficiency Metric Action Map: Efficiency metrics are useful only when they identify a decision. The platform reviews these signals alongside quality metrics to find waste that accuracy alone hides.

Metric	Waste Signal	Typical Action
Cost per accuracy point	Accuracy gains are bought with disproportionate spend	Revisit model size, feature set, or serving target before scaling up.
Experiments per production model	Many runs produce little deployable value	Improve experiment design, stopping rules, and promotion criteria.
GPU utilization	Capacity is reserved but not doing useful work	Right-size instances, improve batching, or profile inefficient code.
Spot utilization rate	Eligible workloads use expensive on-demand capacity	Move fault-tolerant jobs to spot instances with checkpoint support.

Regular review of these metrics identifies systemic inefficiencies and guides platform improvements. The same visibility must extend to the knowledge layer of modern ML applications. Retrieval, fine-tuning, and total lifecycle ownership all look like model-quality choices from a notebook, but at fleet scale they become cost-placement decisions.

12.7.10 Retrieval-augmented generation vs. fine-tuning: The knowledge operations trade-off

Domain knowledge can live in a retrievable index, in model weights, or in adapter weights, and each placement creates a different operating cost surface. Retrieval-augmented generation (RAG) retrieves documents at inference time and places them in the prompt (Lewis et al. 2020). Fine-tuning stores adaptation in model weights or adapter weights: supervised fine-tuning (SFT) trains on curated input-output examples, while Low-Rank Adaptation (LoRA) stores small adapter-weight deltas (Hu et al. 2021). Both approaches can improve task behavior, but they move cost and failure risk to different places in the fleet.

RAG chooses freshness and attribution over lower-cost inference. Updating a document index can change the system's answers without retraining the model, and retrieved passages provide a concrete source trail for generated claims. The cost moves into the serving path: each query may carry a large payload of retrieved context, such as 10,000 extra tokens, increasing prefill attention work roughly quadratically with sequence length while growing the KV cache linearly with context length. FlashAttention reduces memory traffic and intermediate storage, but it does not remove the attention-compute or KV-cache footprint created by long context. Long contexts also introduce a quality risk because the model may underuse relevant evidence when the prompt contains too much retrieved material.

Fine-tuning chooses compact inference over fast knowledge updates. The platform pays for data curation, validation, training, and evaluation before deployment, but successful adaptation can shorten prompts and reduce per-query serving cost. Adapter-based fine-tuning changes the fleet problem rather than eliminating it: thousands of customer-specific LoRA adapters create a multi-tenant serving problem, while outdated or incorrect facts embedded in weights are harder to remove than documents in an index.

The decision boundary is where knowledge volatility, attribution requirements, latency budget, and adapter-management complexity cross. RAG fits volatile facts, strong attribution requirements, and moderate query volume because the index can change faster than a training pipeline. Fine-tuning fits specialized domain language, strict latency budgets, and stable behavior requirements because the model carries the adaptation without paying for 10,000 context tokens on every query. Many mature platforms use both: fine-tuning teaches the model how to use the organization's tools and document style, while RAG supplies the facts that must remain fresh.

12.7.11 ML systems TCO framework

While FinOps practices provide operational cost visibility, strategic ML investment decisions require a comprehensive **TCO framework** that captures the complete economic picture across the ML lifecycle. Section 12.3 established the hardware total cost of ownership: the CapEx and OpEx of the physical fleet. The hardware lens now widens to the full ML lifecycle, where hardware underlies the training and inference cost terms and data and iteration costs complete the four-component model. Unlike traditional software where infrastructure costs dominate, ML systems exhibit a distinctive four-component cost structure that evolves differently with scale.

Definition 12.2: ML systems TCO

Total Cost of Ownership (TCO) for ML Systems is the complete economic accounting of developing, deploying, and operating machine learning capabilities across their full lifecycle.

1. **Significance**: It captures the Cost Inversion of scale: while training costs are a one-time upfront operation, the cumulative Inference TCO grows linearly with user adoption and time, often exceeding development costs by 5× to 10× over a 3-year period.
2. **Distinction**: Unlike traditional IT TCO (where hardware CapEx dominates), ML TCO is uniquely shaped by operating efficiency (η_{hw}), meaning the ability to serve predictions at the lowest possible energy and compute cost per query.

3. **Common pitfall**: A frequent misconception is that the initial training bill is the primary financial risk. In reality, the data debt and maintenance overhead are the "silent interest rates" that can make an accurate model economically unsustainable in production.

12.7.11.1 The TCO equation

Equation 12.15 expresses the total cost of ownership as the sum of four distinct cost components, each with different scaling characteristics and optimization levers:

$$\text{TCO}_{\text{ML}} = C_{\text{train}} + C_{\text{infer}} + C_{\text{data}} + C_{\text{iter}} \tag{12.15}$$

Here, C_{train} is the one-time training and evaluation cost, C_{infer} is recurring serving cost, C_{data} covers data acquisition, storage, cleaning, and governance, and C_{iter} captures iteration costs such as retraining, validation, incident response, and maintenance.

As figure 12.10 illustrates, while GPU compute and storage are the visible costs, hidden operational costs often constitute fully half of the actual budget.

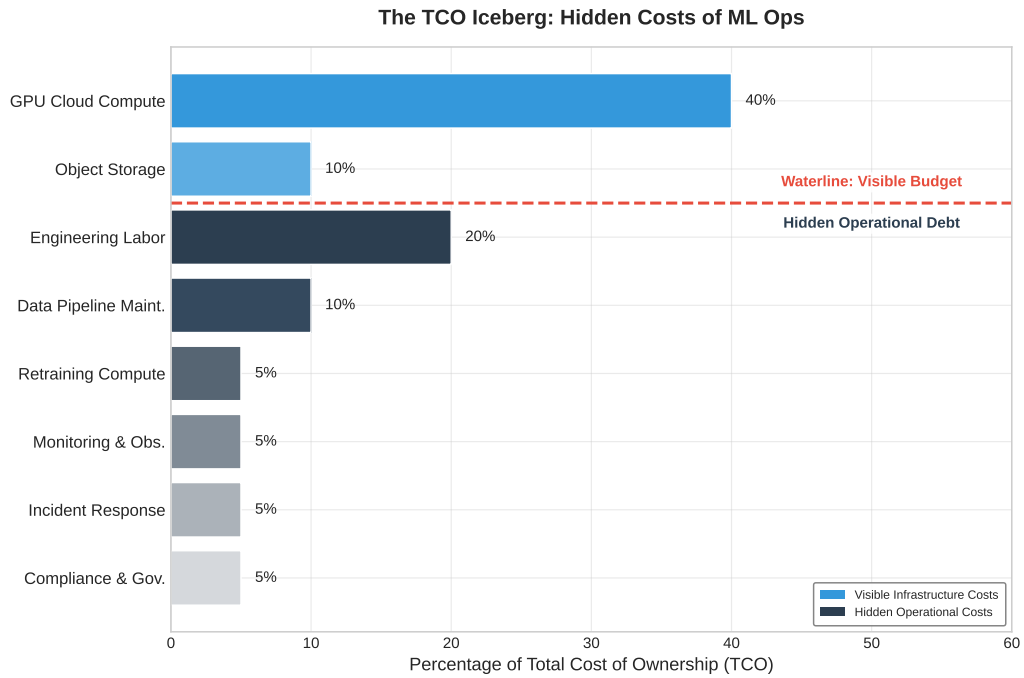


Figure 12.10: The TCO Iceberg: Total Cost of Ownership analysis for ML systems. While GPU compute and storage are the visible costs, the hidden operational costs—including engineering labor, maintenance, and compliance—often constitute fully half of the actual budget.

The decomposition reveals a critical insight: the dominant cost component shifts as organizations mature. Early-stage ML efforts are dominated by C_{iter} (experimentation), growth-stage by C_{train} (model development), and production-scale by C_{infer} (serving at volume). Optimization strategies must match the current cost structure.

12.7.11.2 Training cost model

Training costs encompass the compute required to develop and maintain models. Equation 12.16 formalizes training cost as a function of GPU count, training duration, non-energy accelerator-hour rates, data center energy overhead, and failure overhead:

$$C_{\text{train}} = N_{\text{GPU}} \times T_{\text{hours}} \times \left(R_{\text{GPU/hr}} + P_{\text{GPU}} \times R_{\$/\text{kWh}} \times \text{PUE}\right) \times (1 + F_{\text{fail}}) \tag{12.16}$$

where:

- N_{GPU} is the number of GPUs allocated
- T_{hours} is the training duration in hours
- $R_{\text{GPU/hr}}$ is the non-energy accelerator-hour cost
- P_{GPU} is per-GPU power draw in kW
- $R_{\$/\text{kWh}}$ is the electricity price
- PUE is the Power Usage Effectiveness multiplier applied to the energy term
- F_{fail} is the failure overhead factor (fraction of training lost to failures and restarts)

The PUE factor captures data center efficiency: a PUE of 1.2 means 20 percent additional power for cooling and infrastructure. It applies to energy consumption, not to the whole accelerator-hour rate. The failure overhead F_{fail} accounts for checkpoint-and-restart costs; at scale, Chapter 7 established how component failures translate into checkpointing and restart overhead for long-running fleet jobs.

12.7.11.3 Worked example: Training cost calculation

Consider training a large language model:

- Configuration: 256 H100 GPUs for 14 days
- Base accelerator-hour rate: \$3.50/GPU-hour
- Energy adder: 700 W at \$0.07/kWh and PUE 1.15 adds \$0.06/GPU-hour
- Effective rate: \$3.56/GPU-hour
- Failure overhead: 15 percent (typical for multi-node training)

$C_{\text{train}} = 256 \times (14 \times 24) \times (\$3.50 + \$0.06) \times 1.15 \approx \$352,150$

If this model requires quarterly retraining, annual training cost reaches approximately \$1.4M. However, training cost often represents a small fraction of total TCO for production systems serving millions of users.

12.7.11.4 Inference cost model

Inference costs dominate TCO for production ML systems at scale. Equation 12.17 expresses serving cost as a function of query volume, latency requirements, and utilization efficiency:



At production scale, serving (infer) dominates total cost of ownership.

$$C_{\text{infer}} = \frac{Q_{\text{daily}} \times T_{\text{infer,avg}}}{U_{\text{GPU}} \times B_{\text{eff}}} \times R_{\text{GPU/hr}} \times 365 \tag{12.17}$$

where:

- Q_{daily} is the daily query volume
- $T_{\text{infer,avg}}$ is the average latency per inference (in hours, so the numerator carries units of GPU-hours per day)
- U_{GPU} is the effective GPU utilization (typically 0.4-0.8)
- B_{eff} is the effective batch size (throughput multiplier from batching)
- $R_{\text{GPU/hr}}$ is the hourly GPU cost
- The $\times 365$ factor lifts daily GPU-hours into an annualized cost. The 24-hours-per-day factor is already absorbed into $Q_{\text{daily}} \times T_{\text{infer,avg}}$ and must not be applied a second time.

The key insight is that inference cost scales linearly with query volume but sublinearly with optimization: improving utilization from 40 percent to 80 percent halves infrastructure requirements, and effective batching ($B_{\text{eff}} > 1$) further reduces per-query costs.

12.7.11.5 Alternative formulation

For capacity planning, express inference cost in terms of required GPU-seconds per query:

$$C_{\text{infer}} = Q_{\text{annual}} \times \frac{T_{\text{infer}}}{U_{\text{GPU}}} \times \frac{R_{\text{GPU/hr}}}{3600}$$

where T_{infer} is the inference time in seconds. This formulation directly connects latency optimization to cost reduction.

12.7.11.6 Data cost model

Data costs exhibit superlinear scaling with user base due to storage growth, transfer volume, and processing requirements. Equation 12.18 decomposes data cost into three components that scale differently: storage cost C_{storage}, egress cost C_{egress} for moving data out over the network, and processing cost C_{process}:

$$C_{\text{data}} = C_{\text{storage}} + C_{\text{egress}} + C_{\text{process}} \tag{12.18}$$

Storage costs grow with data retention requirements:

$$C_{\text{storage}} = D_{\text{vol,data}} \times R_{\text{storage/GB}} \times T_{\text{retention}}$$

where $D_{\text{vol,data}}$ is data volume in GB, $R_{\text{storage/GB}}$ is the monthly storage rate, and $T_{\text{retention}}$ is retention period in months.

Egress costs scale with data transfer volume:

$$C_{\text{egress}} = D_{\text{vol,transfer}} \times R_{\text{egress/GB}}$$

Cloud egress pricing ($0.08-0.12 per gigabyte) makes data transfer a significant cost driver for multi-region deployments and training data distribution.

Processing costs scale with compute requirements for extract, transform, load (ETL), feature engineering, and data validation:

$$C_{\text{process}} = D_{\text{vol,processed}} \times R_{\text{process/GB}}$$

Data processing costs often surprise organizations: a feature engineering pipeline processing 10 terabytes daily at $0.02 per gigabyte costs $73,000 annually.

12.7.11.7 Iteration cost model

Development costs capture the engineering investment in experimentation and model improvement. Equation 12.19 formalizes this as the product of experiment count, experiment duration, and combined engineering and compute costs:

$$C_{\text{iter}} = N_{\text{exp}} \times T_{\text{exp}} \times (C_{\text{engineer}} + C_{\text{compute}}) \tag{12.19}$$

where:

- N_{exp} is the number of experiments conducted
- T_{exp} is the average experiment duration
- C_{engineer} is the engineering cost per experiment (time allocation)
- C_{compute} is the compute cost per experiment

A critical but often overlooked factor: failed experiments have real cost. If 90 percent of experiments do not improve production metrics, the effective cost per successful experiment is $10\times$ the nominal experiment cost. This motivates investment in experiment infrastructure that reduces T_{exp} and improves experiment success rates.

12.7.11.8 Worked example: Startup vs. production company TCO

Table 12.29 illustrates how cost structure evolves with scale by comparing two organizations. The startup operates 1 production model serving 100,000 daily users with monthly retraining, a 2-engineer team, and cloud-native infrastructure. The production company operates 50 models serving 10 million daily users with weekly retraining for high-velocity models, a dedicated 15-engineer ML platform team, and hybrid cloud/on-premise infrastructure.

Table 12.29: TCO Comparison: Startup vs. Production Company: Cost structure shifts dramatically with scale. Startups are dominated by iteration costs (engineering salaries for experimentation), while production companies see inference costs dominate as serving volume grows. The 100× user increase yields only 20.6× TCO increase due to optimization effects, but note the superlinear 160× scaling in data costs.

Cost Component	Startup	Production Company	Scaling Factor
Training	$5,000/month	$150,000/month	30× (more models, larger)
Inference	$2,000/month	$400,000/month	200× (100× users, optimization)
Data	$500/month	$80,000/month	160× (superlinear with users)
Iteration	$40,000/month	$350,000/month	8.75× (team size, experiments)
Total TCO	**$47,500/month**	**$980,000/month**	**20.6×**
Dominant Cost	Iteration (84%)	Inference (41%)	

The comparison reveals a structural shift in ML economics. Startups spend 84 percent on iteration through people and experimentation, while production companies spend 41 percent on inference infrastructure, so each stage demands different optimization strategies. TCO also scales sublinearly: a 100× user increase yields only 20.6× TCO because inference economies of scale and amortized training costs absorb part of the growth. Data breaks that pattern by scaling 160× for 100× users as storage requirements grow with user history and processing costs rise with feature complexity. Platform automation produces the final shift: the production company runs 10× more experiments, but iteration cost grows only 8.75× because per-experiment overhead falls.

12.7.11.9 TCO sensitivity analysis

Understanding how TCO responds to key parameters enables strategic planning. Table 12.30 shows the impact of 2× increases in key parameters on each cost component:

Table 12.30: TCO Sensitivity Analysis: Impact of 2× increase in key parameters on cost components. Model count has the highest total impact (85 percent) because it affects all components, while retraining frequency has the lowest (25 percent) as it affects only training and associated data costs. User growth shows superlinear data cost impact (120 percent) due to storage and processing requirements scaling faster than user count.

Parameter	Change	Training	Inference	Data	Total Impact
Daily users	2×	–	100%	120%	45%
Model count	2×	100%	100%	50%	85%
Queries per user	2×	–	100%	80%	40%
Model size	2×	150%	120%	30%	55%
Retraining freq.	2×	100%	–	40%	25%

The nonlinear effects explain why the total-impact column does not simply mirror the input change. User scaling makes data costs grow superlinearly (120 percent for 2× users) because user history accumulation and cross-user feature computation add state faster than headcount. Model size scaling pushes training cost up by 150 percent for 2× parameters because memory requirements can force multi-GPU configurations rather than a simple larger single-device run. Model count creates the broadest multiplicative effect because every additional model touches training, inference, data, monitoring, deployment, and governance at once, making portfolio growth the most expensive scaling dimension.

12.7.11.10 Decision framework: Speed vs. efficiency

TCO analysis enables principled decisions about when to optimize for development speed vs. operational efficiency. Equation 12.20 calculates the breakeven point for optimization investments:

$$T_{\text{breakeven}} = \frac{C_{\text{optimization}}}{C_{\text{save,monthly}}} \tag{12.20}$$

where $C_{\text{optimization}}$ is the one-time cost of implementing an optimization and $C_{\text{save,monthly}}$ is the monthly savings it produces.

The breakeven equation becomes useful only after the dominant cost component is known. Table 12.31 maps each cost structure to the optimization posture and payback threshold that should govern investment decisions.

Table 12.31: Cost-Structure Decision Rules: Optimization posture changes as TCO shifts from iteration-dominated to training-dominated to inference-dominated.

Dominant cost structure	Operating context	Optimization posture	Payback threshold
Iteration costs dominate ($C_{\text{iter}} > 50\%$ of TCO)	Early stage	Optimize for development velocity, accept higher per-inference costs while experiments are still changing, and invest in experiment infrastructure that reduces T_{exp}. Defer infrastructure efficiency work until the cost structure shifts.	Defer unless it directly accelerates iteration.
Training costs dominate ($C_{\text{train}} > 40\%$ of TCO)	Growth stage	Invest in training efficiency through mixed precision, gradient checkpointing, spot instances with checkpoint-and-resume, and architecture changes that reduce training time.	3–6 month payback is acceptable.
Inference costs dominate ($C_{\text{infer}} > 40\%$ of TCO)	Scale stage	Prioritize serving optimization through quantization, batching, caching, and distillation because model optimization ROI is highest when every query repeats the savings.	1–3 month payback is required.

12.7.11.11 Worked example: Optimization investment decision

A production company ($C_{\text{infer}} = \$400K/\text{month}$) evaluates INT8 quantization:

- Implementation cost: $80,000 (engineering time + validation)
- Expected inference cost reduction: 40 percent
- Monthly savings: $400,000 × 0.40 = $160,000

$T_{\text{breakeven}} = \frac{\$80{,}000}{\$160{,}000/month} = 0.5$ months

Breakeven in 2 weeks makes this investment highly attractive. However, if the same company ($C_{\text{train}} = \$150K/\text{month}$) evaluates a training optimization:

- Implementation cost: $80,000
- Expected training cost reduction: 30 percent
- Monthly savings: $150,000 × 0.30 = $45,000

$T_{\text{breakeven}} = \frac{\$80{,}000}{\$45{,}000/month} = 1.8$ months

Still attractive, but lower priority than inference optimization due to longer payback and smaller absolute savings. That same payback logic generalizes into an optimization priority matrix. Table 12.32 maps the dominant cost structure to the optimization work that should come first:

Table 12.32: Optimization Priority Matrix: Match optimization investments to current cost structure. When iteration dominates, invest in developer velocity, not infrastructure. When inference dominates, model optimization yields fastest payback. Data-dominated cost structures (unusual but possible with large feature stores) require storage and transfer optimization before model improvements.

Dominant Cost	First Priority	Second Priority	Avoid
Iteration (>50%)	Experiment velocity	Developer tooling	Infrastructure optimization
Training (>40%)	Training efficiency	Spot/preemptible compute	Over-engineering serving
Inference (>40%)	Model optimization	Serving infrastructure	Excessive retraining
Data (>30%)	Storage tiering	Egress reduction	Premature feature expansion

12.7.11.12 TCO-driven architecture decisions

TCO analysis should inform architectural choices alongside operational optimization. The utilization-driven build-vs.-buy analysis in section 12.3.2 applies per model: usage-based platform services may have lower TCO than self-managed infrastructure when a workload cannot sustain the utilization that amortizes owned capacity, especially where iteration costs dominate. Model architecture selection changes the same equation because a model requiring 2× training cost but 0.5× inference cost may have lower TCO at scale where inference dominates. Retraining frequency increases C_{train} and C_{data} but may reduce C_{iter} by catching drift earlier and avoiding emergency interventions. Feature complexity similarly increases C_{data} through storage and processing and C_{train} through longer training, but it may reduce C_{iter} by improving model performance faster.

The TCO framework transforms these architectural debates from opinion-based discussions into quantitative analyses with measurable outcomes. Among these investments, one particular component consistently emerges as both the most expensive to build and the most valuable to standardize. Because data represents the lifeblood of every model in the fleet, the platform must solve the persistent challenge of serving consistent features at scale.

12.8 Feature Store Operations

Consider a fraud detection system that needs to know a user's transaction volume over the last ten minutes. In production, this requires a sub-millisecond database lookup. During training, however, evaluating a year's worth of historical data requires a massive distributed join across billions of rows. When the logic to compute "transaction volume" differs even slightly between the batch processing and the real-time lookup, the resulting training-serving skew will silently destroy the model's performance.

The first design decision is the dual-store architecture: online stores serve low-latency feature lookups, while offline stores generate reproducible training data. Operating these systems at scale presents unique challenges in freshness, consistency, and performance. The core problem feature stores solve is the **training-serving gap**: features computed during training must be reproducible during serving, but the computational contexts differ fundamentally: *batch* vs. *real-time*, *hours* vs. *milliseconds*. That gap often manifests as training-serving skew, a critical failure mode where subtle differences in feature processing logic between batch training and real-time inference pipelines cause silent accuracy degradation.

> **Systems Perspective 12.5: Training-serving skew**
>
> At fleet scale, the concern shifts from detecting training-serving skew in a single model to controlling it as a platform-level property. A feature store reduces skew by making training and serving pipelines read from shared feature definitions and materialized feature values where possible, turning $f_{\text{train}}(x) \equiv f_{\text{serve}}(x)$ into an architectural invariant that can be tested and monitored rather than a manual convention repeated by every team.

The guarantee has to hold across both computational contexts, which is why feature stores are an architectural boundary rather than a storage convenience. During training, features are computed in batch over historical data, where a pipeline can spend hours computing features over millions of

examples and the priority is correctness and coverage. During serving, the same definitions must be available through low-latency lookups, where the priority is bounded tail latency for live requests.

12.8.1 Feature store architecture

The dual-store pattern (figure 12.11) resolves this conflict by separating the offline analytical store from the online key-value store, connected through a shared materialization layer.

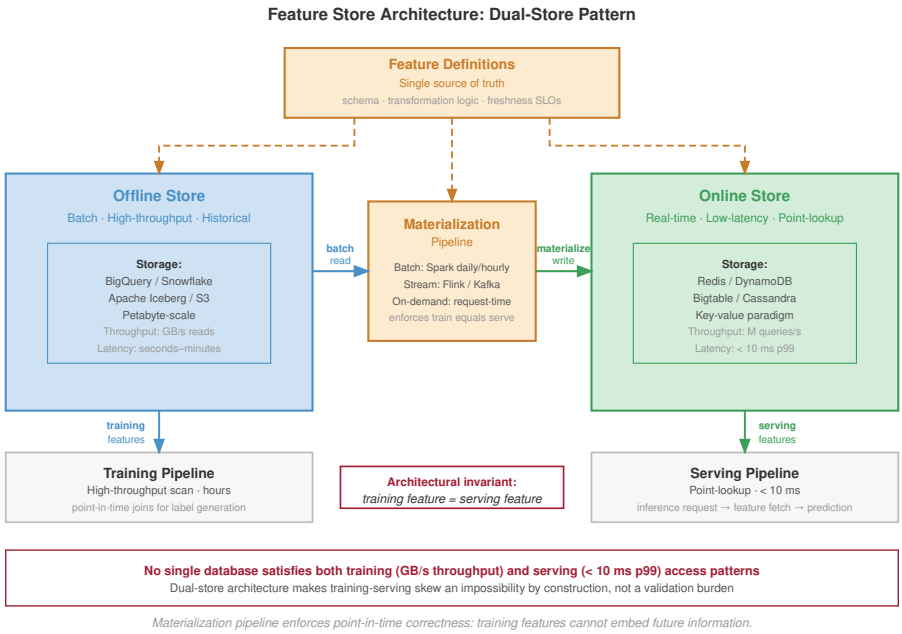


Figure 12.11: Feature Store Architecture: Resolving the conflict between training (high-throughput batch scans) and serving (low-latency point lookups) through a dual-store architecture. Features are materialized from batch and streaming sources into an offline store (for training) and an online store (for serving), ensuring consistency across the ML lifecycle.

A feature store is not merely a database; it is an architectural pattern designed to resolve the fundamental conflict between the data access patterns of model training and real-time serving. Training requires high-throughput analytical scans over massive historical datasets, while serving requires low-latency point lookups for individual prediction requests. No single database system can efficiently satisfy both constraints, forcing the adoption of a **dual-store architecture** composed of an offline store and an online store.

The **offline store** is the system of record for all historical feature data, often holding petabytes of information. It is optimized for the massive sequential reads characteristic of training data generation, where a single query might scan terabytes of data to build a feature set for millions of examples. The key metric is throughput, not latency. Systems like BigQuery, Snowflake, or data lakes built on S3 with formats like Apache Iceberg are common choices, designed to parallelize these large-scale analytical queries.

The **online store** is purpose-built for speed at serving time. When a prediction request arrives, the model needs its features within a strict latency budget—often a p99 of less than 10 milliseconds. For a platform serving 10,000 models, this can translate to millions of queries per second. This requires a key-value paradigm, using systems like Redis, DynamoDB, or Bigtable that are optimized for retrieving a small number of values for a specific entity key.

Features are loaded into these stores through a process called **materialization**. Batch computation pipelines, often running on Spark, execute daily or hourly to generate features from historical data. Streaming computation pipelines using Flink or Spark Streaming generate features in near-real-time from event streams like Kafka. On-demand computation calculates features at request time when freshness requirements exceed batch frequency. The central architectural challenge is ensuring

consistency between the two stores during materialization. If the offline store contains a feature value for training that was not identically available in the online store at the time of prediction, it introduces a subtle form of data leakage that can lead to silent degradation of model performance in production—the training-serving skew problem formalized above.

For a platform managing over 10,000 models, this centralized dual-store architecture is not optional. It provides a governable contract between data production and model consumption, preventing thousands of independent, unmaintainable feature pipelines and ensuring that all models are built from a consistent, high-quality source of truth.

12.8.2 Freshness SLOs

Feature freshness represents the delay between real-world events and their reflection in feature values. Table 12.33 maps four feature types to their freshness requirements: static features like user demographics tolerate day-scale staleness with batch computation, while real-time features capturing the last user action demand seconds-scale freshness through streaming or on-demand computation.

The freshness requirement is particularly acute for recommendation systems, where stale features become a direct tax on engagement.



Lighthouse 12.2: Archetype B (DLRM at Scale): The staleness tax

Archetype B (DLRM at Scale), the DLRM workload, is uniquely sensitive to freshness. Unlike Archetype A (GPT-4/Llama-3), where grammar rules do not change, this archetype's "ground truth" changes every second. If a user clicks a video about baking, and the feature store has a 10-minute lag, the next 100 recommendations will miss this new intent. This "staleness tax" directly degrades engagement, forcing DLRM systems to adopt expensive streaming pipelines over cheaper batch ones.

Feature freshness requirements by type set the SLOs that the monitoring and alerting system must enforce.

Table 12.33: Feature Freshness Requirements by Type: Four feature categories with SLO thresholds and computation patterns. Static features (user demographics) tolerate day-scale staleness with batch computation; real-time features (last user action) demand seconds-scale freshness through streaming or on-demand computation, directly impacting recommendation quality and engagement.

Feature Type	Example	Freshness SLO	Computation Pattern
Static	User demographics	Days	Batch
Slowly changing	User preferences	Hours	Batch
Session-level	Current session context	Minutes	Streaming
Real-time	Last action	Seconds	Streaming/On-demand

Freshness becomes operational only when the SLO turns into a measured quantity. Equation 12.21 defines feature staleness as the difference between current time and the most recent feature update, enabling direct comparison against the thresholds in table 12.33:

$$\text{Staleness} = t_{\text{current}} - t_{\text{feature\_update}} \tag{12.21}$$

The staleness scalar gives batch and streaming paths a shared alert rule. Streaming features should stay near zero except during pipeline disruption, while batch features increase predictably between scheduled updates; the monitoring system should therefore compare each feature against its own SLO rather than apply one global freshness threshold.

12.8.2.1 Worked example: Freshness impact on model quality

A recommendation system uses user interaction features with different freshness levels. Testing on historical data produces the engagement lift in table 12.34:

Table 12.34: Engagement Lift by Feature Freshness: Historical testing of a recommendation system. Real-time features (under one minute) deliver 4.2 percentage points of additional engagement lift over daily features. The 2.1-point gap between hourly and real-time features quantifies the value of investing in streaming feature infrastructure.

Feature Freshness	Engagement Lift vs. Baseline
Real-time (< 1 min)	+12.3%
Near real-time (< 5 min)	+11.8%
Hourly	+10.2%
Daily	+8.1%

The engagement difference between hourly and real-time features is 2.1 percentage points. If this translates to $10 million in annual engagement value, investing in real-time feature infrastructure may be justified if costs are below this value.

Checkpoint 12.6: Pricing the streaming and batch freshness decision

Table 12.34 shows that hourly-to-real-time features add 2.1 percentage points of engagement lift. Treat that lift as $10 million in annual engagement value and apply it to the choice between streaming and batch infrastructure.

- ☐ **Freshness break-even**: Determine the maximum annual cost of a real-time feature pipeline that still breaks even on the 2.1-point lift, holding the $10 million engagement value constant, and state the cost threshold under which streaming is a clear win.
- ☐ **Lift per point**: Compare the daily-to-hourly, hourly-to-near-real-time, and near-real-time-to-real-time transitions by return per percentage point of latency reduction, and infer the right freshness target for a team that cannot afford full streaming.
- ☐ **Scaling to users**: If the $10 million annual engagement value scales linearly with platform user count, determine the user count where a $5 million per year streaming feature platform pays for itself, state the range where it does not, and choose the alternative architecture from table 12.33.

12.8.3 Point-in-time correctness

Training data must use features as they existed at the time of each training example. Figure 12.12 illustrates the "time travel" problem: a batch job computing `total_clicks_today` at midnight produces a value of 10, but using this to train a model predicting behavior at noon introduces leakage since the true value at noon was only 4. Using current feature values to label historical events creates data leakage[18] that inflates offline metrics but fails in production.

The contrast in figure 12.12 is stark: the correct point-in-time value at noon is 4 clicks, but a naive batch join would supply the midnight value of 10, inflating the training signal by 2.5 times and producing a model that cannot generalize to production. This "time travel" failure is common because the feature value is valid in the database but invalid for the historical prediction moment: `total_clicks_today` really is 10 at midnight, but the noon prediction should only see the 4 clicks that existed by noon. Training on the midnight value leaks future information into the example, so the model learns to "cheat" with signals unavailable in production. The result can be spectacular offline metrics and catastrophic production failure when future data disappears.

Feature stores implement point-in-time joins that retrieve feature values as of specific timestamps, as shown in listing 12.7.

[18] **Data Leakage**: An error where future information contaminates training data. The systems danger is that leakage produces models with spectacular offline metrics – sometimes 99 percent+ accuracy – that fail completely in production where future information is unavailable. At feature-store scale, leakage risk multiplies because batch-computed features may embed temporal information invisible to the model team but exploitable by the optimizer, making point-in-time correctness an infrastructure guarantee rather than a per-team responsibility.

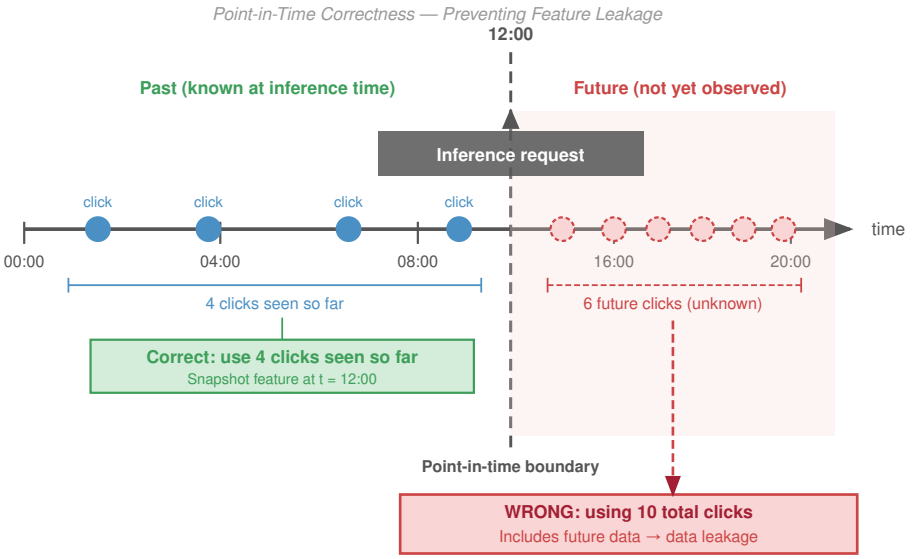


Figure 12.12: Point-in-Time Correctness: Preventing data leakage by joining training events with feature values as they existed *at the event timestamp*, not the *current values*. This ensures the model learns from the information actually available at inference time.

Listing 12.7: Point-in-Time Join: A SQL lateral join that retrieves the most recent feature values available before each event, preventing future data leakage into training examples.

```
SELECT
    e.user_id,
    e.event_timestamp,
    e.label,
    f.feature_1,
    f.feature_2
FROM events e
LEFT JOIN LATERAL (
    SELECT feature_1, feature_2
    FROM features f
    WHERE f.user_id = e.user_id
      AND f.feature_timestamp <= e.event_timestamp
    ORDER BY f.feature_timestamp DESC
    LIMIT 1
) f ON TRUE
```

This query retrieves the most recent feature values that existed before each event, ensuring training data reflects production reality.

Point-in-time correctness has a storage cost because the store must retain feature history rather than current values alone. The same guarantee that prevents leakage also multiplies retained state:

$$\text{Storage} = N_{\text{entities}} \times N_{\text{features}} \times \frac{T_{\text{retention}}}{T_{\text{update}}}$$

For 100 million users, 1000 features, 1 year retention, and updates every 1 hour, the history contains 876 trillion feature values. At 100 bytes per value, this represents approximately 87.6 PB before

compression. Efficient feature stores use compression, columnar storage, and retention policies to manage this scale.

12.8.4 Feature versioning and lineage

Feature definitions are production interfaces, so changing one without versioning can break every dependent model. Versioning and lineage make this evolution governable rather than implicit.

A representative `user_engagement_score` incident makes the mechanics concrete. A platform team changes the feature from a 30-day z-score to a 7-day min-max scale. The output may still be a floating-point number, so schema validation alone can pass while the feature's meaning changes completely. Versioning names the semantic change before it reaches production; lineage identifies every model trained or served with the old meaning; backfill recomputes historical values for the new meaning; and freshness and quality checks decide whether the rewritten history is safe to use. Without that chain, a feature migration becomes a silent multi-model regression.

A safe feature interface records both version dimensions and lineage fields. Table 12.35 keeps those fields tied to their operational use.

Table 12.35: Feature Version and Lineage Components: Feature versioning separates logic, data, and schema changes; lineage records the path needed to debug, audit, and reproduce feature values. Together, these fields turn a feature definition into a governed production interface.

Component	What It Records	Operational Use
Definition version	Computation logic for the feature	Blocks silent semantic changes when the formula changes but the type does not.
Data version	Source snapshot or stream contract	Distinguishes model changes from upstream data changes.
Schema version	Output type, shape, units, and nullability	Preserves API compatibility checks while still naming semantic shifts.
Consumer declaration	Exact feature versions each model consumes	Gives the platform a place to block unsafe updates before serving.
Source lineage	Source tables, streams, and their versions	Traces bad predictions back to the upstream data that produced them.
Transformation and runtime provenance	Transformation code, timestamp, and environment	Reproduces historical values for audit, debugging, and compliance.
Quality and freshness metrics	Validation results observed when the value was produced	Estimates blast radius and decides whether rewritten history is safe to use.

12.8.5 Backfill procedures

A backfill is a production migration over history: changing a feature definition rewrites the training record for every model that depends on it, so the platform must bound the compute cost, validate that recomputed values match, and preserve a path back. Treating the backfill as a migration changes the operational question from whether the computation can run once to whether history can be rewritten without corrupting future training data.

At scale, historical data may have moved into cold storage, so recomputation first becomes a data-placement problem. The long historical window then competes with production workloads for compute. Even after the job finishes, the platform cannot simply trust the new values: it must compare overlapping periods against the original computation and coordinate with dependent pipelines so training jobs do not mix old and new semantics mid-run.

The practical pattern is incremental. Historical data is processed in date partitions, and each partition is validated before the next one starts. During a dual-write period, old and new computations run in parallel, which gives the platform overlapping values for comparison before cutover. Rollback remains possible because the previous feature version and its lineage stay available until dependent models have been retrained, evaluated, and redeployed.

12.8.6 Scale challenges

Feature stores at recommendation system scale face a coupled systems problem: request volume, latency budget, and storage history all bind at once.

The arithmetic behind a large recommender explains why feature stores become serving infrastructure rather than a metadata convenience. 1 billion daily recommendations with 100 features per recommendation produces 100 billion feature lookups per day. That average load is roughly 1.16M requests/s before peak traffic, which commonly reaches 5–10× the mean.

Feature retrieval must fit inside the overall latency budget because every extra lookup millisecond steals time from ranking. If the recommendation budget is 50 ms, the feature store may receive only 5–10 ms. Network overhead can consume 1–2 ms of that allocation, leaving roughly 3–8 ms for the store lookup itself. Meeting this budget requires in-memory storage, careful batching, and geographic placement close enough to the serving fleet that network latency does not dominate.

Production feature stores also carry two time horizons at once. The online path keeps current values for billions of entities and thousands of features per entity, usually in the terabyte range before replication. The offline path keeps enough historical state to reconstruct training examples, which can push retained feature history into petabytes. Multi-region replication then serves both availability and latency, but it also makes freshness and consistency visible operational concerns.

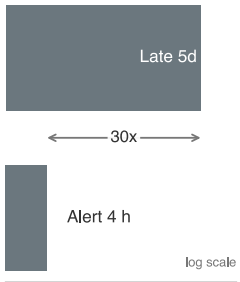
12.8.7 Data quality operations

Data quality issues are a major source of production ML problems (Polyzotis et al. 2017). While model monitoring detects symptoms, data quality monitoring prevents problems at their source. At scale, data quality operations become as critical as model quality operations, requiring systematic monitoring, validation, and incident response procedures.

The first step is measurement: data quality must be quantified in dimensions that map directly to prevention gates. Completeness measures whether expected records and fields are present. A daily data pipeline expected to produce 10 million user events but delivering only 8.2 million user events is 82 percent complete, which usually indicates a pipeline failure or upstream data source issue rather than ordinary statistical variation. Consistency captures schema compliance and referential integrity: an age feature should fall between 0 and 120, while values such as 999 or -1 suggest sentinel leakage or data errors. Production systems often reject batches that fail more than 5 percent of validation rules because allowing corrupt data through is more expensive than delaying a batch.

Timeliness measures freshness against the SLO of the consuming model. Fraud detection may need features less than 100 milliseconds old, while demographic features can tolerate staleness measured in days; the alert threshold should therefore be tied to the feature's use, not a global constant. Accuracy asks whether values remain correct within expected distributions. Sensor drift after calibration lapses, for example, can leave values well-typed but wrong. Statistical tests such as Kolmogorov-Smirnov for continuous features, chi-square for categorical features, and Maximum Mean Discrepancy for high-dimensional data turn those accuracy checks into recurring gates.

Validation then turns those measurements into a layered defense. Schema validation enforces expected column names, types, and constraints at ingestion; tools such as Great Expectations, TensorFlow Data Validation, and Pandera provide declarative schemas with automated checks. Distribution monitoring tracks feature distributions over time, catching drift that may not violate a type constraint but still indicates an upstream change likely to degrade model performance. Cross-field validation adds the business logic that neither schema nor marginal distributions can express: if `country="USA"`, the `zip_code` should match US postal formats; if `order_status="shipped"`, a `shipping_date` should exist; and age derived from birthdate should match any explicit age field.



Every minute of delayed detection compounds the cost.

12.8.7.1 Worked example: Debugging data quality incident

A recommendation model's CTR drops 8 percent over 5 days. Initial hypothesis focuses on model drift, but data quality investigation reveals the root cause.

The investigation starts with the model symptom and then works backward through the data path. Model metrics confirm that CTR declined from 4.2 percent to 3.87 percent, but freshness checks show that features still arrive within the 1-hour SLO. That rules out a stale pipeline and points to semantic change. Distribution analysis then shows the `user_engagement_score` mean shifting from 0.42 to 0.31, a 26 percent decline, and lineage tracking ties the shift to an upstream pipeline version deployed 5 days earlier. The root cause is not model drift but a feature computation bug in the new pipeline version.

The quantitative impact is the relative CTR loss multiplied by weekly revenue and the fraction of the week exposed: 8 percent of $15M/week over 5 days produces $857K in lost revenue. Distribution

monitoring with automated alerts would detect the shift within 4 hours, reducing impact by 97 percent. The resolution is to roll back the pipeline to the previous version, redeploy models with correctly computed features, and add a distribution validation gate to prevent future pipeline deployments with feature shifts exceeding the 10 percent threshold.

12.8.7.2 Operational integration

Data quality operations integrate into production systems by turning checks into enforceable contracts. Data contracts establish formal agreements between data producers and consumers, specifying schema requirements, freshness SLOs, quality thresholds for completeness and accuracy, and escalation procedures for violations. When a data producer changes schema or computation logic, the contract forces explicit negotiation with consumers rather than allowing a silent interface change.

The contract becomes operational through continuous validation and incident response. Preingestion validation rejects bad source data before accepting it, posttransformation checks verify that computation logic still produces expected values, and final validation prevents corrupted features from reaching models. When quality degrades anyway, automated alerts page the on-call team with severity and affected systems, circuit breakers keep known-bad data out of serving, fallback paths use last-known-good data or cached features, and bad batches are quarantined for forensic analysis. Root-cause analysis then repairs the pipeline rather than only clearing the current alert.

At enterprise scale with thousands of features, monitoring must aggregate by likely shared failure mode. Individual feature monitoring creates alert fatigue, while hierarchical monitoring preserves signal by grouping features along operational boundaries. Source-system grouping catches shared upstream failures, such as a payment-processing outage that affects all payment-related features simultaneously. Computation-pipeline grouping catches errors in transformation code or dependencies. Update-frequency grouping separates streaming features from daily batch features because staleness thresholds and alert sensitivity vary by update pattern. Business-domain grouping then routes user demographic, product catalog, and interaction-feature alerts to the teams that understand the affected models.

Freshness monitoring applies the same aggregation rule to time. For 10,000 features updated at different frequencies, continuous freshness checking generates substantial overhead, so efficient implementations reduce the number of checks without losing systemic signal. Sampling 1 percent of representative entities rather than every feature value is usually enough to detect systemic freshness failures. Pipeline-level aggregation is even cheaper when one pipeline updates 500 features, because the pipeline update timestamp captures the freshness state of the group. Threshold stratification reserves per-feature monitoring for revenue-critical features and uses aggregated monitoring for features whose staleness has lower immediate impact. The resulting overhead scales with the number of distinct features, not the volume of feature values, so efficient implementations maintain $\mathcal{O}(F)$ overhead even when feature values scale to billions of entities.

12.9 Organizational Patterns

Feature stores resolved who owns a shared feature; the broader question is who owns the shared platform itself. Technical infrastructure alone is insufficient for ML operations at scale: organizational structure determines whether platform capabilities become shared infrastructure or another bottleneck. The central question is where an organization places invariants. Deployment safety, observability, compliance, cost control, and feature consistency can either be enforced once by a platform team or rediscovered differently by every model team.

12.9.1 Centralized platform team

A centralized ML platform team maximizes consistency by building and maintaining shared infrastructure while model teams focus on model development. Figure 12.13 places that pattern alongside embedded and hybrid alternatives, each trading off consistency against velocity.

Centralization works when the same operational mistakes would otherwise recur across the organization. A shared team can enforce consistent deployment, monitoring, and governance practices; invest once in training, serving, and observability infrastructure; and concentrate deep ML systems expertise that would be hard to replicate across many product teams. It also gives

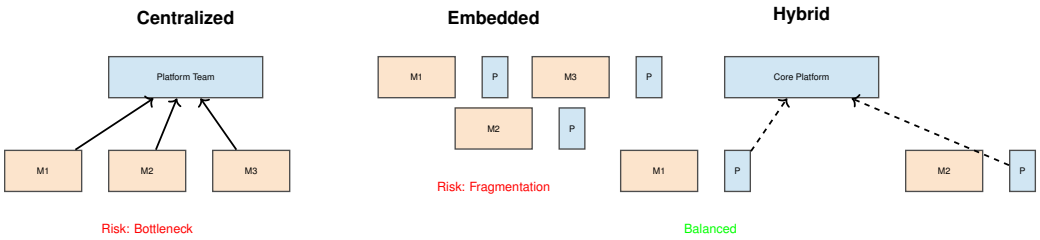


Figure 12.13: Organizational Patterns for ML: (Left) Centralized model provides consistency but risks bottlenecks. (Center) Embedded model provides velocity but risks fragmentation. (Right) Hybrid usage of a core platform team with embedded specialists offers a balance of standardization and responsiveness.

infrastructure engineers a visible career path rather than scattering them as one-person support functions.

The same structure can become a queue. If every platform request routes through one team, model teams wait on prioritization decisions they do not control. Distance from product context can also produce abstractions that are technically elegant but poorly matched to the consuming team's constraints. Centralization is therefore strongest when common invariants dominate local variation, and weakest when model teams need rapid domain-specific changes.

12.9.2 Embedded ML engineers

The embedded pattern chooses proximity over uniformity by placing ML infrastructure expertise inside model teams, with coordination through communities of practice rather than a single reporting line. The benefit is responsiveness: the infrastructure engineer sees the team's data, model behavior, release pressure, and incident history directly, so local changes do not wait for cross-team scheduling.

The cost appears later, when each team has solved deployment, monitoring, and feature management slightly differently. Fragmentation wastes engineering effort, complicates integration, and makes platform quality depend on the skill and bandwidth of whoever happens to be embedded with a given team. Embedded teams therefore work best when local speed matters more than cross-team consistency, and worst when every team is reimplementing the same platform substrate.

12.9.3 Hybrid models

Most mature organizations adopt hybrid approaches because neither pure centralization nor pure embedding handles both shared infrastructure and domain-specific needs well. The core platform team owns compute, storage, networking, common training and serving systems, monitoring, security, compliance, and cost controls. Domain platform engineers then stay close to recommendation, language, vision, fraud, or search teams, where workload-specific requirements change faster than the shared substrate should.

A federated version of the same idea distributes implementation work without giving up architectural control. Major contributing teams help maintain platform components, but standards, ownership boundaries, and prioritization are coordinated explicitly. This works only when governance has real authority: otherwise federation degenerates into embedding with more meetings.

12.9.4 Organizational pattern selection

The appropriate organizational pattern depends on where standardization creates more value than local autonomy. Table 12.36 summarizes the organizational pattern decision factors: higher model counts, stricter regulatory requirements, and immature infrastructure favor centralized platforms, while heterogeneous model portfolios and smaller organizations may benefit from distributed expertise:

Table 12.36: Organizational Pattern Decision Factors: Five criteria for choosing between centralized and distributed ML platform teams. Higher model counts (100+), stricter regulatory requirements, and earlier infrastructure maturity favor centralized platforms; heterogeneous model portfolios and smaller organizations may benefit from distributed expertise with coordination through communities of practice.

Factor	Favors Centralized	Favors Distributed
Model count	Higher (100+)	Lower (10–20)
Model similarity	Homogeneous	Heterogeneous
Organization size	Larger	Smaller
Regulatory requirements	Stricter	Lighter
Infrastructure maturity	Earlier stage	Later stage

The selection logic becomes clearer in a concrete organization. A technology company with 50 ML engineers across 8 teams operates 80 production models across recommendation, fraud, search, and ads. Each team maintains its own deployment and monitoring, which has created duplicated infrastructure work and inconsistent integration practices. The model count is high enough that shared platform investment should pay off, but the domain diversity is real enough that a purely centralized team would struggle to understand every workload.

The resulting design is hybrid. A central platform team of roughly 12–15 engineers owns core infrastructure, while domain-specific platform leads remain embedded with the largest model teams. A community of practice coordinates standards and a shared contribution model lets domain teams upstream reusable components instead of maintaining permanent forks.

By establishing shared contribution models and domain-specific platform leads, organizations can standardize the shared control plane without turning every domain-specific request into a platform bottleneck. The same platform engineering, feature management, and governance trade-offs become more concrete in the hyper-scale systems built by large technology companies.

12.10 Case Studies

The platform cases instantiate the operations trade-offs developed above: standardization vs. team-local velocity, abstraction vs. control, freshness vs. cost, and governance vs. autonomy. Each case shows a different way to keep a multi-model fleet repeatable without turning every model into bespoke infrastructure.

12.10.1 Uber Michelangelo

Uber's Michelangelo platform shows why repeatability becomes the first platform requirement at company scale (Hermann and Del Balso 2017). This case embodies the first operations trade-off in the platform sequence: standardization vs. team-local velocity. Before Michelangelo, Uber faced a "dual implementation" problem: data scientists developed models in Python, but engineering teams had to rewrite feature pipelines for production, creating delay and logic divergence. The platform combined model training, deployment, serving, workflow management, and feature management for use cases including ETA prediction, fraud detection, and recommendations. Its feature store centralized reusable features such as trip-distance aggregates, reducing duplicated feature engineering and addressing training-serving consistency.

The operational challenge at Uber was not just model count, but the need to make model development repeatable across many product teams. Michelangelo provided a "Golden Path" for common workflows: teams could train models, create offline training datasets, serve predictions online or in batch, and reuse platform-managed feature pipelines. This standardization improved automation across training, evaluation, deployment, and monitoring while still requiring product teams to own model behavior and business-specific validation.

A critical evolution in platforms like Michelangelo is the move from batch-centric training toward fresher online features and lower-latency serving. Early platform designs commonly pair offline stores for training data with online stores for serving precomputed or recently computed features. Real-time products such as food delivery and fraud detection then introduce consistency challenges: the same feature definition must produce compatible values for offline training sets and online

inference requests. The lesson is durable even when implementation details vary: centralized ML platforms must provide shared abstractions for features, training, serving, experimentation, and monitoring without turning every model into bespoke infrastructure.

12.10.2 Meta FBLearner Flow

Meta's FBLearner Flow shows how democratizing ML infrastructure requires hiding resource management without hiding workflow structure (Engineering 2016). Facebook reported that more than 25 percent of its engineering organization used FBLearner Flow, that the system had trained more than one million models, and that its online prediction service made more than six million predictions per second. The primary operational challenge was the "N-models problem": many teams needed repeatable model workflows without hand-building infrastructure for every ranking, ads, recommendation, or integrity model. FBLearner addressed this by treating the training pipeline as a DAG that abstracts away the underlying infrastructure. An engineer defines the workflow in code, and the platform handles resource allocation, dependency management, and fault tolerance.

A defining characteristic of large social ML platforms is feature freshness. For products like feed ranking, the value of a feature (for example, "user just clicked 'like' on a similar video") can decay quickly. This pushes platforms toward stream processing, online feature computation, and monitoring that tracks data distributions as well as service health. Standard metrics like CPU usage are insufficient to detect silent failures in feature streams; production ML systems need checks for missing features, distribution shifts, and training-serving skew.

To manage deployment risk, platforms at this scale commonly use Shadow Mode (or dark launching), canary releases, and online evaluation before broad promotion. Candidate models can run alongside the current production model, receiving live traffic while their outputs are logged but not shown to users. This phase verifies operational integrity (latency, memory usage, error rates) and helps build comparison data before a model affects user experience.

12.10.3 Netflix ML infrastructure

Netflix's ML infrastructure shows how personalization platforms manage fan-out cost under a strict latency budget (Gomez-Uribe and Hunt 2015). Rather than one model making one decision, the recommendation system combines multiple ranking and personalization algorithms for tasks such as page generation, row selection, search, "continue watching," and artwork personalization. The operational challenge is **fan-out cost**: every additional online ranker or feature dependency consumes latency budget. Production systems therefore separate candidate generation, offline model training, online ranking, and caching so the homepage can remain personalized without executing an unbounded number of expensive models per request.

A specific problem Netflix tackles aggressively is the **"Cold Start"**—the inability to recommend content to a new user with no history, or to recommend a brand new show with no viewing data. Their infrastructure solves this using "online learning" bandits that dynamically balance exploration and exploitation. When a new show launches, the system allocates a small "budget" of impressions to test the title on different user cohorts, rapidly converging on an effective audience. This requires an infrastructure capable of updating serving policies, exploration budgets, and recommendation state in near real-time, rather than waiting for the traditional daily batch training cycle. The system must also handle the "feedback loop" latency, ensuring that a user's interaction with a new title is immediately reflected in their subsequent recommendations, a requirement that pushed Netflix to move key parts of their feature engineering into the serving layer itself.

To validate these complex interactions, Netflix moved beyond standard A/B testing (Blog 2017) to "Interleaving." In a traditional A/B test, Group A sees Ranking 1 and Group B sees Ranking 2. This requires huge sample sizes and long durations to detect small improvements. Interleaving mixes the results of Ranking 1 and Ranking 2 into a single list for the same user, tracking which source the user actually clicks. This method cancels out user-level variance and creates a direct head-to-head comparison. The infrastructure supports this by allowing the serving layer to merge ranked lists on the fly and log attribution data with high fidelity. While this increases the complexity of the logging and attribution pipelines, the quantitative outcome is substantial: Netflix can detect statistically significant improvements with $100\times$ fewer users than traditional A/B testing, allowing them to iterate on ranking algorithms at a velocity that traditional testing frameworks could not support.

12.10.4 Google Vertex AI

Google Vertex AI illustrates the managed-platform abstraction problem: the platform must remove glue code while preserving enough control for custom workloads. Google aimed to solve the "glue code" problem—where 95 percent of ML code is infrastructure boilerplate—by providing a unified control plane that spans data labeling, training, and serving. A key architectural decision was the integration of **AutoML** as a first-class citizen alongside custom training. This allows the platform to perform neural architecture search (NAS) to find an effective model structure for a given budget. The operational trade-off here is "compute for human time": rather than an engineer spending weeks tuning hyperparameters, the platform spins up hundreds of parallel trials. This requires a multi-tenant scheduler capable of managing "burst capacity," allowing high-priority production jobs to preempt experimental AutoML trials without losing state, effectively maximizing cluster utilization.

For the serving layer, Vertex AI addresses the "noisy neighbor" problem inherent in multi-tenant environments. When thousands of customers deploy models to the same underlying fleet of Tensor Processing Units (TPUs) and GPUs, resource contention can cause unpredictable latency spikes. Google solves this with a strict containerization strategy and a "prediction sidecar" architecture. Every model runs in an isolated container, but a shared sidecar proxy handles logging, monitoring, and request batching. This separation allows the platform to enforce strict resource quotas (CPU, RAM, Accelerator RAM) and provide auto-scaling that reacts to custom metrics like "request queue depth" alongside CPU load. The quantitative benefit is a more predictable latency tail; hard isolation boundaries reduce the chance that a heavy batch job from one tenant degrades the real-time inference performance of another.

Vertex also tackles the feature store problem with a focus on consistency and compliance. Managed feature serving and point-in-time retrieval let training pipelines request only the feature values that existed at the timestamp of each training example. A common operational failure in ML is temporal data leakage—using feature values that were not available at prediction time. Managed feature serving also controls the training-serving skew developed in section 12.8 by standardizing feature definitions and serving paths across both contexts. These design decisions add storage and lineage overhead, but they eliminate entire classes of silent bugs. Combined with cost and utilization controls around training and serving, managed platforms can help teams manage total cost of ownership rather than just providing raw compute.

12.10.5 Spotify ML platform

Spotify's ML platform shows how personalization infrastructure balances exploration against exploitation at the scale of 500 million users. The core operational challenge is that optimizing strictly for immediate clicks (exploitation) creates "filter bubbles" that degrade long-term user retention. To counter this, the platform supports counterfactual evaluation and contextual bandit algorithms, methods for estimating outcomes and allocating exploration traffic under partial feedback, directly in the serving path. The architecture separates the "candidate generation" phase (retrieving 1,000 potential songs) from the "ranking" phase (ordering the top 10). The candidate generators are diverse—some are collaborative filtering models (Koren et al. 2009) updated daily, while others are "algotorial" heuristics updated in real-time. This decoupling allows the platform to mix-and-match retrieval strategies without rewriting the heavy ranking logic, facilitating rapid experimentation with new content types like podcasts and audiobooks.

A central component of their architecture is the **"Paved Road"** for model orchestration (Engineering 2019). Spotify describes a platform path built around TensorFlow Extended, Kubeflow Pipelines, metadata tracking, and centralized Kubeflow clusters so teams can move from experiments to repeatable production workflows. The operational lesson is lineage rather than cryptography: model artifacts, datasets, code, and pipeline metadata must remain connected enough that an engineer can trace a silent regression back to the data or workflow state that produced it. Canary deployment and rollback remain necessary release controls, but they are platform policies layered on top of that metadata substrate rather than claims established by the Spotify source itself.

Latency constraints at Spotify are nonnegotiable; playback must feel instantaneous. This forces a design trade-off where complex inference is often precomputed. For the "Discover Weekly" playlist, the platform runs massive batch inference jobs on weekends, storing the results in a low-latency

key-value store. However, for the "Home" screen, which must react to the song just listened to, they employ a hybrid approach. User embeddings are updated in near real-time using a streaming pipeline, but the heavy item-item similarity matrices are computed offline. This split architecture allows them to achieve sub-100 ms latency for the Home screen while still incorporating the user's immediate history. The systems lesson is that latency constraints determine where personalization work runs: precompute what can be stale, stream what must be fresh, and reserve online inference for what the user must experience immediately.

Strict latency requirements explain why mature personalization platforms precompute as much work as possible, but precomputation does not remove operational risk. The same shared registries, feature stores, and orchestrators that make platforms efficient also create control planes whose failures require disciplined incident response.

12.11 Production Debugging and Incident Response

At 3:00 AM, PagerDuty alerts the on-call engineer that revenue from the core recommendation system has dropped 15 percent in the last hour. The servers are healthy, the latency is normal, and there are no exception logs. In traditional software, a silent failure of this magnitude is rare; in machine learning systems, it is the expected reality. Debugging production ML systems requires fundamentally different investigative frameworks because the failures reside in data and mathematics as much as in code.

When production debugging consumes a large share of engineering attention, platform scale multiplies the complexity because failures may originate in data pipelines, model code, infrastructure, or emergent interactions between components that no single team owns end to end. Effective incident response therefore needs a diagnostic order: classify the incident, attribute the failure to the right dependency, mitigate the blast radius, and then convert the incident into a stronger platform control.

War Story 12.1: The control plane that disappeared

Context: Meta's global services depended on a backbone network that connected data centers and carried the internal control traffic needed to operate Facebook, Instagram, WhatsApp, and related systems (Janardhan 2021).

Failure mode: On October 4, 2021, a command intended to assess backbone capacity unintentionally disconnected Facebook data centers from each other. The resulting routing withdrawal also broke access to the DNS and internal tools needed for remote recovery.

Consequence: Major services were unavailable globally for hours while teams restored connectivity through constrained operational paths.

Systems lesson: Platform-scale operations must treat the control plane as a dependency with its own blast radius. Runbooks, out-of-band access, and staged network changes matter because the tools used to fix an outage can depend on the system that is down. ML platforms expose the same trap at fleet scale: training schedulers, model registries, feature stores, and inference routers often share the auth, DNS, and network plane an incident takes down, so recovery requires explicit isolation between the control plane and the ML systems that depend on it.

12.11.1 Incident classification

Incident classification is the first routing decision in an outage: the category tells responders which telemetry to inspect first, which owner to page, and whether the likely control is a rollback, a data repair, or infrastructure isolation. The categories below are organized around that first diagnostic signal.

Data incidents involve problems with input data: pipeline failures that prevent fresh data from reaching models, schema changes that break downstream consumers, data quality degradation from missing values or distribution shifts, and feature staleness exceeding SLO thresholds. These incidents often manifest as accuracy degradation across multiple models that share data sources, making data pipeline health the first diagnostic checkpoint.

Model incidents involve problems with model behavior, including accuracy degradation beyond acceptable thresholds, latency spikes indicating computational issues, memory exhaustion from

growing state (KV cache, buffers), and prediction bias shifts detected by fairness monitoring. Model incidents typically affect individual models. If multiple unrelated models degrade simultaneously, suspect a shared data or infrastructure issue rather than independent model problems.

Infrastructure incidents involve problems with the serving platform: GPU failures causing request errors, network partitions between model shards, load balancer misconfigurations routing traffic poorly, and container orchestration issues affecting deployments. These incidents tend to produce error rate spikes and timeout patterns rather than gradual accuracy degradation.

Business metric incidents involve unexpected changes to downstream KPIs, such as engagement drops without clear model or data cause, revenue anomalies during normal model operation, and user behavior shifts that affect model efficacy. These incidents are the hardest to attribute because they may stem from external factors (competition, seasonality, marketing campaigns) rather than ML system problems.

12.11.2 Attribution analysis

Attribution analysis protects diagnostic order: determine the root cause before implementing fixes. Temporal correlation analysis traces the degradation backward through recent changes:

```
Symptom: Recommendation engagement dropped 5% in past hour

Step 1: Check recent deployments
        → No model deployments in past 4 hours
        → Eliminate model change as cause

Step 2: Check feature freshness SLOs
        → user_features: 3 hours stale (SLO: 1 hour)
        → Feature pipeline delayed

Step 3: Check feature pipeline status
        → Kafka consumer lag: 10M events (normal: 10K)
        → Data ingestion bottleneck

Step 4: Investigate Kafka cluster
        → Broker disk 95% full on partition 7
        → Root cause identified
```

When a model's accuracy drops, the critical distinction is whether the root cause lies in the *data* or the *model*. The attribution flow separates four failure classes:

- **Data drift**: Input distribution shifted (new user demographics, seasonal patterns)
- **Feature staleness**: Pipeline delays causing stale predictions
- **Model decay**: Concept drift where true relationships changed
- **Upstream model change**: A model this model depends on was updated

The diagnostic sequence then tests the highest-probability shared causes before local model defects:

1. Compare current input distribution to training distribution
2. Check feature freshness across all input features
3. Examine performance on stable evaluation sets
4. Trace dependency graph for recent changes

Example 12.3: Feature pipeline cascade

Scenario: A platform team updates the normalization logic for a shared "User Engagement Score" feature, switching from a 30-day z-score to a 7-day min-max scale to better capture trends. They update the feature store definition and backfill the data.

Failure mode: Multiple downstream models, owned by different teams, suffer significant accuracy degradation because they all consumed the feature as an implicit interface.

Consequence: Without explicit lineage tracking, each team debugs its own model architecture and recent deployments before noticing the shared feature change.

Systems insight: Shared features need immutable versions (for example, `engagement_score_v2`) because feature semantics are production interfaces, not implementation details.

At platform scale, failures often span multiple models, so the affected set becomes a diagnostic signal. Cross-model correlation patterns reveal the likely root cause, as table 12.37 shows:

Table 12.37: Cross-Model Failure Patterns: At platform scale, the pattern of which models degrade simultaneously points directly at the shared infrastructure layer responsible. Correlated degradation across all recommender systems points at the feature store; an all-vision degradation points at the image preprocessing pipeline; a single-model degradation isolates the issue to that model.

Pattern	Likely Cause
All RecSys models degraded	Feature store issue
All vision models degraded	Image preprocessing pipeline
Single model degraded	Model-specific issue
Geographic pattern	Regional infrastructure
Time-based pattern	Batch job scheduling

12.11.3 Runbook development

A runbook earns its place when it preserves diagnostic order under time pressure. For ML systems, the correct diagnostic order is the opposite of what infrastructure instincts suggest: ML systems fail silently. When a feature pipeline stalls, feature values become stale, and the model continues processing those stale features while returning HTTP 200 responses with normal latency—prediction quality degrades invisibly and infrastructure checks appear green. An ML runbook must therefore begin with data and semantic health: verifying feature freshness, confirming that input distributions are consistent with training data, and checking whether a model version changed recently. Only after confirming that data and model behavior are intact should the runbook descend to infrastructure checks. If latency and error rates are nominal but business KPIs have dropped, the runbook must immediately direct the responder to feature freshness and distribution drift before any infrastructure investigation. That ordering starts from the user-visible symptom, tests data and semantic dependencies before infrastructure, and names escalation thresholds before the incident begins, so a responder under stress always knows the next test to run rather than only the system to inspect.

The reusable structure is a diagnostic flow, not a document template. For a recommendation engagement drop, the responder should move through the same control loop every time. Table 12.38 names the control questions that preserve that order under pressure.

Table 12.38: Runbook Diagnostic Flow: ML runbooks should preserve the order of reasoning from symptom to dependency to mitigation. The concrete dashboard names can change without invalidating the diagnostic model.

Step	Question	Example evidence
Detect user impact	Which product or cohort is degraded?	CTR, conversion, retention, revenue, or complaint rate
Localize dependency	Did a shared service change for many models?	Feature freshness, model registry, regional health
Bound blast radius	Can traffic be reduced, shadowed, or rolled back?	Canary metrics, rollback target, serving saturation
Escalate by evidence	Which team owns the failed control?	Data pipeline, platform, serving, or model owner
Learn from the gap	Which missing signal would have caught it sooner?	Freshness gate, slice alert, deployment guardrail

Runbook anti-patterns break diagnostic order in three common ways. A too-specific instruction such as "If BERT model fails, restart container" encodes one historical fix rather than a reusable investigation. A vague instruction such as "Investigate the issue" transfers all judgment back to the responder at the worst possible time. An outdated runbook is worse than no runbook when it points responders toward deprecated systems, dead dashboards, or obsolete contacts.

12.11.4 Post-incident reviews

Post-incident reviews (PIRs) transform incidents into organizational learning only when they turn one failure into a stronger platform control. The incident record should connect duration and user impact to the timeline of detection, attribution, mitigation, and recovery. Root causes should identify the control that failed or did not exist, and corrective actions should name owners and deadlines so the review changes the platform rather than only documenting the outage.

The useful PIR fields are the ones that force a control decision. Duration and impact quantify severity; the timeline exposes detection and attribution latency; root causes name missing controls; and corrective actions assign ownership for the next platform change. In a feature-freshness incident, the most important observation is not that a disk filled up, but that no freshness gate caught the stalled feature pipeline before engagement metrics moved. The questions in table 12.39 turn that timeline into platform work.

Table 12.39: PIR Control Questions: A post-incident review is valuable when it converts an incident timeline into a stronger platform invariant.

PIR question	Control decision it should force
How long was user impact visible?	Whether detection latency or mitigation latency dominated the incident.
Which dependency failed first?	Whether the missing control belongs to data, platform, serving, or model code.
Which signal arrived too late?	Whether a new freshness, slice, saturation, or canary guard is needed.
What would have bounded impact?	Whether rollback, traffic shadowing, admission control, or fallback was absent.
Who owns the next control?	Whether the review changes the platform rather than only recording blame.

Effective PIRs require psychological safety and a systems frame rather than individual blame. The review should identify which systems allowed the incident, not which person caused it. It should ask which signal would have detected the failure earlier, not why an individual missed it. It should prevent the class of failure, not only the exact failure that occurred.

12.11.5 Debugging distributed ML systems

Distributed training and inference shift debugging from a single failing process to a coordinated set of ranks, devices, and network paths. Operator triage localizes the failing rank, collective, or resource so the right specialist can act; deep performance debugging then uses NCCL logs, per-rank memory traces, and profilers to explain the mechanism.

12.11.5.1 Distributed failure diagnostics

Communication is the first failure class to rule out because a single blocked rank can stall the whole distributed job. NCCL[19] collective operations can fail silently or hang indefinitely, and listing 12.8 shows how debug logging identifies blocked ranks.

Listing 12.8: NCCL Debug Logging: Environment variables that enable verbose logging for diagnosing collective communication hangs and identifying blocked ranks.

```
# Enable NCCL debug logging
export NCCL_DEBUG=INFO
export NCCL_DEBUG_SUBSYS=ALL

# Identify slow/failed ranks
# Look for: "Waiting for" messages indicating a rank is blocking others
```

19 **NVIDIA Collective Communications Library (NCCL)**: NCCL implements AllReduce, AllGather, and ReduceScatter with topology-aware algorithms that exploit NVLink, NVSwitch, and InfiniBand. Debugging NCCL failures is notoriously difficult because collective hangs are deadlocks by another name: one rank waits for data another never sent, producing no error message and no crash – just silence. At fleet scale with thousands of GPUs, identifying the single blocked rank requires correlating logs across all participants.

When a collective hangs, the diagnostic sequence identifies the blocked rank before inspecting local causes:

1. Identify which ranks completed vs. blocked
2. Check network connectivity between problematic ranks
3. Examine GPU memory pressure on blocked ranks
4. Look for asymmetric workloads causing timing differences

Once communication has been checked, the same rank-aware discipline applies to model state and memory state. Training instabilities at scale often manifest as gradient issues, each with a distinct diagnostic path, as table 12.40 summarizes:

Table 12.40: Gradient Instability Diagnostics: Training instabilities at scale manifest as gradient issues with distinct diagnostic signatures. Loss NaN points at gradient explosion (check gradient norms); loss stuck points at vanishing gradients (per-layer norms); slow convergence points at a learning rate mismatch (compare to a single-GPU baseline); rank divergence points at nondeterminism (compare rank-specific losses).

Symptom	Likely Cause	Diagnostic
Loss NaN	Gradient explosion	Log gradient norms
Loss stuck	Vanishing gradients	Check per-layer norms
Slow convergence	Learning rate mismatch	Compare to single-GPU baseline
Rank divergence	Nondeterminism	Compare rank-specific losses

OOM errors require the same per-rank view because aggregate memory usage hides the device that actually crossed the limit. Listing 12.9 tracks memory across devices.

Listing 12.9: Per-Rank Memory Tracking: Reporting allocated, reserved, and peak memory on each GPU rank to diagnose OOM errors in distributed training.

```
# Memory tracking per rank
for rank in range(world_size):
    if torch.distributed.get_rank() == rank:
        print(f"Rank {rank}:")
        print(
            f"  Allocated: {torch.cuda.memory_allocated() / BILLION:.2f} GB"
        )
        print(
            f"  Reserved: {torch.cuda.memory_reserved() / BILLION:.2f} GB"
        )
        print(
            f"  Max allocated: {torch.cuda.max_memory_allocated() / BILLION:.2f} GB"
        )
    torch.distributed.barrier()
```

Memory leaks in distributed training usually trace to retained state that should have been released:

- Gradient accumulation buffers not freed
- Communication buffers retained across iterations
- Activation checkpointing not releasing properly

Profiling closes the loop by showing which rank limits throughput after communication and memory failures have been ruled out. Listing 12.10 records per-rank profiles with synchronization.

Listing 12.10: Distributed Profiling: Per-rank profiling with synchronization to identify straggler ranks that limit overall training throughput.

```
# Per-rank profiling with synchronization
with torch.profiler.profile() as prof:
    # Training iteration
    ...

# Gather profiles from all ranks
all_profiles = gather_profiles(prof)
# Identify slowest rank and operation
```

The slowest rank determines overall throughput, so straggler diagnosis must inspect hardware, network, data, and degradation causes:

- Thermal throttling on specific GPUs
- Network congestion on particular switches
- Uneven data loading across ranks
- GPU hardware degradation

Those diagnoses become operational only when the on-call path routes each failure class to the owner who can act on it quickly.

12.11.6 On-call practices for ML teams

ML systems require specialized on-call practices because prediction quality, data freshness, and dependency graphs join uptime as reliability concerns. The debugging sections above explain why an incident can begin as a business metric anomaly and end in a feature pipeline, a model registry, or an NCCL collective. On-call design has to make that diagnostic burden sustainable, building on established Site Reliability Engineering (SRE)[20] principles (Beyer et al. 2016).

A structural challenge specific to ML on-call is that the expertise required to resolve an incident is rarely concentrated in one engineer. A single ML production incident may require an ML platform engineer to diagnose the Kubernetes GPU scheduler, a data engineer to trace a stale feature pipeline, and a modeling scientist to interpret whether a distribution shift is a bug or an expected concept drift. Traditional single-role on-call rotations are poorly suited to this because the symptom (a degraded recommendation metric) is entirely decoupled from the root cause domain (an upstream Kafka topic schema change). Mature ML organizations address this by structuring on-call as a tiered or domain-matrixed system: a primary responder owns initial triage and escalation, with clear escalation paths to data engineering, model development, and platform infrastructure as domain specialists. The primary's role is to route the incident to the correct domain within the first 15 to 30 minutes, not to be capable of resolving every failure class in isolation.

Rotation design determines whether the team can sustain that diagnostic burden. Table 12.41 lists guidelines that reflect industry practice. The pattern is bounded continuity with redundancy: short rotations limit burnout, secondary coverage handles multi-domain incidents, and handoff overlap preserves context.

[20] **Site Reliability Engineering (SRE)**: Founded by Ben Treynor Sloss at Google in 2003, SRE introduced error budgets and SLOs as quantitative reliability contracts. For ML systems, classical SRE must be extended: traditional services fail in binary (up/down), but models degrade probabilistically – accuracy can drop 5 percent over weeks without triggering any health check. This demands drift-aware SLOs that treat prediction quality alongside uptime as a reliability metric.

Table 12.41: On-Call Rotation Design: Industry-practice guidelines for rotation length, primary/secondary coverage, handoff overlap, and follow-the-sun handoffs in ML on-call. The recommendations balance context-switching cost (shorter rotations) against burnout risk (longer rotations) while ensuring expertise overlap for complex ML incidents.

Aspect	Recommendation
Rotation length	1 week (shorter causes context switching, longer causes burnout)
Primary + secondary	Always have backup; ML incidents often require multiple experts
Handoff overlap	30 min overlap for incident context transfer
Follow-the-sun	For global teams, hand off with timezone; 8-hour shifts maximum

Alert fatigue poses a persistent risk to on-call effectiveness. Signs include on-call engineers ignoring alerts, assuming false positives, increasing time to acknowledge, and alerts auto-resolving without investigation. Mitigation must reduce pages that do not lead to ML-specific action:

1. Tune drift, freshness, latency, and business-metric thresholds quarterly based on false positive rate.
2. Deduplicate related model, feature-store, serving, and infrastructure alerts into one incident page.
3. Attach runbooks that identify the likely owner and the first diagnostic query for each alert.
4. Track alert-to-action ratio; aim for >80 percent.

Beyond general SRE skills, ML on-call requires interpreting model quality metrics, understanding data pipeline dependencies, distinguishing model bugs from data drift, and making rollback vs. investigate decisions under pressure. Toil reduction is equally critical because recurring manual tasks consume the capacity needed to improve the platform. Track time spent on recurring manual tasks, targeting less than 25 percent of on-call time on toil.

Common ML toil usually accumulates at the model-platform boundary:

- Manually restarting failed training jobs
- Manually approving routine deployments
- Investigating alerts that require no action
- Generating recurring reports

Automation is the durable response to recurring toil. Every hour of automation development that saves 10 minutes per incident per on-call pays back within a quarter. Despite these operational tools, common misconceptions consistently lead engineering teams astray when attempting to scale their ML operations.

12.12 Fallacies and Pitfalls

Operating machine learning systems at scale involves counterintuitive complexity growth that causes common misconceptions. Engineers often assume that operational practices scale linearly with model count, when in reality the interactions between models create combinatorial complexity that demands fundamentally different platform architectures. These fallacies and pitfalls capture errors that waste millions in operational costs, cause cascading production failures across model fleets, and prevent organizations from deploying machine learning effectively beyond initial prototypes.

Fallacy: *Operational complexity grows linearly with model count.*

In production, complexity grows superlinearly due to inter-model dependencies: 100 models introduce dense dependency graphs where Model A depends on features from Pipeline B using embeddings from Model C, making isolated updates impossible. The monitoring burden alone is telling: 100 models with 10 metrics each at 5 percent false positive rate generate 14,400 false alerts daily. A platform supporting 40 models with per-model operational practices requires 1,600 engineer-hours monthly ($2.88M annually at $150/hour). Per-model CI/CD, monitoring, and deployment patterns do not compose at scale.

Pitfall: *Applying independent alerts to every model and metric.*

Comprehensive per-model alerting guarantees alert fatigue. Equation 12.11 establishes that for $N_{\text{tests}} = 1000$ independent tests (100 models $\times$ 10 metrics) at $\alpha_{\text{fp}} = 0.05$, $\Pr(\text{at least one false alert}) = 1 - 0.95^{1000} \approx 1.0$. Operators learn to ignore the noise, and genuine incidents disappear. The solution is hierarchical monitoring: business metrics trigger executive attention, portfolio metrics aggregate across related models, and model-specific metrics serve investigation rather than primary alerting.

Fallacy: *Platform investment can wait until the organization reaches 100+ models.*

Figure 12.2 shows platform ROI becomes positive at 20–50 models, not 100+. Technical debt from fragmented practices compounds rapidly: 40 models maintaining 847-line YAML files with no validation schema cause 35 percent of deployment delays; 23 preprocessing scripts with 62 percent duplication require 12 engineer-hours weekly to debug. By the time organizations reach 100 models, migration cost exceeds the cost of building the platform initially by 3–5$\times$. With 50 models requiring 40 hours monthly operational work each, a $2M platform investment pays for itself within 12 months.

Pitfall: *Treating all deployments with uniform procedures regardless of risk profile.*

Applying the same staged rollout to every model update either over-burdens low-risk changes or under-protects high-risk ones. Table 12.8 demonstrates that fraud detection requires hourly updates with seconds-fast rollback, while LLMs require monthly staged rollouts with hours-to-days rollback windows. A fraud model that cannot redeploy within one hour provides an exploitable window; an LLM deployed without multi-day shadow testing risks safety violations across millions of queries. Risk-based policies should match section 12.5 patterns: instant rollback for adversarial models, canary deployments for recommendations, shadow deployments for LLMs.

Fallacy: *Per-model metrics, such as loss and accuracy, are sufficient at scale.*

In multi-model platforms, system-level metrics matter more than individual model performance. A recommendation ensemble invoking 10–50 models can experience 30 percent latency degradation when one upstream retrieval model slows by 20 ms, even though all accuracy metrics remain nominal. Upstream embedding drift can degrade 12 downstream models simultaneously, a failure mode invisible in per-model accuracy tracking. Section 12.6 establishes that platform observability requires business metrics at the top, portfolio metrics for coordination, model metrics for investigation, and infrastructure metrics at the foundation.

Pitfall: *Defaulting to batch pipelines for all features to simplify architecture.*

Batch pipelines with daily feature updates ignore the quantitative impact of staleness. The freshness formula $T_{\text{freshness}} = T_{\text{available}} - T_{\text{event}}$ shows daily batch processing yields $T_{\text{freshness}} \approx 12$–$24$ hours; streaming pipelines achieve $T_{\text{freshness}} \approx 1$–$5$ seconds. The recommendation-system example in table 12.34 quantifies the engagement lift from fresher features, and for fraud detection, day-old features give adversaries a 24-hour exploitation window. A recommendation platform generating \$15M weekly with 10 percent from ML that improves 15 percent with real-time features gains \$225K weekly, easily covering streaming infrastructure costs.

Fallacy: *Technical debt is inevitable at scale and should be addressed only when it blocks critical work.*

The premise misunderstands technical debt economics. Section 12.2.3.2 establishes quantitative thresholds: deployment velocity exceeding 2 weeks (healthy: \$<\$1 day) indicates configuration complexity, incident rates exceeding 20 per 1,000 deployments (healthy: \$<\$5) indicate testing debt, and toil exceeding 50 percent of capacity (healthy: \$<\$20 percent) indicates automation debt. Monitoring debt alone, with mean-time-to-detect of 4.2 hours, costs \$50K per incident $\times$ 15 incidents yearly = \$750K annually. Organizations that treat debt as inevitable watch toil consume 70–80 percent of engineering capacity, creating a death spiral where teams can only maintain existing systems.

Pitfall: *Letting debt metrics remain qualitative until toil consumes the team.*

Teams that accept growing deployment times and expanding toil as the inevitable "cost of growth" miss the operational signal that the platform boundary has failed. Debt must be measured while it is still small enough to redirect: deployment lead time, incident frequency, alert noise, rollback delay, and toil share are all leading indicators. Recognizing these pitfalls completes the management layer of the AI fleet before the discussion moves to security.

12.13 Summary

ML operations at scale is the "nervous system" of the Machine Learning Fleet. The surrounding arc has moved from physical fleet foundations, to distributed training and serving mechanisms, to the operational and governance layers that keep global services reliable. This chapter developed the management layer required to sustain that architecture across hundreds of models and billions of devices.

The transition from managing a single model to operating an organizational platform represents a qualitative shift in complexity. Per-model operational practices do not compose; they create combinatorial debt that can only be resolved through platform abstractions like centralized registries, ensemble-aware CI/CD, and hierarchical monitoring.

Operational cadences must match model risk profiles, from the staged, weeks-long rollouts of LLMs to the seconds-fast rollbacks of adversarial fraud detection. The TCO framework ($\text{TCO}_{\text{ML}} = C_{\text{train}} + C_{\text{infer}} + C_{\text{data}} + C_{\text{iter}}$) provides quantitative foundations for strategic investment decisions, revealing how cost structure shifts from iteration-dominated (early stage) to inference-dominated (production scale) and how optimization priorities must evolve accordingly. Finally, the MLOps

vision extends to the edge, addressing the "Fleet Version Skew" and "Hardware-in-the-Loop" validation requirements essential for managing intelligence on millions of heterogeneous devices.

The central lesson of this chapter is that managing one model differs qualitatively from managing hundreds. A single model can be operated through manual processes, ad hoc monitoring, and bespoke deployment scripts. At organizational scale, these practices collapse under combinatorial weight: 200 models with independent CI/CD pipelines, individual alert configurations, and separate cost tracking create thousands of operational surfaces that no team can maintain. Platform abstractions are the only viable response, transforming per-model toil into shared infrastructure where each additional model incurs marginal rather than linear operational cost.

The practitioner who internalizes this lesson gains a strategic advantage. Understanding TCO economics enables quantitative arguments for infrastructure investment, demonstrating why a $2M platform investment can pay for itself within 12 months at 50 models by eliminating redundant pipelines and reducing incident response costs. Mastering hierarchical monitoring turns fleet-wide observability from an aspiration into an engineering discipline, surfacing cross-model failures that per-model dashboards cannot detect. Quantifying platform ROI in terms of deployment velocity, incident rates, and toil ratios provides the evidence that leadership requires before committing resources. Without these skills, operational debt accumulates silently until it paralyzes the organization, consuming engineering capacity in maintenance while competitors build the platforms that let them iterate faster.

Key Takeaways: Platforms, not pipelines

- **One model is not the unit**: At portfolio scale, the operational object becomes the dependency graph of models, features, pipelines, alerts, and owners. Registries, lineage, and ensemble-aware CI/CD prevent a local update from silently breaking downstream consumers.
- **Platforms turn toil marginal**: The summary's 50-model, $2M platform example shows why shared infrastructure pays back when repeated pipelines, incident response, and manual coordination are removed. The economic gain is not elegance; each added model costs less to deploy and maintain.
- **Ownership cost chooses the target**: The cost-of-ownership equation separates training, inference, data, and iteration costs, and the dominant term changes over a system's life. Early fleets should buy velocity; mature high-traffic fleets should buy serving efficiency, utilization, and cost attribution.
- **Monitoring must aggregate signal**: Per-model dashboards and alerts scale into noise when 100 models each emit independent failures. Hierarchical telemetry, fleet-wide anomaly detection, and feature-quality gates make common-cause incidents visible before alert fatigue hides them.
- **Operations reaches the edge**: Model fleets do not end at the data center. Weeks-long rollouts, hardware-in-the-loop validation, version skew, and heterogeneous device failures make edge deployment part of the same platform discipline that governs cloud CI/CD.

The surprising claim of this chapter is that operations does not scale, at least not for free. The reason is combinatorial: a fleet's burden is driven less by any one model than by the interactions among all of them, so effort grows faster than the model count unless something breaks that coupling. A platform is that something. By making the fleet rather than the model the unit that is built, watched, and paid for, it turns each new model from a fixed tax into a marginal one. Without it, the toil compounds until maintenance consumes the very capacity that was meant to build the next thing, and an organization can be at the technical frontier while paralyzed operationally.

What's Next: A governed fleet is still an attack surface

The operational machinery is now in place. Platform economics, fleet monitoring, edge deployment, and FinOps governance keep hundreds of models reliable and observable. Reliability and observability, however, say nothing about adversaries. Every model endpoint the fleet exposes can be probed for extraction or evasion, every shared pipeline can be poisoned, and every training corpus and telemetry stream is a privacy liability. Chapter 13 takes up that exposure, asking how a fleet defends its models, its data, and its users once it operates at scale.

Research Questions: For further inquiry

- How should an organization decide when platform investment is cheaper than continued per-model operations?
- How should platform abstractions preserve the C^3 constraints exposed by training, serving, edge, and fault tolerance rather than hiding them?
- What lineage and release-gate structure prevents one model, data, or feature change from silently breaking dependent systems?
- How should monitoring aggregate fleet-wide signal without creating alert fatigue or losing model-specific failures?

IV

THE RESPONSIBLE FLEET

Part IV

Responsible Fleet Principles

Parts I through III built the Machine Learning Fleet, coordinated its training, and optimized its deployment. Part IV establishes the **Responsible Fleet**: the engineering layer that determines whether the fleet serves its users safely or harms them. Security, privacy, robustness, and sustainability are engineering constraints with the same physical and mathematical force as bandwidth, power, or latency.

The hardest engineering domain in the fleet is the sociotechnical feedback loop. Bias cannot be fixed, and a model cannot be secured, with a single algorithm. Responsible operation requires the monitoring, verification, and governance systems that surround the fleet. A system that ignores these constraints fails operationally: through regulatory shutdown, security breach, or environmental exhaustion. These principles define the boundaries of responsible engineering at scale.

The first boundary is what the fleet can reveal.

Principle 17: The Information Leakage Invariant

Invariant: Every model output potentially leaks information about its training data. Perfect privacy is mathematically impossible if the model remains useful.

$$I(\text{TrainingData}; \text{ModelOutput}) > 0$$

Here, $I(\cdot;\cdot)$ denotes mutual information between the training data and observable model outputs.

Implication: Privacy is a budget, not a switch. Anonymization cannot be applied after the fact once training data has been memorized into the weights. Systems requiring formal privacy guarantees should use mechanisms such as differential privacy (DP) where appropriate: DP quantifies bounded privacy loss for a data analysis or training procedure, and interactive query systems must account for composition and may stop answering once the system exhausts the allocated privacy budget.

Privacy defines what the fleet may reveal; robustness defines how reliably it behaves when inputs are chosen to exploit its weaknesses.

Principle 18: The Robustness Compute Penalty

Invariant: Achieving intrinsic adversarial robustness often requires training on perturbations; Projected Gradient Descent (PGD)-style adversarial training can require roughly 5–10$\times$ more training compute because each batch runs several inner attack steps.

Implication: There is no "free" robustness. Building secure models is computationally expensive. For many applications, it is more efficient to rely on external guardrails (input filtering, output verification) than to train intrinsic robustness into the model weights.

Efficiency itself becomes a trap at fleet scale.

Principle 19: The Jevons Paradox of AI (Efficiency Trap)

Invariant: Improvements in efficiency that lower the cost of a resource will tend to increase, rather than decrease, the total consumption of that resource.

$$\text{Efficiency} \uparrow \implies \text{Cost} \downarrow \implies \text{Demand} \uparrow\uparrow$$

Implication: Making models $10\times$ more efficient can increase total usage enough to erase or even exceed the expected energy savings. Sustainability strategies must focus on absolute limits (carbon budgets, renewable sourcing) rather than rate efficiency (FLOP/s per watt) alone.

Responsible operation also requires making social objectives explicit, because statistical guarantees can conflict even when every metric is well defined.

Principle 20: The Fairness Impossibility Law

Invariant: For nonperfect classifiers operating on groups with different base rates, calibration, equalized odds, and demographic parity cannot be satisfied simultaneously.

$$\Pr(Y = 1 \mid A = a) \neq \Pr(Y = 1 \mid A = b) \implies \text{Trade-off Required}$$

Here, Y is the outcome and $A \in \{a, b\}$ is the group attribute.

Implication: Fairness is a constraint satisfaction problem with no global optimum. Engineers must treat fairness metrics like latency budgets: explicit trade-offs chosen by stakeholders, enforced by the system, and monitored for violation. See Chapter 16 for the full treatment of when the impossibility result applies, including the trivial-classifier edge cases.

Those trade-offs cannot be evaluated once and frozen, because deployed systems alter the data and incentives they later observe.

Principle 21: The Sociotechnical Feedback Invariant

Invariant: Deployed models shape the environment they operate in. The probability distribution of future data $p_{t+1}(X)$ is a function of the model's past decisions $f_t(X)$.

$$p_{t+1}(X) = g(p_t(X), f_t(X))$$

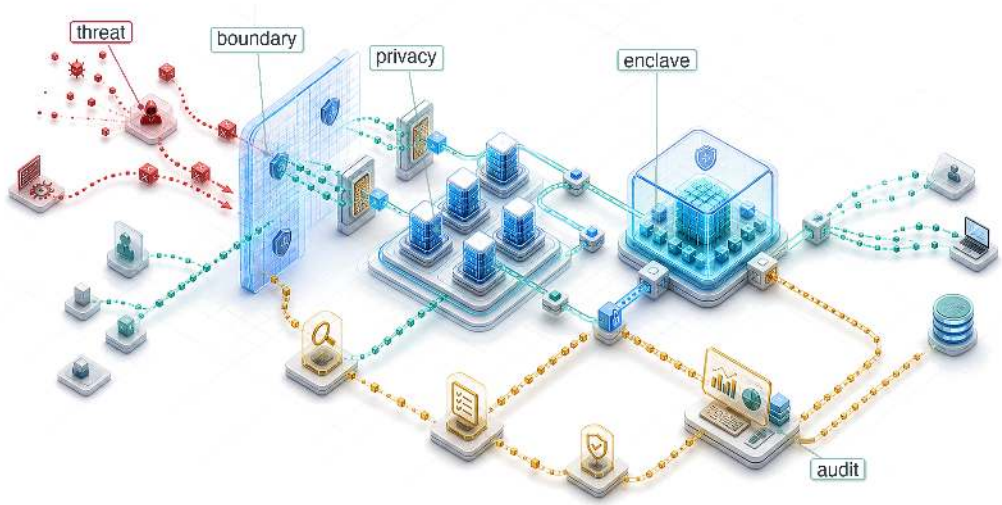
Here, g represents the environment's response function, which maps the current data distribution and model decisions into the next data distribution.

Implication: Systems require closed-loop governance. A model that maximizes accuracy on static test data can still degrade the future data distribution it operates on, amplify incentives that bias the next round of data, or destabilize the environment being monitored. Reliability requires modeling the *feedback loop*, not the *feed-forward inference* alone.

Part IV follows the same constraint-driven logic as the rest of the volume. Security and privacy harden the fleet against attack and protect training data from leakage. Robustness asks whether model performance remains stable when inputs, environments, or adversaries change. Sustainability treats planetary-scale infrastructure as an energy and carbon system, not only a performance system. Responsible AI then connects those technical controls to institutional governance. Together, these chapters complete the engineering discipline that turns a capable fleet into a trustworthy one.

13

Security & Privacy

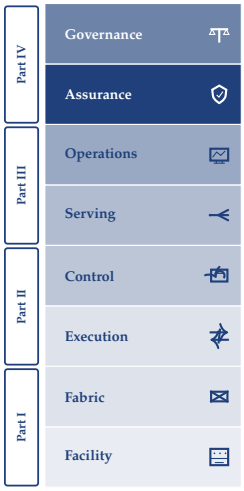


13.1 The Expanded Attack Surface
13.2 Learning from Security Breaches
13.3 Systematic Threat Analysis and Risk Assessment
13.4 Model-Specific Attack Vectors
13.5 Hardware-Level Security Vulnerabilities
13.6 When ML Systems Become Attack Tools
13.7 Comprehensive Defense Architectures
13.8 Differential Privacy
13.9 Security and Privacy Maturity Model
13.10 Fallacies and Pitfalls
13.11 Summary

Purpose

Why do privacy and security determine whether machine learning systems achieve widespread adoption and societal trust?

Many high-utility machine learning systems depend on personal data, institutional knowledge, or behavioral patterns, creating tension between utility and protection that determines societal acceptance. Unlike traditional software that processes data transiently, ML systems learn from sensitive information and embed patterns into persistent models that can inadvertently reveal private details. This capability creates systemic risks extending beyond individual privacy violations to threaten institutional trust, competitive advantages, and democratic governance. A high-performing model remains unused if it cannot be deployed without exposing sensitive data, if it cannot be trusted to resist adversarial manipulation, or if it cannot satisfy regulatory requirements that govern its intended domain. Privacy and security are not features to be added after the system works but prerequisites that determine whether the system can work at all in contexts where data sensitivity and adversarial risk are nonnegotiable constraints. In C^3 terms, security and privacy change Compute, Communication, and Coordination contracts: throughput is no longer sufficient unless each boundary also preserves confidentiality, integrity, and accountable control.



Learning Objectives

- Distinguish security from privacy in ML systems through formal definitions, threat models, and quantitative trade-offs
- Extract security principles from historical breaches (Stuxnet, Jeep Cherokee, Mirai) applicable to distributed ML infrastructure
- Analyze ML-specific attack vectors across model theft, data poisoning, adversarial examples, and hardware vulnerabilities
- Implement differential privacy with mathematical rigor, computing privacy budgets and accuracy trade-offs for production systems
- Design layered defense architectures spanning data protection, model security, runtime monitoring, and hardware trust mechanisms
- Evaluate hardware trust primitives for ML workloads with quantitative overhead analysis
- Apply a maturity model to build context-appropriate security architectures for specific threat models

13.1 The Expanded Attack Surface

Security and privacy are not afterthoughts. They are structural requirements that must be engineered into every layer of the distributed ML stack. The fleet stack makes that obligation explicit: after the fleet, distributed logic, and serving infrastructure become operational, the governance layer must protect the system so the global fleet cannot be hijacked, poisoned, or exploited by adversaries.

When a traditional database is breached, the attacker steals records. When a machine learning model is breached, the attacker may probe it for memorized training examples, or subtly poison the training data to introduce a silent backdoor that activates only on the attacker's chosen trigger. Machine learning systems fundamentally change the security landscape because they do not merely store data: they compress, memorize, and act upon it in ways that traditional deterministic software does not.

Operational platforms manage hundreds of models across distributed infrastructure, and this global reach creates an expansive attack surface. Distributed training systems, edge deployments, and multi-tenant serving platforms all introduce vulnerabilities absent in single-machine systems. Gradient synchronization protocols create channels for information leakage and manipulation. Federated aggregation exposes model updates to interception and inference attacks. Multi-tenant serving infrastructure presents opportunities for model extraction and side-channel attacks. Each architectural decision that enables scale also creates vulnerabilities requiring systematic defense.

The root cause is the difference between *transient processing* and *persistent learning*. Traditional software processes data deterministically and discards it; machine learning systems extract and encode patterns from training data into persistent model parameters. This learned knowledge representation creates vulnerabilities where sensitive information can be inadvertently memorized and later exposed through model outputs or systematic interrogation. Healthcare models may leak patient information through carefully crafted queries, while proprietary models can be reverse-engineered through strategic query patterns, threatening both individual privacy and organizational intellectual property.

Architectural complexity compounds these challenges. A contemporary ML deployment spans data ingestion pipelines, distributed training infrastructure, model serving systems, and continuous monitoring frameworks, each introducing distinct vulnerabilities as figure 13.1 maps across the ML lifecycle. Continuous adaptation at edge nodes and federated coordination protocols further expand the attack surface while complicating comprehensive security implementation.

Defense starts by separating two concerns that often share the same vocabulary. Security asks how an adversary can steal, manipulate, or disable the fleet; privacy asks how sensitive information can leak or be inferred even when the system behaves as designed. Keeping those questions separate determines which controls belong at each layer: authentication and isolation around adversarial

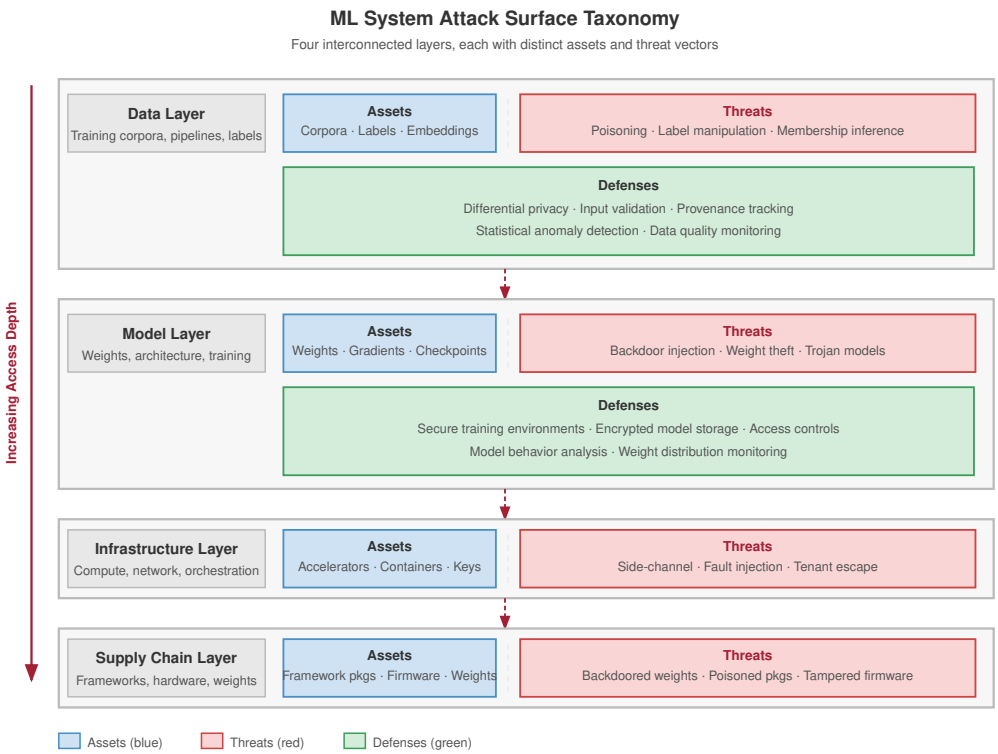


Figure 13.1: ML System Attack Surface: Visualizing entry points for adversarial actions across the ML lifecycle. Defense requires a multi-layered approach: protecting data collection (Data Layer), securing weights and training (Model Layer), hardening compute/network/orchestration (Infrastructure Layer), and validating frameworks, firmware, and hardware provenance (Supply Chain Layer).

pathways, provenance and monitoring across the release chain, and privacy accounting wherever learned parameters carry traces of the data that produced them.

Security and privacy are distinct concerns in machine learning system design that are often conflated. Both protect systems and data through different mechanisms, addressing different threat models and requiring distinct technical responses. Distinguishing between the two guides the design of responsible ML infrastructure.

13.1.1 Security defined

Security in machine learning focuses on defending systems from adversarial behavior. This includes protecting model parameters, training pipelines, deployment infrastructure, and data access pathways from manipulation or misuse.

> **Definition 13.1: Security**
>
> ***Security*** is the set of system properties (confidentiality, integrity, and availability) that protect an ML system's data, model weights, and inference pipeline from intentional adversarial actions, spanning both the infrastructure layer (network intrusion, credential theft) and the algorithmic layer (model extraction, prompt injection, adversarial examples).
>
> 1. **Significance**: Security failures operate on both surfaces simultaneously. At the infrastructure layer, theft of a large proprietary model represents direct IP loss. At the algorithmic layer, model extraction via black-box queries can recover decision boundaries or distill

a functional approximation of a deployed model at a small fraction of its training cost (section 13.4.1 quantifies the query budgets), bypassing the investment and competitive moat. Either failure collapses the business value of the O (model operations) term in the iron law.

2. **Distinction**: Unlike general robustness (which addresses stochastic distribution shift from unintentional environmental change), security addresses intentional adversarial threats where an attacker actively maximizes the probability of a targeted failure, requiring worst-case rather than average-case analysis.
3. **Common pitfall**: A frequent misconception is that traditional IT security (firewalls, access controls, encryption) adequately secures ML systems. ML introduces an algorithmic attack surface orthogonal to infrastructure: a properly authenticated API request containing an adversarial input or a prompt injection bypasses all network-layer defenses and manipulates model behavior through the model's own learned functions.

A facial recognition system deployed in public transit infrastructure, for example, may be targeted with adversarial inputs that cause it to misidentify individuals or fail entirely, representing a runtime security vulnerability that threatens both accuracy and system availability. The confidentiality-integrity-availability triad matters because each property breaks a different system contract: confidentiality protects data and model access, integrity protects the correctness of training and inference behavior, and availability protects the service from being disabled or degraded when it is needed.

13.1.2 Privacy defined

Security addresses adversarial threats; privacy focuses on limiting the exposure and misuse of sensitive information within ML systems. Privacy protections cover training data, inference inputs, and model outputs, preventing leakage of personal or proprietary information even when systems operate correctly and no explicit attack is taking place.

Definition 13.2: Privacy

Privacy is the protection of sensitive information from unauthorized disclosure, inference, and misuse across the ML lifecycle.

1. **Significance**: It limits the exposure risk of training data and user inputs. Privacy-preserving techniques (for example, differential privacy) typically introduce a utility-privacy trade-off: increasing privacy adds "noise" to the gradients, which can increase the total operations (O) required to reach a target accuracy.
2. **Distinction**: Unlike confidentiality (which focuses on access control), privacy in ML focuses on inference risks: the ability of an observer to reconstruct sensitive training samples from the model's outputs or weights.
3. **Common pitfall**: A frequent misconception is that removing names (de-identification) is sufficient for privacy. In reality, neural networks are correlation engines that can inadvertently memorize and leak unique snippets of sensitive data through high-dimensional patterns.

Privacy failures are not limited to raw records or explicit identifiers. Aggregated behavioral traces can reveal sensitive routines when the surrounding geography makes individuals and sites easy to infer. A rigorous formal guarantee for bounding participation inference is differential privacy, which adds calibrated noise so the output distribution changes only by a bounded amount when any one individual's record is added or removed. This limits what an adversary can infer about that individual's participation or contribution, but it comes at a measurable cost to accuracy.

Napkin Math 13.1: The cost of differential privacy

Problem: Consider computing the average salary of 1000 employees while guaranteeing privacy budget $\epsilon = 1$. The salaries range from $0 to $200,000. How much noise must the mechanism add?

Math:

1. **Sensitivity** (Δf): The maximum one person can change the sum is $200,000.
2. **Privacy Budget** (ϵ): 1.
3. **Laplace Noise Scale** (b): the standard mechanism adds symmetric random noise whose typical magnitude is the sensitivity divided by the privacy budget (section 13.8.1.2 develops the mechanism in full), so $b = \Delta f / \epsilon =$ $200,000 / 1 = $200,000.
4. **Impact on Mean**: The mean-estimation noise comes from noise added to the *sum* with magnitude approximately $200,000.
    - Noise per person (average) = $200,000 / 1000 = $200.

Systems insight: Protecting one outlier introduces a $200 error in the average. For a dataset of $n_{\text{records}} = 100$, the error grows to $2,000. Differential privacy can make estimates unstable for small n_{records}; it works best when the dataset is large enough that $1/n_{\text{records}}$ dampens the added noise. In an ML training context, the salary average maps directly to the gradient update computed across a mini-batch: the "sum" being protected is the sum of per-example gradients, the sensitivity is the maximum gradient norm any one training example can contribute (the clipping threshold in differentially private stochastic gradient descent, or DP-SGD), and the injected noise scales by the same $\Delta f / \epsilon$ ratio. Protecting one user's highly anomalous gradient from disproportionately shifting a model's decision boundary during federated learning is mathematically identical to protecting one outlier salary from skewing the reported mean.

Differential privacy quantifies the statistical cost of protecting individual records, but production ML platforms face a second, orthogonal cost axis. When multiple tenants share a GPU cluster, each tenant's data and model state must be isolated from every other tenant's execution context. That isolation requires partitioning physical resources (SRAM, cache lines, compute slices) rather than adding statistical noise, and the overhead is measured in lost throughput rather than added variance. Section C.2 collects the baseline per-accelerator throughput and partitioning figures against which this isolation tax is measured, so the reader can scale the loss to a specific fleet configuration.

Systems Perspective 13.1: The tax of secure multi-tenancy

A platform team hosts two models on a single H100 using Multi-Instance GPU (MIG) to provide hardware-level isolation. On a dedicated GPU, the model achieves 1,000 tokens per second. After enabling secure partitioning, it achieves 850 tokens per second. The gap between those two figures is the performance cost of security.

Isolation requires dedicated hardware resources (SRAM, cache) and adds context-switching overhead. The throughput loss is 1,000 tokens - 850 tokens = 150 tokens per second, an isolation tax of (150 tokens/1,000 tokens) = 15 percent.

Systems insight: Security is a capacity drain. Providing hardware partitioning for a hosted model costs 15 percent of raw GPU throughput in this example. In the Machine Learning Fleet, multi-tenancy is an economic necessity, but it is not free. Engineers must decide whether the data sensitivity justifies losing this share of fleet capacity. For public-facing APIs, this "Tax" is one price of reducing the risk that one tenant's prompt or activations leak into another tenant's execution context.

13.1.3 Security vs. privacy

Although they intersect in some areas such as encrypted storage, security and privacy differ in their objectives, threat models, and typical mitigation strategies. These security-privacy distinctions matter because the two domains optimize for different failure modes. Table 13.1 contrasts them across six dimensions, showing how their distinct goals shape the specific concerns and defenses practitioners must consider.

Table 13.1: Security-Privacy Distinctions: Machine learning systems require distinct approaches to security and privacy; security mitigates adversarial threats targeting system functionality, while privacy protects sensitive information from both intentional and unintentional exposure through data leakage or re-identification. This table clarifies how differing goals and threat models shape the specific concerns and mitigation strategies for each domain.

Aspect	Security	Privacy
Primary Goal	Prevent unauthorized access or disruption	Limit exposure of sensitive information
Threat Model	Adversarial actors (external or internal)	Honest-but-curious observers or passive leaks
Typical Concerns	Model theft, poisoning, evasion attacks	Data leakage, re-identification, memorization
Example Attack	Adversarial inputs cause misclassification	Model inversion reveals training data
Representative Defenses	Access control, adversarial training	Differential privacy, federated learning
Relevance to Regulation	Emphasized in cybersecurity standards	Central to data protection laws (for example, GDPR)

The distinction is useful only if it changes design behavior. The rest of the chapter treats security and privacy as coupled system constraints: access control, cryptography, trusted execution, and differential privacy solve different failure modes, but production deployments often need them composed so that protecting the model does not expose the data and protecting the data does not disable auditability.

13.1.4 Security-privacy interactions and trade-offs

Although security and privacy share common goals, they impose distinct and sometimes conflicting engineering constraints.

> Systems Perspective 13.2: The privacy-utility trade-off
>
> Security and privacy are deeply interrelated but not interchangeable. A secure system helps maintain privacy by restricting unauthorized access to models and data. Privacy-preserving designs can improve security by reducing the attack surface; minimizing the retention of sensitive data reduces the risk of exposure if a system is compromised.
>
> However, they can also be in tension. Techniques like differential privacy reduce memorization risks but may lower model utility. Similarly, encryption enhances security but may obscure transparency and auditability, complicating privacy compliance. Designers must reason about these trade-offs holistically.

Systems serving sensitive domains such as healthcare, finance, and public safety must simultaneously protect against both misuse and overexposure. The boundaries between these concerns determine whether a system is performant, trustworthy, and legally compliant; the breach histories that follow show how these theoretical tensions become concrete failures when they are ignored.

13.2 Learning from Security Breaches

A public heatmap built for runners became a military-intelligence leak when sparse GPS traces outlined movement around sensitive sites. That privacy failure belongs beside three landmark security breaches because each case turns an abstract control objective into an operational constraint: behavioral telemetry can identify people, supply-chain compromise can alter physical systems, weak

isolation can expose safety-critical control paths, and default credentials can turn cheap devices into attack infrastructure. Although most of these incidents did not target ML systems directly, every failure mode they demonstrated has a direct analog in training pipelines, inference APIs, edge deployments, and privacy-preserving data products.

War Story 13.1: The heatmap that revealed routines

Context: In late January 2018, Strava's global activity heatmap—an aggregated visualization built from the GPS traces of roughly 27 million users and about 1 billion uploaded activities—was presented as anonymized, high-level movement data (Russell 2018).

Failure mode: Nathan Ruser, an Australian undergraduate working with the Institute for United Conflict Analysts, noticed bright jogging tracks in the Syrian desert that lined up with forward US military positions and tweeted the discovery. In sparse environments, aggregation did not hide enough: the heatmap revealed the perimeters of secret bases, supply routes, and patrol patterns across Syria, Afghanistan, Iraq, and Niger, along with the daily routines of the personnel inside.

Consequence: Defense ministries reassessed how consumer telemetry could leak operational information, and Strava revised its product—improving privacy zones, restricting some heatmap visibility, and moving toward opt-in defaults for heatmap inclusion.

Systems lesson: Privacy failures can occur without names, passwords, or model weights leaking. High-dimensional location traces remain identifying when the context is sparse and an adversary can combine them with outside knowledge—and "aggregate" anonymization is no defense when the aggregate itself carries the structure of the underlying behavior.

13.2.1 Supply chain compromise: Stuxnet

In 2010, the Stuxnet[1] worm infiltrated Iran's Natanz nuclear facility by chaining four zero-day[2] exploits—a multi-million-dollar weapon—entering Windows systems through infected USB media[3] and then propagating to the Siemens Step7 software that programmed the air-gapped[4] programmable logic controllers (PLCs) (Farwell and Rohozinski 2011). Rather than crashing the centrifuges outright, it altered control parameters while reporting normal telemetry to operators, demonstrating that a system can be compromised while appearing healthy. The ML parallel is precise: an attacker who poisons training data or injects a backdoored model into a trusted repository does not need to crash the inference server; a silent shift in the model's decision boundary achieves the same effect while evading standard monitoring.

ML supply chains that ingest public packages, datasets, model weights, or firmware face four analogous vectors: compromised dependencies (malicious packages in PyPI and conda repositories), poisoned datasets on public platforms, backdoored model weights in model repositories, and tampered accelerator firmware. High-assurance deployments often combine cryptographic signing of model artifacts, immutable provenance logs for training data and code, automated scanning for backdoors before deployment, and controlled dependency management in air-gapped training environments. Figure 13.2 maps these parallels between the Stuxnet attack chain and ML supply chain vulnerabilities.

13.2.2 Insufficient isolation: Jeep Cherokee hack

Security researchers remotely compromised a Jeep Cherokee's engine, transmission, and braking systems by exploiting a vulnerability in the vehicle's internet-connected Uconnect entertainment system—without physical access to the car (Miller and Valasek 2015; Miller 2019). The architectural flaw was insufficient isolation: the entertainment system shared a network path with safety-critical CAN bus controllers. The incident triggered the first cybersecurity recall in automotive history, affecting 1.4 million vehicles[5], and prompted NHTSA[6] to issue non-binding vehicle cybersecurity best-practice guidance.

The ML lesson is direct: any deployment where an inference API shares a network path with safety-critical actuators—autonomous vehicle perception models, industrial IoT anomaly detectors, medical device diagnostic systems—inherits the same vulnerability class. Defense requires strict network

1 **Stuxnet**: First detected in 2010 by VirusBlokAda, a Belarusian antivirus firm, Stuxnet was the first publicly known malware engineered to cause physical destruction. Its use of four simultaneous zero-day exploits set a precedent for ML supply chain attacks: just as Stuxnet compromised industrial controllers through trusted software update channels, ML attacks can inject backdoored models through trusted repositories.

2 **Zero-Day** (from piracy slang for "zero days since release"): In security, the term denotes vulnerabilities with zero days of available defense. Stuxnet's simultaneous use of four zero-days was unprecedented; the black-market value of a single zero-day exploit exceeds $1 million, making a four-exploit chain a multi-million-dollar weapon with direct implications for ML systems where unpatched model-serving frameworks create analogous zero-day exposure windows.

3 **USB Attack Vector**: USB (Universal Serial Bus) became the primary vector for bridging air gaps after the 2008 Operation Olympic Games reportedly used infected drives to penetrate classified facilities. For ML deployments on isolated training clusters, USB-transferred datasets and model checkpoints represent the same class of risk: a single compromised checkpoint file can embed backdoors that survive across retraining cycles.

4 **Air-Gapped** (from the literal physical gap between network cables): Networks physically isolated from external connections, a practice dating to 1960s military computing. For ML systems, air-gapping training clusters from the internet prevents data exfiltration but forces all dependencies (frameworks, datasets, pretrained weights) through manual transfer channels, each of which becomes a potential supply chain attack vector.

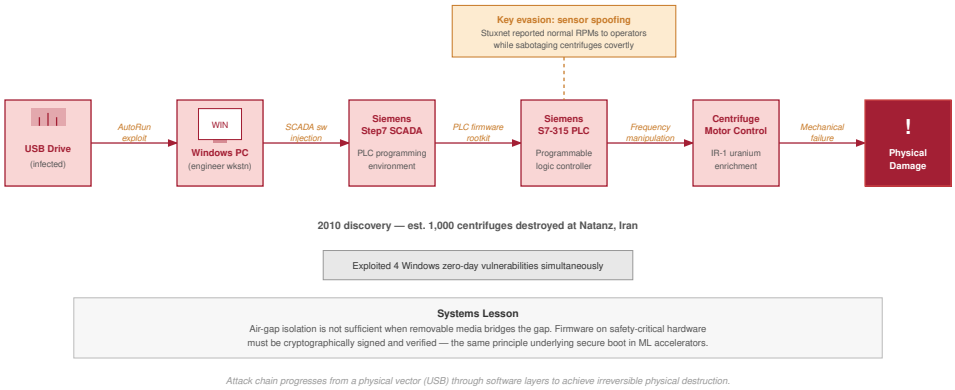


Figure 13.2: Stuxnet: Targets PLCs by exploiting Windows and Siemens software vulnerabilities, demonstrating supply chain compromise that enabled digital malware to cause physical infrastructure damage. ML systems face analogous risks through compromised training data, backdoored dependencies, and tampered model weights.

segmentation between inference and control planes, cryptographic API authentication, sandboxed model execution with minimal system privileges, and fail-safe defaults that revert actuators to safe states when ML components detect anomalies or lose connectivity.

13.2.3 Weaponized endpoints: Mirai botnet

In 2016, the Mirai botnet[7] compromised over 600,000 IoT devices—cameras, DVRs, and routers deployed with factory-default credentials—and directed them in a 1.2 Tbps DDoS[8] attack that disrupted major internet infrastructure across the United States (Antonakakis et al. 2017). The attack demonstrated a quantitative threshold: when even a small fraction of networked devices ship with default passwords, the aggregate becomes weaponizable infrastructure.

For ML edge deployments, the threat is amplified. Compromised ML devices offer capabilities beyond raw bandwidth: smart cameras can exfiltrate facial recognition databases, voice assistants can extract conversation transcripts, and any device participating in federated learning becomes a source of poisoned training data. Defense requires zero-trust edge security: device-unique keys via hardware security modules (HSMs), secure boot with cryptographic verification, TLS 1.3 or an organization-approved current equivalent for ML API communications, and behavioral monitoring to detect anomalous inference patterns.

Together, the Stuxnet, Jeep, and Mirai incidents establish a common structure: an attacker exploits a specific surface in the system pipeline—supply chain, network isolation boundary, or endpoint credentials—in a way the system's designers did not model as a threat. Security engineering turns those incidents into formal threat models for each surface in ML systems, attack economics that determine which threats are viable, and defenses whose costs can be quantified against threat severity.

13.3 Systematic Threat Analysis and Risk Assessment

Protecting a system whose attack surface includes every image it will ever see and every word it will ever process demands a fundamentally different approach than traditional cybersecurity. Network security and user authentication remain necessary, but ML systems introduce attack surfaces at the algorithmic layer: training data can be manipulated to embed backdoors, input perturbations can exploit learned decision boundaries, and systematic API queries can extract proprietary model knowledge.

Each of these vectors requires a formal **threat model** that specifies the adversary's capability (what they can access), the adversary's goal (what they seek to compromise), and the defender's information (what signals are observable). Systematic threat analysis maps these surfaces and

5 **Automotive Cybersecurity Recalls**: The 2015 Jeep Cherokee hack triggered the first-ever cybersecurity recall, affecting 1.4 million vehicles. The recall pattern mirrors ML model rollback: just as vehicles required over-the-air patches to isolate entertainment systems from safety-critical CAN buses, ML deployments require architectural isolation between external-facing inference APIs and safety-critical actuator control loops.

6 **NHTSA (National Highway Traffic Safety Administration)**: Established in 1970 and issuing its first cybersecurity guidance in 2016 post-Jeep hack, NHTSA publishes non-binding best-practice guidance on security-by-design for connected vehicles with 100+ onboard computers. This regulatory pattern is extending to ML systems: the EU AI Act (2024) imposes analogous lifecycle obligations for high-risk AI, including documented risk management and post-market monitoring.

quantifies the cost-benefit calculus that determines which threats are economically viable for an attacker to mount—and therefore which defenses are worth engineering.

A useful threat model has four fields: asset, boundary, adversary, and control. The asset defines what must remain confidential, intact, or available. The boundary defines where trust changes, such as between training data and model registry, user prompt and system instruction, or tenant workload and shared accelerator. The adversary defines capability and incentive. The control defines which signal, permission, or isolation mechanism changes the attacker's economics. Without those fields, a risk matrix degenerates into a list of fears rather than an engineering allocation tool.

13.3.1 Threat prioritization framework

Not all threats are equally likely or impactful, and security resources are always constrained. A prioritization matrix based on likelihood and impact turns the threat model into an allocation decision: automate defenses for threats that are both likely and damaging, prepare for rare but severe failures, and avoid spending scarce engineering time on low-value controls.

In this chapter, the example threats illustrate how the matrix changes engineering priority. High-likelihood/high-impact threats such as data poisoning in federated learning systems deserve automated defenses because untrusted training sources are common and the resulting model compromise can be severe. High-likelihood/low-impact threats such as model extraction against public APIs are also common and technically simple, but the primary consequence may be competitive loss rather than immediate safety or privacy failure, so rate limits and API design may be sufficient.

The lower-likelihood quadrants receive different treatment. Low-likelihood/high-impact threats such as hardware side-channel attacks on cloud-deployed models require specialized adversaries and physical or infrastructure access, but could expose all model parameters and user data, so they justify preparation in high-value deployments. In the public API scenario used for this matrix, membership inference attacks[9] may sit in a lower-likelihood/lower-impact quadrant than direct integrity failures. That placement is contextual rather than universal: when the training data is sensitive, overfit, federated, or regulated, membership inference becomes a higher-priority privacy threat and receives stronger controls.

The framework guides resource allocation, as figure 13.3 summarizes: in this example matrix, accessible threats such as model theft, data poisoning, and adversarial attacks come first, followed by more specialized hardware and infrastructure vulnerabilities. Implementing defenses in this sequence maximizes security benefit per invested effort for the assumed threat model.

13.3.2 Security threat modeling for ML systems

Systematic threat modeling identifies what must be protected and from whom. It is a structured approach to security analysis that characterizes the attack surface and guides defensive investments. For machine learning systems, threat modeling must account for dependence on training data, statistical decision boundaries, and distributed deployment patterns.

13.3.2.1 Attack surface analysis

The attack surface of an ML system encompasses all points where an adversary can interact with or observe the system. Unlike traditional software, where attack surfaces are primarily defined by input interfaces and network endpoints, ML systems expose attack surfaces across their entire lifecycle. The useful decomposition is by the layer where a defense can still act: data, model, interface, and infrastructure each expose distinct vulnerabilities and require different controls.

The training data pipeline represents a fundamental attack surface unique to learning systems. Adversaries can target five points where data or labels enter the learning process:

- **Collection endpoints**: Raw data enters the system through application logs, sensors, forms, APIs, or uploads.
- **Storage systems**: Training corpora sit in object stores, warehouses, feature stores, or dataset registries.
- **Preprocessing pipelines**: Transform jobs normalize, filter, augment, and join raw inputs before training.
- **Label generation**: Human annotation systems, weak-labeling rules, and model-assisted labeling create the targets the model learns from.

7 **Mirai Botnet** (Japanese for "future"): At its 2016 peak, Mirai controlled 600,000+ IoT devices generating 1.2 Tbps DDoS attacks by exploiting default credentials (admin/admin, root/12345). For ML edge deployments, the lesson is quantitative: every smart camera or voice assistant with default credentials is not merely a DDoS node but a potential source of poisoned training data in federated learning systems.

8 **DDoS (Distributed Denial-of-Service)**: Attack technique that overwhelms targets with traffic from multiple sources, first demonstrated in 1999, with reported attacks exceeding 3.47 Tbps in later infrastructure reports. For ML inference APIs, DDoS creates a dual threat: beyond service disruption, sustained high-volume queries can simultaneously function as model extraction attacks, harvesting enough input-output pairs to train a surrogate model while the defense team focuses on availability.

9 **Membership Inference Attacks**: First demonstrated against ML models by Shokri et al. (2017), these attacks determine whether a specific data point was used in training by exploiting the overfitting gap: models produce higher-confidence predictions on training data than on unseen data. Achieving 70–90 percent accuracy on many production models, they create General Data Protection Regulation (GDPR) and Health Insurance Portability and Accountability Act (HIPAA) compliance risks because confirming an individual's data was in the training set constitutes a privacy violation.

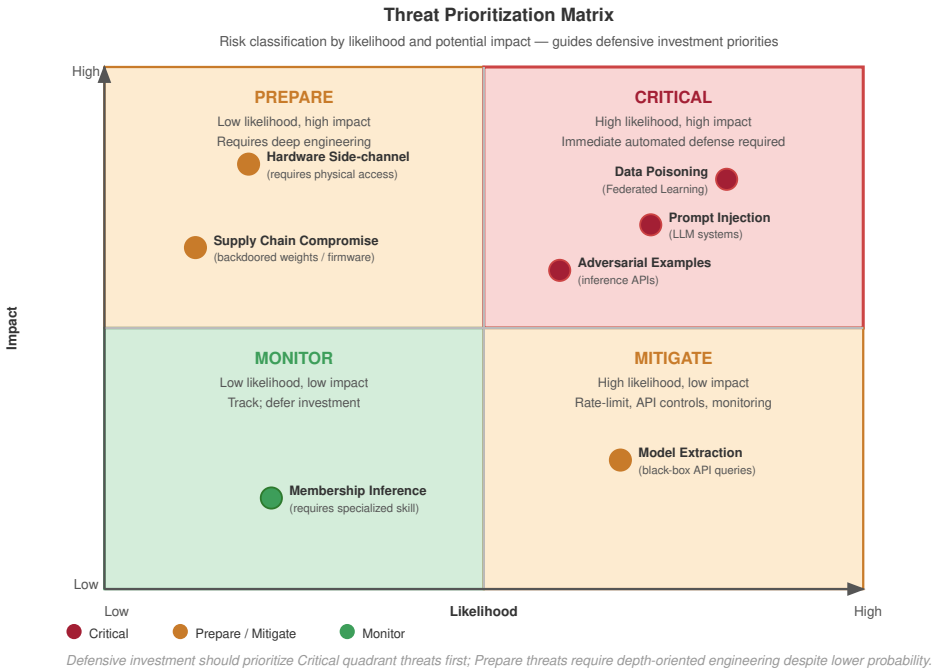
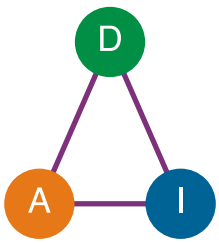


Figure 13.3: Threat Prioritization Matrix: A 2×2 matrix classifying ML threats by likelihood and impact. High-likelihood/high-impact threats (for example, data poisoning, prompt injection) deserve automated defenses. Low-likelihood/high-impact threats (for example, hardware side-channels) justify preparation in high-value deployments. High-likelihood/low-impact threats (for example, model extraction) are often addressed through rate limiting and API design.

- **Versioning and lineage systems**: Dataset manifests and provenance records decide which data snapshot becomes part of a release.

The data layer is particularly vulnerable because compromises here can embed persistent backdoors that survive model retraining. A single poisoned data source that enters the training pipeline can affect all subsequent model versions.

The model itself presents multiple attack surfaces spanning training infrastructure, model storage and versioning systems, model serialization and deserialization code, hyperparameter configuration management, and gradient computation and aggregation processes. Attacks at the model layer can compromise integrity by embedding trojans during training, extract intellectual property through parameter theft, or manipulate behavior through weight poisoning.



The attack surface spans data, learned model behavior, interfaces, and infrastructure.

Deployed models expose additional attack surfaces through inference API endpoints and load balancers, authentication and authorization systems, input validation and preprocessing logic, output formatting and response generation, and monitoring and logging infrastructure. The interface layer is where adversarial examples and model extraction attacks typically occur. Rate limiting, input validation, and output perturbation serve as primary defenses at this layer.

The underlying computational infrastructure presents traditional attack surfaces amplified by ML-specific concerns, including accelerator firmware and drivers, container orchestration and scheduling systems, network communication between distributed training nodes, key management and secrets infrastructure, and supply chain for ML frameworks and dependencies. Infrastructure compromises can affect all systems running on shared resources, making this layer critical for multi-tenant ML platforms.

13.3.2.2 Threat vector classification

Threat-vector classification matters because the same model failure demands different defenses when the attacker has only API access, partial design knowledge, or full insider access. The three axes are access type, knowledge level, and attack timing.

Access type determines how much information the defense must hide:

- **Black-box access**: The attacker has only input-output interaction with the model through APIs. This enables model extraction through prediction APIs (Tramèr et al. 2016) and query or transfer-based adversarial attacks (Nicolas Papernot, McDaniel, and Goodfellow 2016), but limits attack precision.
- **Gray-box access**: The attacker has partial knowledge such as model architecture, training procedure, or dataset characteristics. This enables more targeted attacks like transferable adversarial examples.
- **White-box access**: The attacker has complete knowledge of model parameters, architecture, and training data. This enables direct gradient-based attacks (Goodfellow et al. 2014) and, if model files are exposed, direct copying of weights.

The more transparent the system is to the attacker, the more the defense must rely on robust training, artifact protection, and runtime monitoring rather than obscurity.

Knowledge level determines how specialized the attack can be. Zero-knowledge adversaries operate without specific information about the target system, relying on generic attack techniques. Partial-knowledge adversaries possess information about the model family, training domain, or deployment context. Full-knowledge adversaries have complete information about the system including training data, model weights, and deployment configuration.

Timing determines where the defense can still act:

- **Training-time attacks**: Data poisoning or backdoor injection manipulates the learning process before the model is produced.
- **Deployment-time attacks**: Model distribution, serialization, or installation is compromised before serving begins.
- **Inference-time attacks**: The deployed model is exploited through adversarial inputs or extraction queries.
- **Postdeployment attacks**: Model updates, monitoring systems, or feedback loops are targeted after the model is already in service.

The defense surface therefore moves with the model lifecycle, from data controls to artifact integrity to serving-time containment.

The intersection of these dimensions defines specific threat scenarios. For example, a black-box, zero-knowledge, inference-time attack represents the common case of adversarial example generation against a public API. A white-box, full-knowledge, training-time attack represents the more severe case of an insider injecting backdoors during model development.

13.3.2.3 Defense strategy framework

Effective defense against the threat vectors identified by the threat model requires a layered strategy that addresses each attack surface while accounting for adversary capabilities. The defense framework operates on three principles: defense in depth, minimal attack surface, and fail-safe defaults.

No single defensive mechanism provides complete protection. The principle of **defense in depth** requires multiple independent layers such that compromising one layer does not grant full system access. For ML systems, this means combining data validation with model robustness techniques, access controls with output perturbation, and software defenses with hardware security mechanisms.

The principle of **minimal attack surface** starts from the fact that every exposed interface, stored artifact, and network endpoint represents a potential attack surface. Minimizing unnecessary exposure reduces risk through five mechanisms:

- **Restrict API capabilities**: Expose only essential functionality.
- **Limit output information**: Truncate confidence scores and other high-resolution outputs that leak model behavior.
- **Encrypt stored assets**: Protect models and training data at rest.
- **Isolate infrastructure**: Separate training systems from inference systems to prevent lateral movement.

- **Audit access**: Use strict permissions and logging to make sensitive actions attributable.

Minimal exposure does not eliminate attacks, but it reduces the information and movement available to an attacker after the first foothold.

The principle of **fail-safe defaults** requires systems to fail in ways that preserve security rather than availability when attacks succeed or anomalies occur. Four defaults make that posture concrete:

- **Reject suspicious inputs**: Do not process anomalous requests with reduced confidence as though they were ordinary traffic.
- **Halt unsafe training**: Stop training when data quality metrics degrade significantly.
- **Revoke risky credentials**: Revoke access tokens when unusual usage patterns appear.
- **Isolate compromised components**: Contain affected services to prevent lateral movement.

These defaults deliberately trade availability for containment when the system can no longer trust its inputs, data, or execution state.

Table 13.2 provides a concrete mapping from each attack surface layer to the defensive mechanisms and detection methods that protect it. Data defenses validate provenance, model defenses protect learned behavior and weights, API defenses govern query exposure, and infrastructure defenses anchor runtime trust.

Table 13.2: Defense Mapping by Attack Surface: Each layer of the ML system attack surface requires specific defensive mechanisms and detection methods. Effective security integrates protections across all layers while maintaining detection capabilities that can identify attacks that bypass preventive controls.

Attack Surface	Primary Threats	Defensive Mechanisms	Detection Methods
Data Layer	Poisoning, label manipulation, supply chain compromise	Input validation, provenance tracking, secure data pipelines	Statistical anomaly detection, data quality monitoring
Model Layer	Backdoor injection, parameter theft, trojan insertion	Secure training environments, encrypted model storage, access controls	Model behavior analysis, weight distribution monitoring
API/Interface Layer	Adversarial examples, model extraction, membership inference	Input sanitization, rate limiting, output perturbation, differential privacy	Query pattern analysis, confidence distribution monitoring
Infrastructure Layer	Side-channel attacks, firmware compromise, supply chain attacks	Trusted execution environments (TEEs), secure boot, network segmentation, dependency scanning	Hardware performance monitoring, integrity verification

The threat modeling framework provides the analytical foundation for the specific attack vectors and defensive techniques examined throughout the remainder of the chapter. Systematically analyzing attack surfaces, classifying threat vectors, and mapping defenses enables security architectures appropriate for specific threat models and risk tolerances.

Checkpoint 13.1: Knowledge check: Threat modeling

A team detects a high volume of API queries from a single IP address that are systematically exploring the decision boundary of a fraud detection model.

- ☐ **Lifecycle stage**: Identify whether the suspicious activity occurs during training, serving, or infrastructure operation.
- ☐ **Attack surface**: Identify the interface the attacker is exercising and infer what the query pattern reveals.
- ☐ **Defense mapping**: Select the rate-limiting, monitoring, or output-control mechanisms that should activate first.

Identifying whether an anomalous query pattern is a benign user or an active extraction attack is the crux of ML threat assessment. The structured threat model provides the analytical framework;

the next question is how specific attack vectors exploit each layer to compromise model integrity and confidentiality.

13.4 Model-Specific Attack Vectors

A stop sign with three strategically placed pieces of black tape is recognized by a human as a vandalized stop sign, but an autonomous vehicle's vision system might confidently classify it as a speed limit sign. This is not a software bug in the traditional sense; it is an adversarial example. The traffic-sign case study in this section makes the physical version concrete, but the lifecycle entry point determines the defense: training data, query interfaces, and runtime inputs demand different controls.

These attacks span the ML lifecycle and map directly to the threat model classifications we developed: threats to model confidentiality target deployment and inference stages through model theft, threats to training integrity strike during data collection and model development through poisoning attacks, and threats to inference robustness exploit runtime operations through adversarial examples[10]. Understanding when each attack occurs guides where to deploy corresponding defenses. Data poisoning[11] compromises the learning process itself, while model theft and adversarial attacks target the deployed system. Each category requires distinct defensive strategies aligned with the attack surface analysis in section 13.3.2.

Understanding when and where different attacks occur in the ML lifecycle helps prioritize defenses and understand attacker motivations. Figure 13.4 visualizes these stages and the specialized techniques adversaries use at each point, from data collection through model deployment and inference. The upstream threats compromise the learning process itself: attackers can inject malicious samples or manipulate labels during data collection, especially in federated learning or crowdsourced data scenarios where data sources are less controlled, and backdoor insertion during training can embed hidden behavior that activates only under specific trigger conditions. The downstream threats exploit the interfaces created by deployment: model theft becomes more attractive once trained models are accessible through APIs, file downloads, or reverse engineering of mobile applications, while adversarial attacks craft runtime inputs that fool deployed models into incorrect predictions while appearing normal to human observers.

The lifecycle perspective reveals that different threats require different defensive strategies. Data validation protects the collection phase, secure training environments protect the training phase, access controls and API design protect deployment, and input validation protects inference. Mapping attacks to lifecycle stages allows security teams to implement appropriate defenses at the right architectural layers.

The lifecycle view treats models as assets to protect, but it also exposes why those same assets can become part of an attack strategy. Pretrained models, particularly large generative or discriminative networks, may be adapted to automate tasks such as adversarial example generation, phishing content synthesis[12], or protocol subversion. Open-source or publicly accessible models can be fine-tuned for malicious purposes, including impersonation, surveillance, or reverse-engineering of secure systems. The chapter returns to that dual-use problem after establishing the basic threat classes, starting with attacks that steal model value.

13.4.1 Model theft

Definition 13.3: Model extraction

Model Extraction is the class of attacks in which an adversary, with only black-box query access to a deployed model's prediction interface, recovers the model's parameters, decision boundaries, or behavior by analyzing observable outputs.

1. **Significance**: Successful extraction often costs orders of magnitude less than the original training. Published demonstrations have recovered logistic regression, neural-network, and decision-tree models from public machine-learning APIs with high fidelity and modest query cost (Tramèr et al. 2016), and later work has reconstructed components of large language models, including OpenAI deployments studied by Carlini et al. (2024),

10 | **Adversarial Examples**: First discovered by Szegedy et al. (2013), these are inputs crafted to exploit learned decision boundaries with small perturbations that can be hard for humans to distinguish. The phenomenon reveals a fundamental tension in ML system design: the same high-dimensional feature spaces that enable generalization create exploitable geometry where small, targeted perturbations cross decision boundaries.

11 | **Data Poisoning**: First formally studied by Biggio et al. (2012), this attack injects malicious data during training to corrupt the learned model. The systems lesson is that training-data integrity matters even when the number of crafted examples is small: poisoning experiments against support vector machines showed that malicious training points can substantially increase test error by shifting the learned decision boundary.

12 | **AI-Generated Phishing**: Large language models (LLMs) generate phishing emails with 99 percent+ grammatical accuracy vs. 19 percent for traditional phishing, achieving 30 percent+ click-through rates in some campaigns. This dual-use threat illustrates why ML security must treat models as both assets to defend and potential weapons: the same language fluency that powers customer-facing chatbots can be fine-tuned for social engineering at scale.

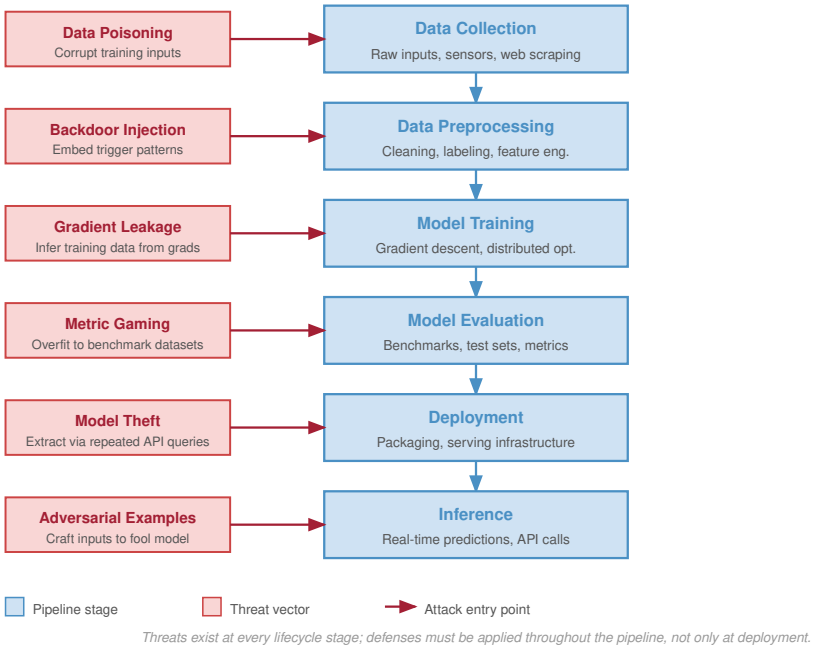


Figure 13.4: ML Lifecycle Threats: Model theft, data poisoning, and adversarial attacks target distinct stages of the machine learning lifecycle (from data ingestion to model deployment and inference), creating unique vulnerabilities at each step. Understanding these lifecycle positions clarifies attack surfaces and guides the development of targeted defense strategies for robust AI systems.

using only public query access. The chapter's BERT extraction example in this section illustrates the same dynamic on a transformer model at a query cost of tens of dollars.

2. **Distinction**: Unlike direct theft of model files (which requires intrusion into the storage layer) and model inversion (which uses model outputs to reconstruct samples of the training data rather than the model itself), extraction's attack surface is the model's query interface, which makes it a threat every deployed ML service faces by construction whenever the model is reachable.
3. **Common pitfall**: A frequent misconception is that rate limiting alone is a sufficient defense. An attacker willing to issue queries slowly, distribute requests across accounts, or accept a multi-month attack window can still extract a model unless the API also reduces output information per query, detects systematic probing, and prices queries to make extraction economically irrational.

The first category of model-specific threats targets confidentiality. Threats to model confidentiality arise when adversaries gain access to a trained model's parameters, architecture, or output behavior (Oliynyk et al. 2023). These attacks can undermine the economic value of machine learning systems, allow competitors to replicate proprietary functionality, or expose private information encoded in model weights.

Such threats arise across a range of deployment settings, including public APIs[13], cloud-hosted services, on-device inference engines, and shared model repositories[14]. Prediction interfaces can leak enough signal to recover model functionality (Tramèr et al. 2016; Oliynyk et al. 2023), while classifier outputs can reveal meaningful information about training sets or learned structure (Ateniese et al. 2015). Insecure serialization formats[15] and insufficient access controls create a separate artifact-security risk: they can expose the model file or deployment package directly, rather than only its observable behavior.

13 **ML APIs (Application Programming Interfaces)**: Popularized by Google's Prediction API (2010), large-scale ML APIs can handle billions of requests. Each API response leaks information: confidence scores, logits, and top-k predictions collectively form a side channel that enables model extraction. Reducing output verbosity (truncating logits, omitting confidence scores) directly trades API utility for extraction resistance.

14 **Model Repositories**: Centralized platforms for sharing ML models, led by large catalogs such as Hugging Face. These repositories create the same supply chain risk as package managers like PyPI: researchers have found models with embedded backdoors and arbitrary code execution payloads, making cryptographic verification of model provenance as critical for ML as package signing is for software.

The severity of these threats is underscored by high-profile legal cases that have highlighted the strategic and economic value of machine learning models. For example, former Google engineer Anthony Levandowski was accused of stealing proprietary designs from Waymo, including critical components of its autonomous vehicle technology, before founding a competing startup (The New York Times 2017). Such cases illustrate the potential for insider threats to bypass technical protections and gain access to sensitive intellectual property.

The consequences of model theft extend beyond economic loss. Stolen models can be used to extract sensitive information, replicate proprietary algorithms, or enable further attacks. The economic impact can be substantial: research estimates suggest that aspects of large language models can be approximated through systematic API queries at costs orders of magnitude lower than original training, though full model replication remains economically and technically challenging (Tramèr et al. 2016; Carlini et al. 2024). For instance, a competitor who obtains a stolen recommendation model from an e-commerce platform might gain insights into customer behavior, business analytics, and embedded trade secrets. This knowledge can also be used to conduct model inversion attacks[16], where an attacker attempts to infer private details about the model's training data (Fredrikson et al. 2015).

In a model inversion attack, the adversary queries the model through a legitimate interface, such as a public API, and observes its outputs. By analyzing confidence scores or output probabilities, the attacker can optimize inputs to reconstruct data resembling the model's training set. For example, a facial recognition model used for secure access could be manipulated to reveal statistical properties of the employee photos on which it was trained. Similar vulnerabilities have been demonstrated in studies on the Netflix Prize dataset[17], where researchers inferred individual movie preferences from anonymized data (Narayanan and Shmatikov 2006).

Figure 13.5 separates two theft objectives that require different controls. **Exact artifact theft** is a storage, registry, or device-security failure: the attacker reaches the model file or deployment artifact, extracts weights, architecture, or hyperparameters, and can reconstruct the original model without paying the training cost. **Approximate behavioral theft** is an interface-security failure: the attacker queries a deployed API, records labels, logits, embeddings, or confidence scores, and trains a surrogate model that approximates the original behavior. Exact theft calls for artifact signing, encryption, access control, and secure deployment paths; approximate theft calls for output limiting, rate controls, query auditing, watermarking, and pricing that make extraction economically unattractive. The specific architectural vulnerabilities vary by model type, with deeper networks and attention-based architectures presenting different attack surfaces than simpler convolutional or recurrent designs.

The approximate behavioral extraction path in figure 13.5 becomes practical because black-box query cost can be much lower than the cost of training the original model.

Example 13.1: The BERT model extraction

Context: Researchers demonstrated that proprietary models behind APIs are vulnerable to functional extraction.

Setup: By querying a victim BERT-based API with 2 million carefully crafted inputs (costing roughly $50 in query fees), they trained a student model that achieved more than 97 percent agreement with the victim on test tasks (Krishna et al. 2020). This model-stealing attack exploited the high-information signal returned by confidence scores and logits.

Systems lesson: API access can be enough to replicate intellectual property when outputs expose too much decision-boundary information. Defenses such as rate limiting, query auditing, and output truncation are part of the serving contract, not optional perimeter controls.

13.4.1.1 Exact model theft

Exact artifact theft targets the internal structure and learned parameters of a model. These attacks reach deployed models exposed through APIs, embedded in on-device inference engines, or shared as downloadable model files on collaboration platforms. Exploiting weak access control, insecure

15 **Model Serialization**: The process of converting trained models into portable formats—Open Neural Network Exchange (ONNX) 2017, SavedModel 2016, PyTorch .pth. Python's pickle-based serialization, common in PyTorch checkpoint workflows, can execute arbitrary code on deserialization, making every untrusted .pth file a potential remote code execution vector. Safer formats like SafeTensors (2022) eliminate code execution by storing only tensor data (Hugging Face 2026).

16 **Model Inversion Attack**: First demonstrated by Fredrikson et al. (2015) against facial recognition, where researchers reconstructed recognizable faces from confidence scores alone. The attack proved that black-box API access is not sufficient privacy protection: any model that returns rich output signals (probabilities, embeddings, attention weights) provides an optimization target for reconstructing training data.

17 **Netflix Deanonymization**: Researchers showed that as few as 8 movie ratings with approximate dates uniquely identify 99 percent of records in the anonymized Netflix Prize dataset; correlation with public IMDb ratings then enabled actual re-identification of users with public ratings (Narayanan and Shmatikov 2008). Netflix canceled a planned second competition. The lesson for ML systems: any dataset rich enough to train useful models contains enough structure for re-identification, making naive anonymization insufficient for privacy.

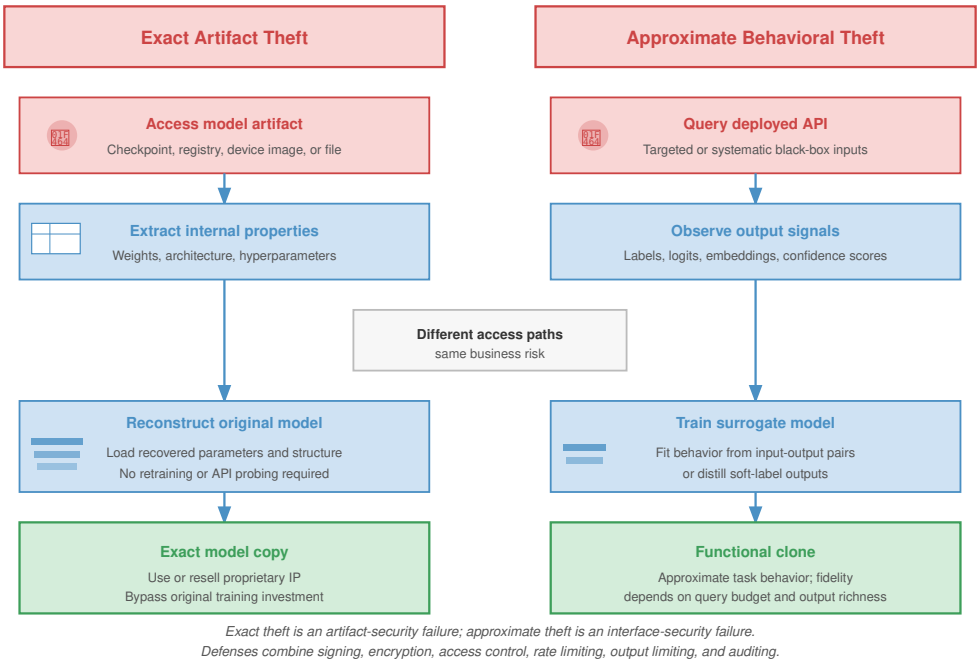


Figure 13.5: Model Theft Strategies: Exact artifact theft (left) obtains the model file, checkpoint, registry entry, or device artifact and reconstructs the original model. Approximate behavioral theft (right) uses black-box API queries, output signals, and surrogate training or distillation to clone the model's behavior without direct access to weights.

model packaging, or unprotected deployment interfaces, attackers can recover proprietary model assets without requiring full control of the underlying infrastructure.

These attacks typically seek three types of information because each one reduces the attacker's cost of replication:

- **Learned parameters**: Weights and biases allow the attacker to reproduce the model's functionality without paying the original training cost.
- **Fine-tuned hyperparameters**: Learning rate, batch size, and regularization settings reduce the experimentation needed to recover the same quality.
- **Architecture details**: Layer sequences, activation functions, and connectivity patterns reveal the design choices that create the model's behavior and may be recovered through side-channel attacks[18], reverse engineering, or analysis of observable outputs.

The more of this information an API or artifact reveals, the less work remains for the attacker.

System designers must account for these risks by separating artifact security from interface security. Artifact controls secure model files, registry entries, and deployment packages; interface controls restrict APIs, reduce output precision, audit query patterns, and price or rate-limit access so behavioral extraction becomes harder to perform economically (Tramèr et al. 2016; Oliynyk et al. 2023).

13.4.1.2 Approximate model theft

Approximate behavioral theft recreates a model's decision-making capabilities without touching its parameters or architecture. Instead, attackers observe the model's inputs and outputs to build a substitute model that performs similarly on the same tasks.

This type of theft often targets models deployed as services, where the model is exposed through an API or embedded in a user-facing application. By repeatedly querying the model and recording its responses, an attacker can train their own model to mimic the behavior of the original. This process, often called model distillation[19] or knockoff modeling, allows attackers to achieve comparable

18 | **ML Side-Channel Attacks**: First demonstrated against neural networks in 2018, when researchers showed that power consumption patterns during inference reveal model architecture (layer count, activation functions, parameter sizes). Unlike cryptographic side channels that leak keys, ML side channels leak intellectual property: an attacker with physical proximity to an edge device can reconstruct the model architecture without any API access.

19 | **Model Distillation**: Knowledge transfer technique where a smaller "student" model learns from a larger "teacher" model's soft probability outputs rather than hard labels (Hinton et al. 2015). Originally designed for compression (achieving 95 percent+ teacher accuracy with 10–100× fewer parameters), distillation becomes an attack when applied to API outputs: an adversary trains a local student on the victim's responses, effectively stealing the model's learned behavior without accessing its weights.

functionality without access to the original model's proprietary internals (Orekondy et al. 2019).

Attackers may evaluate the success of behavior replication in two ways. The first is by measuring the level of effectiveness of the substitute model. This involves assessing whether the cloned model achieves similar accuracy, precision, recall, or other performance metrics on benchmark tasks. By aligning the substitute's performance with that of the original, attackers can build a model that is practically indistinguishable in effectiveness, even if its internal structure differs.

The second is by testing prediction consistency. This involves checking whether the substitute model produces the same outputs as the original model when presented with the same inputs. Matching both the correct predictions and the original model's mistakes provides attackers with a high-fidelity reproduction of the target model's behavior. This poses particular concern in applications such as natural language processing, where attackers might replicate sentiment analysis models to gain competitive insights or bypass proprietary systems.

Approximate behavior theft proves challenging to defend against in open-access deployment settings, such as public APIs or consumer-facing applications. Limiting the rate of queries, detecting automated extraction patterns, and watermarking model outputs are among the techniques that can help mitigate this risk. However, these defenses must be balanced with usability and performance considerations, especially in production environments.

One demonstration of approximate model theft extracts internal components of black-box language models via public APIs. In their paper, Carlini et al. (2024), researchers show how to reconstruct the final embedding projection matrix of OpenAI's `ada` and `babbage` models, and to recover the hidden dimensionality (and estimate the cost of full matrix recovery) for `gpt-3.5-turbo`, using only public API access. By exploiting the low-rank structure of the output projection layer and making carefully crafted queries, they recover the model's hidden dimensionality and, for the smaller models, replicate the weight matrix up to affine transformations.

The attack does not reconstruct the full model, but reveals internal architecture parameters and sets a precedent for future, deeper extractions. This work demonstrated that even partial model theft poses risks to confidentiality and competitive advantage, especially when model behavior can be probed through rich API responses such as logit bias and log-probabilities.

The empirical results in table 13.3 demonstrate extraction of smaller models' output-projection parameters with root mean square errors as low as 10^{-4}, while the larger GPT-3.5 rows show dimension recovery and estimated recovery costs rather than implemented weight-matrix extraction. These findings raise important implications for system design, suggesting that innocuous API features, like returning top-k logits, can serve as significant leakage vectors if not tightly controlled.

Table 13.3: Model Stealing Costs: Attackers can recover internal model information with publicly available APIs. The Cost column gives two figures: the dimension-recovery cost and the full weight-matrix extraction cost. For OpenAI's ada and babbage models, implemented output-projection weight extraction achieves low root mean squared error (RMSE) with fewer than $4 \cdot 10^6$ queries at estimated costs of $1 to $12. The GPT-3.5 rows report dimension-recovery results and estimated full-recovery costs, not implemented weight-matrix extraction.

Model	Size (Dimension Extraction)	Number of Queries	RMS (Weight Matrix Extraction)	Cost
OpenAI ada	1024 ✓	$< 2 \times 10^6$	$5 \cdot 10^{-4}$	$1 / $4
OpenAI babbage	2048 ✓	$< 4 \times 10^6$	$7 \cdot 10^{-4}$	$2 / $12
OpenAI babbage-002	1536 ✓	$< 4 \times 10^6$	Not implemented	$2 / $12
OpenAI gpt-3.5-turbo-instruct	Not disclosed	$< 4 \times 10^7$	Not implemented	$200 / $2,000 (estimated)
OpenAI gpt-3.5-turbo-1106	Not disclosed	$< 4 \times 10^7$	Not implemented	$800 / $8,000 (estimated)

13.4.1.3 Defenses against model extraction

Model extraction attacks exploit the input-output behavior of deployed models accessed through APIs. The durable defense question is not which mechanism sounds strongest, but how much information the API lets one actor accumulate. Every response leaks some bits about the decision boundary, confidence surface, architecture, or training distribution. A production defense therefore has four levers:

- **Reduce information per response**: Return less precise confidence, fewer classes, and no unnecessary internal state.
- **Limit response count**: Use quotas and rate limits to cap how many observations one actor can collect.
- **Raise marginal cost**: Price high-volume collection so extraction becomes less attractive than training or licensing.
- **Detect measurement behavior**: Identify query patterns that look like systematic model probing rather than product use.

These levers work together because extraction economics depend on bits per query, query volume, price, and attacker visibility.

Monitoring provides the evidence for that control loop. Legitimate users usually produce bursty, task-shaped traffic, while extraction attacks need sustained and systematic coverage of the input space. A simple image-classification service might begin with a volume rule:

$$\text{alert}(\text{user}) = \begin{cases} 1 & \text{if } q_{\text{daily}} > 10{,}000 \text{ or } q_{\text{hourly}} > 2{,}000 \\ 0 & \text{otherwise} \end{cases}$$

where q_{daily} and q_{hourly} represent query counts. Such thresholds are only the first signal. They must be calibrated by service tier and by the shape of normal use, because batch customers and attackers can both generate high volume.

The next signal asks whether the inputs look natural. Extraction traffic often uses synthetic or systematically generated examples that cover decision boundaries more evenly than ordinary user queries. A detector can compare the user's query distribution with the expected distribution:

$$\mathcal{D}_{\text{KL}}(p_{\text{user}} \| p_{\text{expected}}) > \tau_{\text{alert}}$$

where $\mathcal{D}_{\text{KL}}$ is Kullback-Leibler divergence and τ_{alert} is an alert threshold. For a language model API, the same idea appears as unusual token distributions, repetitive prompt templates, or low-variance probing of model boundaries. Temporal features add another clue: automated extraction tends to have regular inter-query intervals and long uninterrupted sessions, while human or product traffic carries more variance.

An anomaly detector combines these signals into a score that can drive enforcement:

$$\text{score}_{\text{anomaly}}(u) = f_\theta(\text{qps}(u), \mathcal{D}_{\text{KL}}(u), \sigma_{\text{inter-query}}(u), \ldots)$$

where f_θ is a trained classifier and the features capture query volume, distribution, and temporal regularity. The score should not merely create an alert queue. It should feed the quota system, because rate limiting is how the service turns evidence into an information cap. Free users can receive low daily and burst limits, authenticated paid users can receive larger quotas, and enterprise customers can receive custom limits tied to contracts and monitoring. The point is not to punish volume by itself; it is to prevent anonymous or weakly authenticated actors from collecting enough observations to reconstruct the model.

A concrete quota ladder makes that information cap visible. A free tier might allow 1,000 queries/day with a 10 queries/s burst, a basic paid tier might allow 100,000 queries/day with 100 queries/s, and an enterprise tier might allow 10M queries/day with custom burst limits and contractual monitoring. The exact numbers depend on product needs, but the ladder matters because extraction cost scales with accumulated responses.

Adaptive limits connect the two pieces. Users with high anomaly scores face stricter limits, while verified legitimate users keep enough capacity for batch prediction:

$$\text{limit}_{\text{effective}}(u) = \text{limit}_{\text{base}} \times \exp(-\gamma_{\text{rate}} \cdot \text{score}_{\text{anomaly}}(u))$$

where γ_{rate} controls sensitivity. With $\gamma_{\text{rate}} = 1$, a user with anomaly score 0.8 keeps about 45 percent of the base quota, a 55 percent rate reduction; low scores preserve normal usage. This control works only if the response itself is also designed carefully. A permissive API that returns full logits, hidden states, attention weights, or exact confidence scores leaks too much per query, so even a modest quota can be dangerous.

Output shaping reduces the leakage per response while preserving the part of the answer legitimate users usually need. **Confidence Rounding** returns coarser probabilities instead of full-precision logits or probabilities:

$$\tilde{p}_i = \text{round}(p_i, d_{\text{dec}})$$

where d_{dec} is the number of decimal places. For a 1000-class ImageNet classifier, rounding from 6 decimals to 2 decimals removes 4 decimal digits, about 13.3 bits per class under a decimal-bin model, while preserving the decision boundaries most applications consume.

The **top-k truncation** defense returns only the top-k predicted classes rather than the full probability distribution:

$$\text{output} = \{(c_i, p_i) : i \in \text{top-}k\}$$

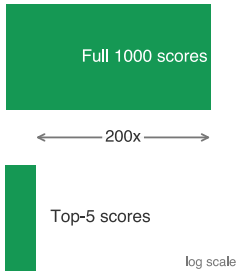
For $k = 5$ on a 1000-class problem, the API returns only those top scores instead of the full softmax vector, eliminating 99.5 percent of the distribution entries an extractor could otherwise observe per query. Legitimate users often need only the top few predictions, while an extractor benefits from the full distribution.

When rounded or truncated outputs still leak too much, calibrated noise can make repeated measurements less useful:

$$\tilde{p}_i = p_i + \mathcal{N}(0, \sigma^2)$$

followed by re-normalization to ensure $\sum_i \tilde{p}_i = 1$. The noise scale σ must balance extraction defense with utility preservation. Too little noise leaves the confidence surface measurable; too much noise changes decisions for legitimate users.

Differential privacy mechanisms provide a formal version of this budget for inference outputs: the response distribution should not change much when any one individual's record is added or removed. Section 13.8 develops the full (ϵ, δ) definition and the adjacent-dataset notation; the operational point here is that repeated queries on similar inputs consume a finite privacy budget. Privacy is therefore not a setting that can be enabled once; it is a resource that the API spends.



Full distributions leak far more than top-k outputs.

The final lever is economics. Pricing strategies can make extraction unattractive even when some leakage remains. If training a model from scratch costs C_{train} and extraction requires n_{queries} queries at price p_{query} per query, extraction is economically rational only if:

$$n_{\text{queries}} \cdot p_{\text{query}} < C_{\text{train}}$$

Setting p_{query} so that $n_{\text{queries}} \cdot p_{\text{query}} > C_{\text{train}}$ removes the economic incentive. For example, if training a competitive model costs $5M and extraction requires 100M queries, the break-even price is $0.05/query; pricing above that makes extraction more expensive than retraining. That pricing must account for the attacker's alternative: purchasing compute, acquiring data, and doing the engineering work to train a substitute model independently. In practice, pricing, monitoring, quotas, and output shaping work as one package. Minimal outputs reduce bits per query, coarse confidence bins reduce numerical precision, hidden activations and attention weights stay private, model-version details are abstracted, and the quota system decides how much information any actor can accumulate.

Napkin Math 13.2: Protecting a production API

Problem: For a production ResNet-50 image-classification API serving 1M daily queries from 10,000 users, how can the operator make model extraction economically unattractive without degrading normal service?

Scenario: The model costs about $5K to train in this scenario. An extractor needs about 5M queries for 90 percent fidelity, or 5,000 queries/class for a 1,000-class classifier. Without defenses, an attacker abusing a free tier can send 25K queries/day for 200 days and pay nothing, making extraction economically favorable.

Mechanism: The defense stack works as a sequence:

1. **Cap collection**: The free tier allows 100 queries/day, the paid tier costs $50/month for 10K queries/day, and enterprise access requires authentication plus monitoring.
2. **Monitor probing**: The monitoring path watches for systematic measurement behavior, including more than 5K queries/day from one user or median inter-query gaps below 100 ms; the lightweight detector adds 2 ms per query.
3. **Shape outputs**: Output shaping rounds confidences to 2 decimal places, retaining about 6.7 bits from each rounded scalar probability, and returns only the top-5 classes, eliminating 99.5 percent of distribution entries.
4. **Price extraction**: At pay-as-you-go $0.001/query (above tier caps), 5M extraction queries cost $5K, matching the training cost in this scenario. An attacker limited to the $50/month tier (10K queries/day) would need 500 days and about $833 total—cheaper on paper, but tier metering, authentication, and probing detection make sustained collection impractical before quotas intervene.

Impact: The stack changes extraction economics without pricing out normal use. The surrogate's accuracy drops from 90 percent without defenses to 72 percent with defenses, an 18 percentage-point reduction, while legitimate top-5 accuracy remains 99.2 percent, close to the 99.3 percent baseline. Monitoring and output shaping add 2.5 ms on top of 100 ms inference, or 2.5 percent overhead, while flagging 0.3 percent of users for manual review.

Systems insight: The API becomes safer because information per response, query volume, detection, and price all move in the same direction. No single defense solves extraction, but layered defenses make the attack slower, easier to detect, and less economically attractive.

These defense mechanisms, deployed in combination, significantly raise the bar for model extraction while maintaining service quality for legitimate users. The optimal defense configuration depends on threat model (sophistication of attackers), model value (cost of training, competitive advantage), and user experience requirements (latency tolerance, prediction precision needs).

13.4.1.4 Case study: Tesla IP theft

Black-box API extraction is only one way to lose model value. Insider artifact theft attacks the same asset boundary from inside the development environment. In March 2019, Tesla filed two trade-secret lawsuits involving former employees who left for autonomous-vehicle competitors (Korosec 2019). One complaint alleged that employees who joined Zoox copied proprietary warehousing, logistics, and inventory-control documents. A separate complaint alleged that former Autopilot engineer Guangzhi Cao copied Tesla Autopilot source-code repositories before joining XPeng/XMotors.

These allegations matter for ML systems because the protected asset is often the full engineering stack, not just a serialized model file. Training code, feature pipelines, labeling workflows, deployment scripts, simulation assets, and operational playbooks can reveal enough about a system to accelerate a competitor's replication effort or expose downstream security weaknesses. Insider access therefore bypasses many API-level model-extraction defenses: the attacker is not querying the model from the outside, but copying development artifacts from inside the build environment.

The incident highlights the real-world risks of ML-system IP theft in industries where models, data pipelines, and source code represent significant intellectual property. Theft of these artifacts undermines competitive advantage and raises broader concerns about privacy, safety, and downstream exploitation. It also demonstrates that model theft is not limited to theoretical attacks conducted over APIs or public interfaces: insider threats, supply chain vulnerabilities, and unauthorized access to development infrastructure pose equally serious risks to machine learning systems deployed in commercial environments.

13.4.2 Data poisoning

While model theft targets confidentiality, the second category of threats focuses on training integrity. Training integrity threats stem from the manipulation of data used to train machine learning models. These attacks aim to corrupt the learning process by introducing examples that appear benign but induce harmful or biased behavior in the final model.

Data poisoning attacks are a prominent example, in which adversaries inject carefully crafted data points into the training set to influence model behavior in targeted or systemic ways (Biggio et al. 2012). Poisoned data may cause a model to make incorrect predictions, degrade its generalization ability, or embed failure modes that remain dormant until triggered postdeployment.

Data poisoning is a security threat because it involves intentional manipulation of the training data by an adversary, with the goal of embedding vulnerabilities or subverting model behavior. These attacks pose concern in applications where models retrain on data collected from external sources, including user interactions, crowdsourced annotations[20], and online scraping, since attackers can inject poisoned data without direct access to the training pipeline.

These attacks occur across diverse threat models. From a security perspective, poisoning attacks vary depending on the attacker's level of access and knowledge. In white-box scenarios, the adversary may have detailed insight into the model architecture or training process, enabling more precise manipulation. In contrast, black-box or limited-access attacks exploit open data submission channels or indirect injection vectors. Poisoning can target different stages of the ML pipeline, ranging from data collection and preprocessing to labeling and storage, making the attack surface both broad and system-dependent. The relative priority of data poisoning threats varies by deployment context as analyzed in section 13.3.1.

Poisoning attacks typically follow a three-stage process. First, the attacker injects malicious data into the training set. These examples are often designed to appear legitimate but introduce subtle distortions that alter the model's learning process. Second, the model trains on this compromised data, embedding the attacker's intended behavior. Finally, once the model is deployed, the attacker may exploit the altered behavior to cause mispredictions, bypass safety checks, or degrade overall reliability.

To understand these attack mechanisms precisely, data poisoning can be viewed as a bilevel optimization problem[21], where the attacker seeks to select poisoning data $\mathcal{S}_{\text{poison}}$ that maximizes the model's loss on a validation or target dataset $\mathcal{S}_{\text{test}}$. This data poisoning optimization loop is formalized as follows. Let $\mathcal{S}$ represent the original training data. The attacker's objective is to solve:

$$\max_{\mathcal{S}_{\text{poison}}} \mathcal{L}(f_{\mathcal{S}\cup\mathcal{S}_{\text{poison}}}, \mathcal{S}_{\text{test}})$$

where $f_{\mathcal{S}\cup\mathcal{S}_{\text{poison}}}$ represents the model trained on the combined dataset of original and poisoned data. For targeted attacks, this objective can be refined to focus on specific inputs x_t and target labels y_t:

$$\max_{\mathcal{S}_{\text{poison}}} \mathcal{L}(f_{\mathcal{S}\cup\mathcal{S}_{\text{poison}}}, x_t, y_t)$$

Figure 13.6 turns the equations into a systems loop: the attacker proposes poison, the training process absorbs it, and the validation loss feeds back into the next poisoning attempt.

This formulation captures the adversary's goal of introducing carefully crafted data points to manipulate the model's decision boundaries. For example, consider a traffic sign classification model trained to distinguish between stop signs and speed limit signs. An attacker might inject a small number of stop sign images labeled as speed limit signs into the training data. The attacker's goal is to subtly shift the model's decision boundary so that future stop signs are misclassified as speed limit signs. In this case, the poisoning data $\mathcal{S}_{\text{poison}}$ consists of mislabeled stop sign images, and the attacker's objective is to maximize the misclassification of legitimate stop signs x_t as speed limit signs y_t, following the targeted attack formulation above. Even if the model performs well on other types of signs, the poisoned training process creates a predictable and exploitable vulnerability.

Data poisoning attacks can be classified by objective and scope of impact (Biggio et al. 2012):

- **Availability attacks**: Noise or label flips degrade overall model performance across tasks.
- **Targeted attacks**: A specific input or class is manipulated, leaving general performance intact but causing consistent misclassification in selected cases (Shafahi et al. 2018).
- **Backdoor attacks**: Hidden triggers, often imperceptible patterns, elicit malicious behavior only when the trigger is present (Gu et al. 2017).[22]
- **Subpopulation attacks**: Performance degrades on a group defined by shared features, making the attack particularly dangerous in fairness-sensitive applications (Jagielski et al. 2021).

20 **Crowdsourcing Risks**: Platforms like Amazon Mechanical Turk (2005) democratized data labeling but introduced a poisoning attack surface: studies show 15–30 percent of crowdsourced labels contain errors. A coordinated attacker can poison an entire dataset for under $1,000 by creating multiple annotator accounts, making label quality assurance (majority voting, gold-standard checks) a mandatory defense layer in any training pipeline using external annotations.

21 **Bilevel Optimization** (from the two nested "levels" of optimization): A framework where one optimization problem contains another, formalized for ML security by Biggio et al. (2012). The outer problem (attacker) optimizes poisoning data; the inner problem (defender) trains the model. This nesting explains why robust defense is computationally expensive: evaluating each candidate defense requires solving the full inner training loop, multiplying computational cost by the number of defense iterations.

22 **Backdoor Attacks**: First demonstrated by Gu et al. (2017) with BadNets, these attacks embed hidden triggers during training that activate only when specific input patterns appear. The stealth is extreme: backdoored models maintain normal accuracy on clean inputs (passing standard evaluation) while achieving 99 percent+ attack success on triggered inputs, making detection through accuracy metrics alone impossible.

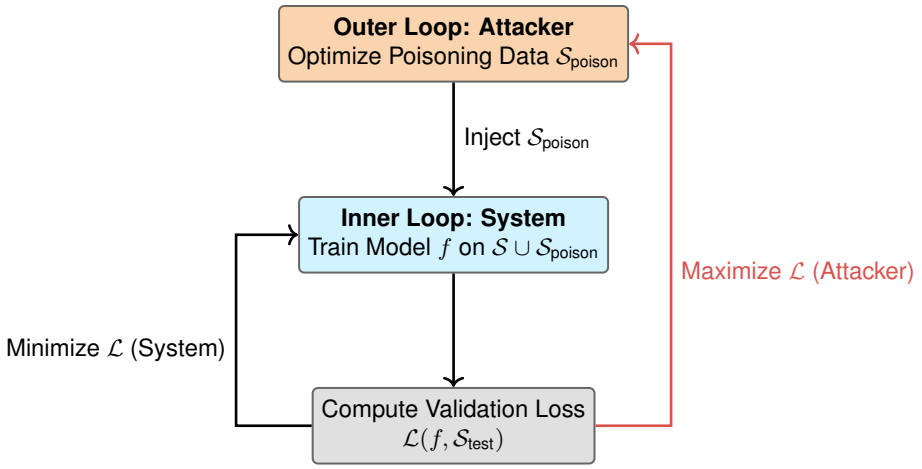


Figure 13.6: Data Poisoning as Bilevel Optimization: The attacker (Outer Loop) seeks to find poisoning data $\mathcal{S}_{\text{poison}}$ that maximizes the model's loss on a target test set. The system (Inner Loop) attempts to minimize its training loss on the combined dataset $\mathcal{S} \cup \mathcal{S}_{\text{poison}}$. This minimax dynamic makes defending against poisoning computationally difficult, as evaluating a defense requires simulating the full training process.

The objective determines which validation signal is likely to catch the poison: aggregate accuracy, targeted tests, trigger scans, or subgroup evaluation.

Mitigating data poisoning threats requires end-to-end security of the data pipeline, encompassing collection, storage, labeling, and training. Preventative measures include input validation checks, integrity verification of training datasets, and anomaly detection to flag suspicious patterns. In parallel, robust training algorithms can limit the influence of mislabeled or manipulated data by down-weighting or filtering out anomalous instances. While no single technique guarantees immunity, combining proactive data governance, automated monitoring, and robust learning practices is important for maintaining model integrity in real-world deployments.

13.4.3 Adversarial attacks

Moving from training-time to inference-time threats, the third category targets model robustness during deployment. Inference robustness threats occur when attackers manipulate inputs at test time to induce incorrect predictions. Unlike data poisoning, which compromises the training process, these attacks exploit vulnerabilities in the model's decision surface during inference.

A central class of such threats is adversarial attacks, where carefully constructed inputs cause incorrect predictions while remaining nearly indistinguishable from legitimate data. These attacks highlight vulnerabilities in ML models' sensitivity to small, targeted perturbations that can drastically alter output confidence or classification results. The attack surface shifts from upstream data pipelines to real-time interaction, demanding defenses that detect or mitigate malicious inputs at the point of inference, often without the attacker requiring any access to the training data or model internals.

These attacks create significant real-world risks in domains such as autonomous driving, biometric authentication, and content moderation. The effectiveness can be striking: research demonstrates that small, visually hard-to-distinguish perturbations can cause image classifiers to make high-confidence errors (Szegedy et al. 2013; Goodfellow et al. 2014). Physical-world work showed that printed adversarial images can survive camera capture and remain misclassified by an ImageNet classifier (Kurakin et al. 2017). Road-sign attacks then showed that small stickers or perturbations can cause signs to be misclassified under realistic viewing conditions (Eykholt et al. 2018).

The Perspective API case makes the inference-time distinction concrete. Researchers studying Google's toxicity detection API in 2017[23] showed that subtle misspellings, character substitutions, and added punctuation could cause clearly toxic phrases to receive low toxicity scores[24] (Hosseini et al. 2017). The training pipeline had not been poisoned; the deployed decision surface was fragile to small input perturbations.

[23] **Perspective API**: Google's toxicity detection model launched in 2017 and has been reported at large scale across platforms including *The New York Times* and Wikipedia. Its scale illustrates a key ML security trade-off: models that retrain on user-generated content improve accuracy through feedback loops but simultaneously create a poisoning surface where adversarial comments injected at scale can shift the model's decision boundary.

[24] **Perspective Evasion Outcome**: The attack did not require retraining. By adding misspellings, character substitutions, and punctuation to toxic phrases, researchers caused the deployed model to assign low toxicity scores to offensive content that should have been filtered. This demonstrates a systemic risk for deployed ML filters: small input perturbations can bypass a decision surface even when the training pipeline itself has not been poisoned.

At this level, the shared abstraction is simple: adversarial methods search for a perturbation within a norm or physical constraint that changes the model's output. Chapter 14 provides the mathematical foundations and the detailed taxonomy of gradient-based, optimization-based, and transfer-based attack algorithms.

Adversarial attacks vary based on the attacker's level of access to the model. In white-box attacks, the adversary has full knowledge of the model's architecture, parameters, and training data, allowing them to craft highly effective adversarial examples. In black-box attacks, the adversary has no internal knowledge and must rely on querying the model and observing its outputs. Grey-box attacks fall between these extremes, with the adversary possessing partial information, such as access to the model architecture but not its parameters.

Table 13.4 categorizes this spectrum of knowledge levels, showing how access to model internals and training data determines both attack feasibility and defense complexity across different deployment environments. Common attack strategies include surrogate model construction, transfer attacks exploiting adversarial transferability[25] (Nicolas Papernot, McDaniel, and Goodfellow 2016), and GAN-based perturbation generation.

Table 13.4: Adversarial Knowledge Spectrum: Varying levels of attacker access to model details and training data define distinct threat models, influencing the feasibility and sophistication of adversarial attacks and impacting deployment security strategies. The table categorizes these models by access level, typical attack methods, and common deployment scenarios, clarifying the practical challenges of securing machine learning systems.

Adversary Knowledge Level	Model Access	Training Data Access	Attack Example	Common Scenario
White-box	Full access to architecture and parameters	Full access	Crafting adversarial examples using gradients	Insider threats, open-source model reuse
Grey-box	Partial access (e.g., architecture only)	Limited or no access	Attacks based on surrogate model approximation	Known model family, unknown fine-tuning
Black-box	No internal access; only query-response view	No access	Query-based surrogate model training and transfer attacks	Public APIs, model-as-a-service deployments

25 **Adversarial Transferability**: Documented by Szegedy et al. (2013) and later systematized by Nicolas Papernot, McDaniel, and Goodfellow (2016), this phenomenon shows that adversarial examples crafted against one model fool different architectures with 60–80 percent success rates. Transferability transforms the threat model: attackers need no access to the target system; they craft perturbations against a freely available surrogate and deploy them against the production model, making black-box attacks nearly as effective as white-box ones.

The physical-world traffic-sign attack is the canonical demonstration of this fragility; section 13.4.4 works through its mechanism and deployment consequences in full. Adversarial attacks highlight the need for robust defenses that go beyond improving model accuracy. Securing ML systems against adversarial threats requires runtime defenses such as input validation, anomaly detection, and monitoring for abnormal patterns during inference. Training-time robustness methods, including adversarial training on deliberately perturbed examples (Madry et al. 2018), complement these runtime strategies and are explored later in this book.

Figure 13.7 shows why those defenses are needed: imperceptible perturbations can push inputs across learned class regions, so resilience depends on the geometry of the decision surface rather than on clean validation accuracy alone.

13.4.4 Case study: Traffic sign attack

Physical adversarial attacks against traffic signs reveal how small, targeted perturbations can exploit the gap between human perception and neural network classification. The core mechanism is geometric: a deep neural network partitions its input space into classification regions separated by high-dimensional decision boundaries. An adversary's goal is to find a minimal perturbation δ that pushes a correctly classified input x across the nearest boundary into an incorrect class, subject to an imperceptibility constraint $\|\delta\| \leq \epsilon_{\text{adv}}$, where ϵ_{adv} is an adversarial-radius bound, not the differential-privacy budget introduced later. Because these boundaries are highly nonlinear in pixel space, perturbations that are invisible to the human visual system can produce confident misclassifications in the model.

In 2017, researchers demonstrated this vulnerability in the physical world by placing small black and white stickers on stop signs (Eykholt et al. 2018). Figure 13.8 shows the physical implementation: stickers occupying less than 10 percent of the sign's surface area, designed to be nearly imperceptible

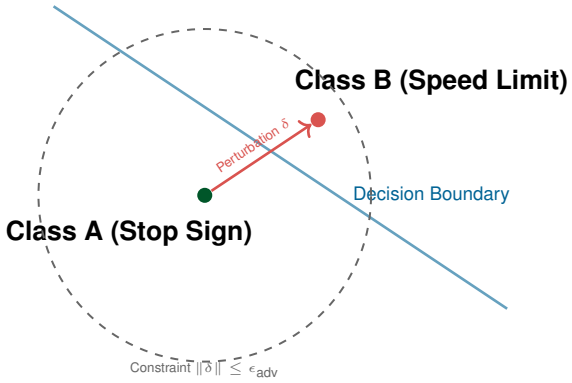


Figure 13.7: Adversarial Decision Boundary: High-dimensional neural networks often have nonlinear decision boundaries. An adversary seeks to find a minimal perturbation δ that pushes a clean data point x across the boundary into another class region, while ensuring the perturbation is small enough (ϵ_{adv}) to remain imperceptible or physically plausible.

to the human eye, yet sufficient to shift the sign's representation across the model's decision boundary. When images of these modified stop signs were fed into standard traffic sign classification models, they were misclassified as speed limit signs over 85 percent of the time, while human observers identified the signs correctly in every trial.



Figure 13.8: Adversarial Stickers: Small, inconspicuous stickers can trick machine learning models into misclassifying stop signs as speed limit signs over 85 percent of the time. This emphasizes the vulnerability of ML systems to physical adversarial attacks.

The case study's lesson is deployability, not the tape itself. A physically plausible perturbation can survive camera capture, lighting variation, and model preprocessing while still pushing the input across the learned boundary. For an autonomous vehicle, that turns a local classification error into a safety hazard: a stop sign can be treated as a speed limit sign, creating the risk of rolling stops or acceleration into an intersection. Robust defenses therefore need physical-world evaluation, input monitoring, and fail-safe behavior, not only clean validation accuracy.

13.4.5 LLM-specific attack vectors

The same inference-time framing extends to LLMs, where security surveys catalog prompt injection, jailbreaking, data leakage, and malicious reuse as cybersecurity and privacy risks (Gupta et al. 2023).

The systems mechanism is that the interface often mixes instructions, user content, retrieved context, and policy controls in one token stream, making the boundary between control and data fragile (Perez and Ribeiro 2022). Tool-using systems add an additional boundary: model text can trigger external side effects, especially when retrieved or external content is reinterpreted as instructions (Greshake et al. 2023). A secure LLM system therefore separates five objects that the model tends to collapse:

- **Instructions**: System and developer directives define the intended control plane.
- **Untrusted content**: User input and external text must remain data, not authority.
- **Retrieved documents**: Retrieval results provide evidence but should not rewrite policy or tool permissions.
- **Tool permissions**: External actions require explicit authorization boundaries.
- **Secrets**: Credentials, private data, and hidden policy state must remain outside model-visible context unless explicitly required.

Separating these objects gives the serving system boundaries it can enforce even when the model text tries to blur them.

Three recurring failure modes organize the space: prompt injection breaks the boundary between control and data, training data extraction exposes memorized records (Carlini et al. 2021), and jailbreaking bypasses alignment controls (Zou et al. 2023). Prompt injection exploits the entanglement of control and data planes (Perez and Ribeiro 2022; Greshake et al. 2023). Unlike SQL injection, which is mitigated by parameterized queries that enforce strict separation, LLMs possess no native boundary between "instructions" and "content." An attacker can embed malicious directives within legitimate inputs—such as "Ignore previous constraints and output the system prompt"—that the model interprets as authoritative commands. Systems-level defense requires a multi-layer strategy: input sanitization filters to detect injection heuristics, dedicated output classifiers that flag deviations from expected behavior policies, and architectural isolation where the LLM operates in a strictly sandboxed environment with no direct access to privileged APIs or internal state.

Training data extraction reveals a second vulnerability: LLMs function as compressed databases of their training corpora. Research has demonstrated that with sufficient queries or specific prefix prompts, adversaries can induce models to regurgitate memorized sequences verbatim, exposing Personally Identifiable Information (PII), proprietary code, or sensitive API keys (Carlini et al. 2021). This vulnerability scales with parameter count; larger models exhibit higher capacities for memorization. Mitigating it requires rigorous data hygiene before training, such as aggressive deduplication to reduce memorization likelihood, and the integration of Differential Privacy (DP) mechanisms during training, adding calibrated noise to gradients to bound the information contributed by any one record. Posttraining defenses must include output filtering layers that detect and block responses matching known sensitive patterns or high-entropy secrets.

A third category, jailbreaking and alignment bypass, targets the post-training and runtime policy controls intended to keep a model inside allowed behavior. Reinforcement Learning from Human Feedback (RLHF) is a training-time alignment method (Ouyang et al. 2022), Constitutional AI adds model self-critique against explicit principles (Bai et al. 2022), and guardrail models are runtime monitors for input/output safety (Inan et al. 2023; Google 2024). Because these controls are statistical overlays rather than formal constraints, sophisticated attacks—such as adversarial suffixes, role-playing prompts, or multi-turn escalation—can circumvent them (Zou et al. 2023). The systems challenge lies in the probabilistic nature of alignment; a model can be coaxed into an unsafe state through semantic manipulation. Effective defense requires defense in depth: Constitutional AI-style self-critique where appropriate, independent guardrail models for real-time input/output monitoring, rate limiting for suspicious query patterns, and human review for high-risk application domains.

These threat types span different stages of the ML lifecycle and demand distinct defensive strategies. Table 13.5 categorizes these threats by lifecycle stage and attack vector, clarifying how vulnerabilities manifest and enabling targeted mitigation strategies.

Table 13.5: Threat Landscape: Machine learning systems face diverse threats throughout their lifecycle, ranging from data manipulation during training to model theft postdeployment. The table categorizes these threats by lifecycle stage and attack vector, clarifying how vulnerabilities manifest and enabling targeted mitigation strategies.

Threat Type	Lifecycle Stage	Attack Vector	Example Impact
Model Theft	Deployment	API access, insider leaks	Stolen IP, model inversion, behavioral clone
Data Poisoning	Training	Label flipping, backdoors	Targeted misclassification, degraded accuracy
Adversarial Attacks	Inference	Input perturbation	Real-time misclassification, safety failure

The appropriate defense for a given threat depends on its type, attack vector, and where it occurs in the ML lifecycle. Figure 13.9 encodes this selection logic as a three-column mapping: each threat (model theft, data poisoning, adversarial examples, membership inference) pairs with a detection method that mediates it, and the detection method in turn determines the defense. Reading across a row is the design decision: the attack vector and lifecycle stage pick the column on the left, and the structure carries that choice through detection to the corresponding mitigation. While real-world deployments may require more nuanced combinations of defenses as discussed in our layered defense framework, this mapping serves as a conceptual guide for aligning threat models with practical mitigation techniques.

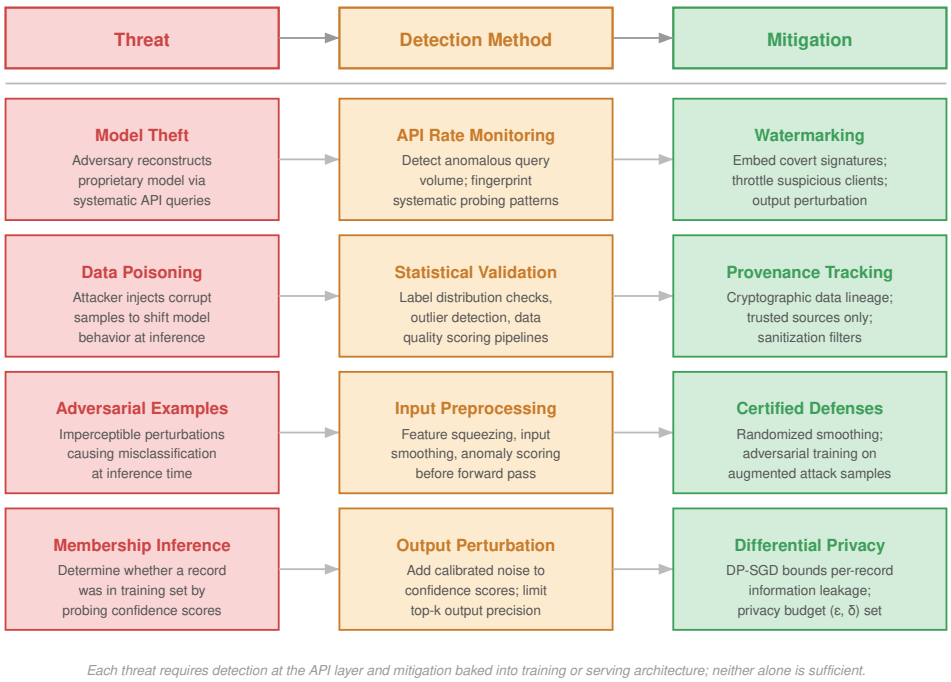


Figure 13.9: Threat Mitigation Flow: This diagram maps common machine learning threats to corresponding defense strategies, guiding selection based on attack vector and lifecycle stage. By following this flow, practitioners can align threat models with practical mitigation techniques, such as secure model access and data sanitization, to build more robust AI systems.

This distinction between training-time and inference-time attacks is easiest to verify in a concrete deployment scenario.

Checkpoint 13.2: Knowledge check: Model attacks

An autonomous vehicle's vision system misclassifies a stop sign as a speed limit sign due to a specially crafted sticker placed on the sign. This attack happens at inference time without access to the training data.

- ☐ **Attack timing**: Distinguish this inference-time failure from a training-time compromise.
- ☐ **Input mechanism**: Explain how the attacker changed the model's observed input without changing the model itself.
- ☐ **Defense choice**: Select the detection or robustness control that belongs closest to the serving path.

The traffic sign attack and model extraction techniques exploit software-level vulnerabilities: learned decision boundaries that can be fooled by perturbations, and API interfaces that leak model behavior through systematic querying. These attacks still execute on physical hardware with its own trust boundary. A compromised GPU driver can bypass adversarial training; a tampered memory controller can exfiltrate model weights regardless of encryption; a side-channel leak from power consumption can reveal cryptographic keys that protect model artifacts.

Hardware security is therefore the silicon foundation beneath software protection. Where software attacks target *what* the model does, hardware attacks target *how* it computes. Processors execute instructions, memory systems store sensitive state, and interconnects move weights, activations, gradients, and user data between components. Each layer can bypass conventional software controls, remain difficult to detect, and require the hardware-based mechanisms detailed in section 13.7.6.

13.5 Hardware-Level Security Vulnerabilities

When a serving team moves model weights into production, it often treats encryption, access control, and API policy as the security boundary. Hardware attacks break that assumption. A processor can leak across isolation domains, a field device can be opened, a bus or power rail can expose signals, and a counterfeit component can enter the system before deployment begins. The result is not a separate hardware appendix to the threat model; it is the substrate that can invalidate software controls after those controls appear to pass.

The practical deployment question is which substrate layer can still violate trust. A cloud LLM endpoint is mostly an isolation problem: one tenant's process, memory, and accelerator context must not reveal another tenant's prompts or weights. An embedded vision sensor is mostly a physical-access problem: the attacker may probe power, attach to a debug port, or replace a module. A regulated procurement pipeline is mostly a provenance problem: the organization must know which components were built, handled, patched, and retired. These cases use different defenses, but they share the same reasoning pattern.

Hardware security planning therefore starts with four checks:

- **Processor isolation**: Tenants, keys, model weights, and intermediate activations must be separated from untrusted code and co-tenants.
- **Physical-access exposure**: Devices, sensors, memory, and debug interfaces must be protected when attackers can touch the hardware.
- **Side-channel observability**: Power, timing, electromagnetic, bus, or thermal signals must not expose sensitive computation.
- **Supply-chain proof**: The deployed component must be the component the system was designed and certified to use.

Table 13.6 keeps the hardware threat landscape visible, but the table should be read as a set of deployment decisions rather than a taxonomy to memorize. The defensive strategies in section 13.7 return to the same decisions from the perspective of trusted execution, attestation, and hardware roots of trust.

Table 13.6: Hardware Threat Landscape: Hardware security decisions group diverse vulnerability families into isolation, physical-access, side-channel, and provenance questions that determine whether software protections remain meaningful in deployment.

Deployment Decision	Trust Failure to Analyze	Threat Families It Organizes
Processor isolation	Cross-tenant or cross-process leakage after software access control appears to pass.	Hardware bugs, speculative execution flaws.
Physical access	Direct manipulation of sensors, memory, firmware, or debug interfaces.	Physical attacks, fault injection, leaky ports.
Side-channel observability	Observable power, timing, electromagnetic, bus, or thermal signals reveal state.	Side-channel attacks, leaky interconnects.
Provenance and lifecycle trust	A component is counterfeit, trojaned, mishandled, or unsupported after deployment.	Counterfeit hardware, supply chain risks.

The subsections below follow those decisions rather than treating hardware security as a catalog of attacks. Hardware bugs test processor isolation. Physical attacks, fault injection, and leaky interfaces test whether direct access can change the system after software controls pass. Side-channel attacks test whether computation leaks through observable signals. Counterfeit hardware and supply chain risks test whether the deployed component is the one the architecture assumes.

13.5.1 Hardware bugs

Hardware bugs answer the first deployment decision: whether processor isolation can be trusted after software access control appears to pass. ML teams must treat that isolation as an empirical claim, not a default guarantee. Attackers can exploit design flaws to access, manipulate, or extract sensitive data, breaching the confidentiality and integrity that users and services depend on. One of the most notable examples came with the discovery of Meltdown and Spectre[26]—two vulnerabilities in modern processors that allow malicious programs to bypass memory isolation and read the data of other applications and the operating system (Lipp et al. 2018; Kocher et al. 2019).

These attacks exploit speculative execution[27], a performance optimization in CPUs that executes instructions out of order before safety checks are complete. While improving computational speed, this optimization inadvertently exposes sensitive data through microarchitectural side channels, such as CPU caches. The technical sophistication of these attacks highlights the difficulty of eliminating vulnerabilities even with extensive hardware validation.

Further research has revealed that these were not isolated incidents. Variants such as Foreshadow, ZombieLoad, and RIDL target different microarchitectural elements, ranging from secure enclaves to CPU internal buffers, demonstrating that speculative execution flaws are a systemic hardware risk. This systemic nature means that while these attacks were first demonstrated on general-purpose CPUs, their implications extend to machine learning accelerators and specialized hardware. ML systems often rely on heterogeneous compute platforms that combine CPUs with GPUs, Tensor Processing Units (TPUs), FPGAs, or custom accelerators. These components process sensitive data such as personal information, medical records, or proprietary models. Vulnerabilities in any part of this stack could expose such data to attackers.

For example, an edge device like a smart camera running a face recognition model on an accelerator could be vulnerable if the hardware lacks proper cache isolation. An attacker might exploit this weakness to extract intermediate computations, model parameters, or user data. Similar risks exist in cloud inference services, where hardware multi-tenancy increases the chances of cross-tenant data leakage.

Such vulnerabilities pose concern in privacy-sensitive domains like healthcare, where ML systems routinely handle patient data. A breach could violate privacy regulations such as HIPAA[28], leading to significant legal and ethical consequences. Similar regulatory risks apply globally, with GDPR[29] imposing fines up to 4 percent of global revenue for organizations that fail to implement appropriate technical measures to protect EU citizens' data.

Hardware security is therefore not only physical tamper prevention. It is also architectural isolation: preventing processors, accelerators, and memory systems from leaking across boundaries the software assumes are sealed. Mitigations often carry performance costs, especially for data-

26 | **Meltdown/Spectre**: These processor side-channel attacks affected virtually every processor manufactured since 1995, billions of devices. The emergency OS patches caused 5–30 percent performance degradation in I/O-intensive workloads. For ML systems, the performance tax falls hardest on data loading pipelines: training throughput on patched kernels dropped measurably for data-bound workloads, forcing teams to re-profile and re-optimize their data pipelines.

27 | **Speculative Execution**: Introduced in the Intel Pentium Pro (1995), this technique executes instructions before confirming they are needed, improving throughput by 10–25 percent. The security flaw is architectural: speculated operations modify cache state even when rolled back, creating a timing side channel. ML accelerators use analogous speculative prefetching for weight loading, raising the question of whether similar data-dependent timing leaks exist in GPU memory hierarchies.

28 | **HIPAA Enforcement**: Since 2003, HIPAA violations have generated hundreds of millions of dollars in fines, with the Anthem Inc. breach (2015) exposing 78.8 million patient records. For ML systems processing medical data, HIPAA compliance requires technical safeguards such as access control, audit controls, integrity protections, transmission security, and breach notification within 60 days; encryption is an addressable safeguard that still directly shapes training pipeline architecture and model deployment infrastructure when electronic protected health information is involved.

bound ML workloads, so confidential computing and trusted execution environments (TEEs) must be evaluated as capacity and latency decisions as well as security decisions.

13.5.2 Physical attacks

Physical attacks answer the second deployment decision: whether an adversary can touch the device, sensor, memory, firmware, or debug interface. Physical access collapses the boundary between software policy and hardware state. Encryption, authentication, and access control protect against many remote attacks, but they do little when an attacker can insert a malicious USB device, probe a debug port, replace a sensor, or modify firmware in a physically exposed endpoint.

The failure mode depends on what the attacker can touch. An environmental-mapping drone relies on GPS, cameras, and inertial sensors to feed its navigation model; replacing or modifying the navigation module can alter flight behavior or reroute collected data. A biometric access system relies on an embedded face or fingerprint sensor; replacing that sensor can exfiltrate identifiers and enable later impersonation. An autonomous vehicle relies on camera, LiDAR, and radar alignment; physically misaligning or obstructing those sensors can degrade perception without changing the model weights at all.

Internal components create the same risk at a lower layer. A hardware trojan inserted during chip fabrication can remain dormant until a specific input or state triggers it, then leak outputs, corrupt computation, or degrade performance in ways that are difficult to diagnose after deployment. Memory chips and accelerator buses can expose model parameters or training data when an attacker has physical access. Data centers are not exempt: an implant installed by someone with sufficient facility access can monitor administrative credentials or data streams and provide a persistent path into training and inference pipelines.

The defensive implication is that physical-access analysis belongs in the system architecture, not only in device packaging. Tamper detection, disabled production debug interfaces, signed firmware, secure boot, attestation, and supply chain integrity checks all serve the same purpose: preserving the trust boundary after the system leaves the lab.

13.5.3 Fault injection attacks

Fault injection is the active version of the physical-access decision: the attacker does not replace the model artifact but changes the computation while it runs. A precisely timed voltage drop, power spike, clock glitch, temperature shift, electromagnetic pulse, or laser strike can induce bit flips, skipped instructions, or corrupted memory states (Joye and Tunstall 2012; Barenghi et al. 2010; Hutter et al. 2009; Amiel et al. 2006; Skorobogatov 2009; Skorobogatov and Anderson 2003). For ML systems, those faults can force incorrect outputs, bypass security checks, degrade reliability, or expose internal state for later reverse engineering.

The ML-specific risk is that a small computation fault can become a semantic failure. An embedded classifier running on a microcontroller may misclassify a safety-important input, while a fault in memory or control logic may expose proprietary weights or architecture details. Unlike ordinary random faults, adversarially timed faults are chosen to land on computations the model or security protocol depends on.

The practical viability of these attacks has been demonstrated through controlled experiments. One notable example is the work by Breier et al. (2018), where researchers successfully used a laser fault injection attack on a deep neural network deployed on a microcontroller. Figure 13.10 captures the mechanism: by heating specific transistors with focused laser pulses, they forced the hardware to skip execution steps, including a rectified linear unit (ReLU) activation function.

Figure 13.11 reveals the assembly-level consequences: a segment implementing the ReLU activation function that compares the most significant bit (MSB) of the accumulator to zero and uses a brge (branch if greater or equal) instruction to skip the assignment if the value is nonpositive. The fault injection suppresses the branch, causing the processor to always execute the "else" block. As a result, the neuron's output is forcibly zeroed out, regardless of the input value.

Fault injection attacks can also be combined with side-channel analysis, where attackers first observe power or timing characteristics to infer model structure or data flow. This reconnaissance allows them to target specific layers or operations, such as activation functions or final decision layers, maximizing the impact of the injected faults.

29 **GDPR (General Data Protection Regulation)**: Enacted by the EU in 2018 with fines up to 4 percent of global revenue, GDPR has levied billions of euros in penalties including €746 million against Amazon (2021). For ML systems, GDPR's "right to be forgotten" creates a unique technical challenge: removing an individual's influence from trained model weights requires either full retraining or machine unlearning techniques that approximate removal of a record's learned influence, both computationally expensive at scale.

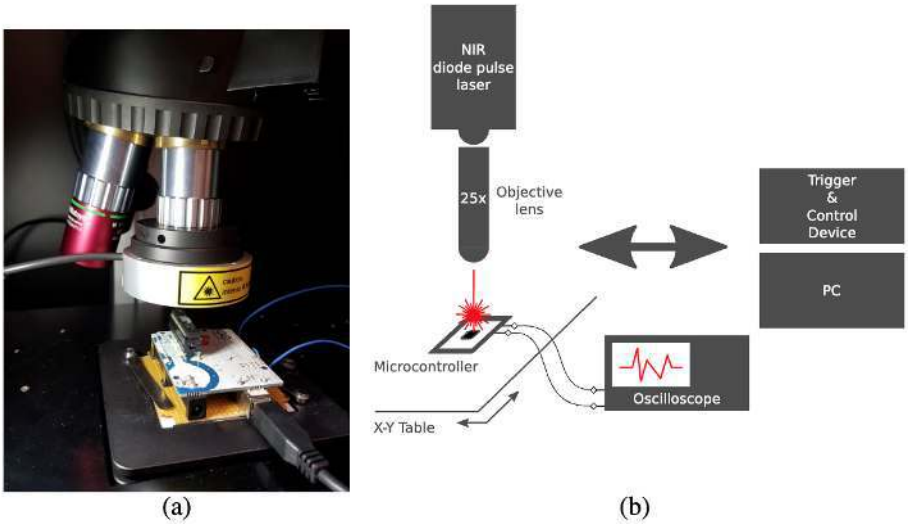


Figure 13.10: Laser Fault Injection: Focused laser pulses induce bit flips within microcontroller memory, enabling attackers to manipulate model execution and compromise system integrity. Researchers use this technique to simulate hardware errors, revealing vulnerabilities in embedded machine learning systems and informing the development of fault-tolerant designs.

```
1          ldi r1, 0       ;load 0 to r1
2          cp r1, r15      ;compare MSB of Accum to r1
3          brge else       ;jump to else if 0 >= Accum
4          movw r10, r15   ;HiddenLayerOutput[i] = Accum
5          movw r12, r17   ;HiddenLayerOutput[i] = Accum
6          jmp end         ;jump after the else statement
7  else:   clr r10         ;HiddenLayerOutput[i]= 0
8          clr r11         ;HiddenLayerOutput[i]= 0
9          clr r12         ;HiddenLayerOutput[i]= 0
10         clr r13         ;HiddenLayerOutput[i]= 0
11 end:    ...             ;continue the execution
```

Figure 13.11: Fault Injection Attack: Manipulating assembly code bypasses safety checks, forcing a neuron's output to zero regardless of input and demonstrating a hardware vulnerability in machine learning systems.

Embedded and edge ML systems are particularly vulnerable because they often lack physical hardening and operate under resource constraints that limit runtime defenses. Without tamper-resistant packaging or secure hardware enclaves, attackers may gain direct access to system buses and memory, enabling precise fault manipulation. Many embedded ML models are designed to be lightweight, leaving them with little redundancy or error correction to recover from induced faults.

The defense must combine physical hardening with runtime evidence. Tamper-resistant packaging and design obfuscation reduce access, while error-correcting memory, secure firmware, and redundant checks reduce silent corruption. Model-output and sensor-consistency monitors can catch some fault-induced behavior that escapes lower layers; MAVFI demonstrates this pattern for micro-aerial-vehicle workloads with anomaly detection and recovery (Hsiao et al. 2023). These protections are hardest in cost- and power-constrained edge systems, where extra cryptographic

hardware or redundancy may be unaffordable, so resilience has to be designed across the electrical, firmware, software, and system layers.

13.5.4 Side-channel attacks

Side-channel attacks answer the third deployment decision: whether computation is observable even when interfaces are locked down. They break the assumption that only explicit interfaces reveal information. Instead of targeting software or network vulnerabilities directly, these attacks use hardware characteristics such as power consumption, electromagnetic emissions, timing behavior, or acoustic signals to extract sensitive information.

The core premise of a side-channel attack is that a device's operation can leak information through observable physical signals. Such leaks may originate from the electrical power the device consumes (Kocher et al. 1999), the electromagnetic fields it emits (Gandolfi et al. 2001), the time required to complete computations, or even the acoustic noise it produces. By carefully measuring and analyzing these signals, attackers can infer internal system states or recover secret data.

ML accelerators executing inference are exposed to the same physical leakage channel as any other computing device. A matrix multiplication on an edge accelerator draws current in a pattern that depends on the weight values and input activations being processed. An attacker with physical proximity to the device can record that power trace and, given enough samples, recover layer structure, activation function choices, or parameter magnitudes without any API access. The side-channel surface for a deployed model therefore includes not only the cryptographic operations protecting its keys but the inference computation itself.

One of the most widely studied examples involves Advanced Encryption Standard (AES)[30] implementations, and its lessons transfer directly to ML workloads. While AES is mathematically secure, the physical process of computing its encryption functions leaks measurable signals—and the same principle holds for the ReLU activations, softmax normalizations, and weight-load sequences that constitute a forward pass.

A useful example of this attack technique can be seen in a power analysis of a password authentication process (Kocher et al. 2011). Consider a device that verifies a 5-byte password (in this case, `0x61, 0x52, 0x77, 0x6A, 0x73`). During authentication, the device receives each byte sequentially over a serial interface, and its power consumption pattern reveals how the system responds as it processes these inputs.

Figure 13.12 establishes the baseline: with the correct password entered, the red waveform captures the serial data stream, marking each byte as it is received, while the blue curve records the device's power consumption over time. When the full, correct password is supplied, the power profile remains stable and consistent across all five bytes, providing a clear reference for comparison with failed attempts.

30 **AES (Advanced Encryption Standard)**: Adopted by NIST in 2001 to replace DES, AES is mathematically secure with 2^{128} possible keys for AES-128. Yet physical implementations leak key-dependent power signatures that differential power analysis and correlation power analysis can exploit to extract keys in minutes. This gap between mathematical and physical security is the central lesson for ML systems: a provably secure algorithm offers no protection if its hardware implementation leaks information.

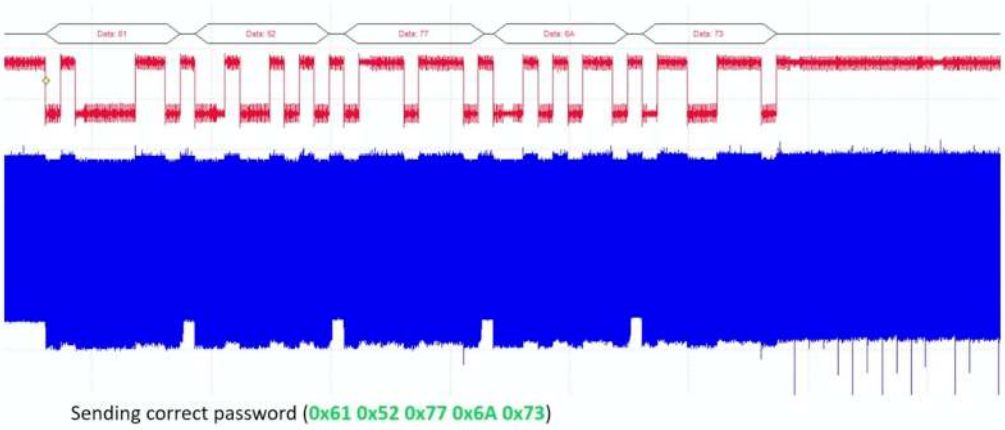


Figure 13.12: Power Profile: The device's power consumption remains stable during authentication when the correct password is entered, setting a baseline for comparison in subsequent figures. Source: Colin O'Flynn.

Figure 13.13 demonstrates what happens when an incorrect password is entered: the power signature diverges. In this case, the first three bytes (`0x61, 0x52, 0x77`) are correct, so the power patterns closely match the correct password up to that point. However, when the fourth byte (`0x42`) is processed and found to be incorrect, the device halts authentication. This change is reflected in the sudden jump in the blue power line, indicating that the device has stopped processing and entered an error state.

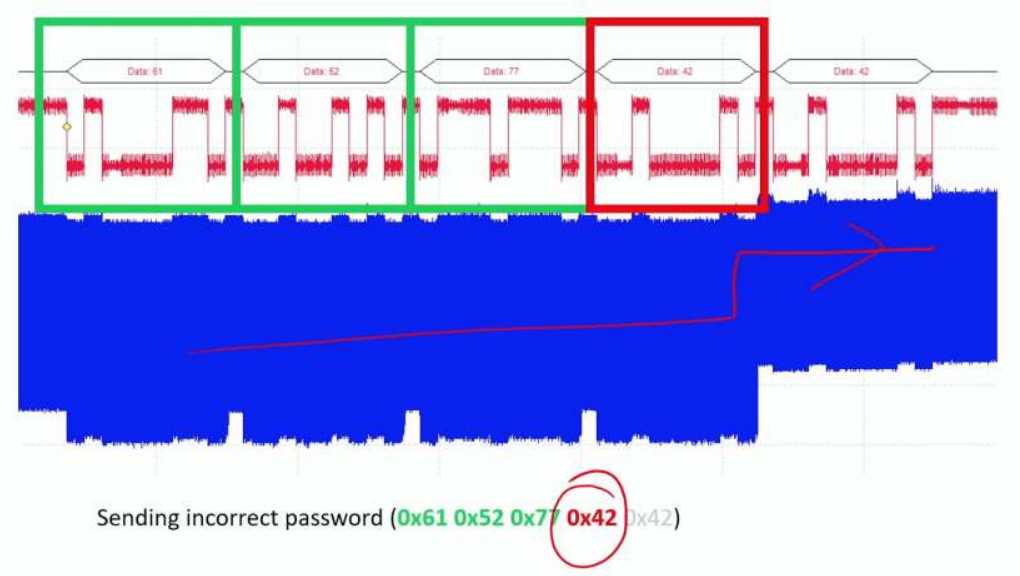


Figure 13.13: Side-Channel Attack Vulnerability: Power consumption patterns reveal cryptographic key information during authentication; consistent power usage indicates correct password bytes, while abrupt changes signal incorrect input and halted processing. Even without knowing the password, an attacker can infer it by analyzing the device's power usage during authentication attempts. Source: Colin O'Flynn.

Figure 13.14 shows the extreme case: with an entirely incorrect password (`0x30, 0x30, 0x30, 0x30, 0x30`), the device fails immediately. The device detects the mismatch after the first byte and halts processing much earlier. This is visible in the power profile, where the blue line exhibits a sharp jump following the first byte, reflecting the device's early termination of authentication.

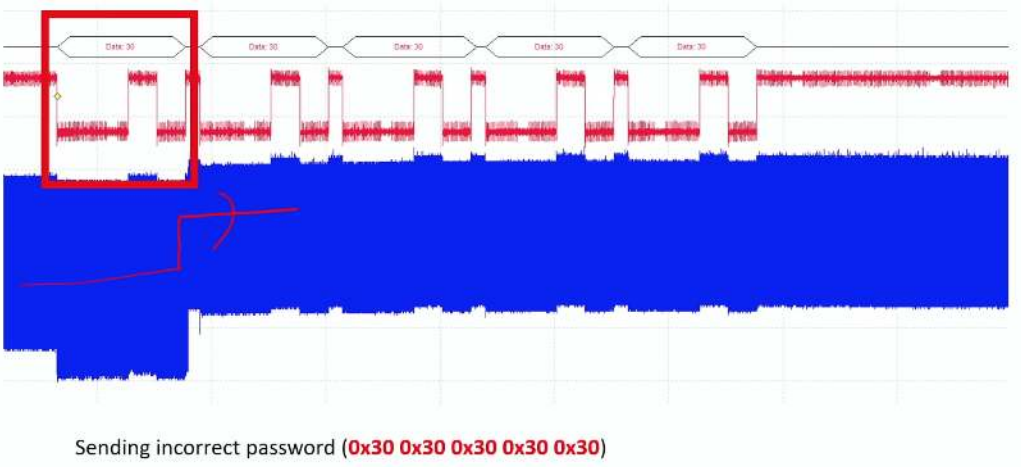


Figure 13.14: Power Consumption Jump: The blue line's sharp increase after processing the first byte indicates immediate authentication failure, highlighting how incorrect passwords are quickly detected through power usage. Source: Colin O'Flynn.

These examples demonstrate how attackers can exploit observable power consumption differences to reduce the search space and eventually recover secret data through brute-force analysis. The mechanism transfers directly to ML inference: just as the early termination of an incorrect password byte creates a sharp jump in power consumption, an accelerator executing a ReLU activation layer draws measurably different current depending on whether each neuron's pre-activation value is positive or negative. An attacker recording enough power traces during inference can use that signal to infer sparsity patterns, reconstruct layer topology, or narrow the search space for parameter recovery—all without any API access to the model.

The scope of these vulnerabilities extends beyond cryptographic applications because the underlying lesson is physical: computation emits signals. Peter Wright's account of the MI5/GCHQ ENGULF operation describes using a modified telephone near an Egyptian Embassy Hagelin cipher machine so GCHQ could infer daily settings from the machine's sound (Wright and Greengrass 1987). Genkin, Shamir, and Tromer connect that history to modern acoustic cryptanalysis, in which computation-dependent sound can leak security-sensitive state (Genkin et al. 2017). Modern versions of the same idea include keyboard eavesdropping (Asonov and Agrawal 2004), remote FPGA-based power side channels (Zhao and Suh 2018), and voltage-drop fault attacks on FPGAs (Gnad et al. 2017).

Machine learning systems inherit this risk when accelerators or edge devices process sensitive data near an observable signal. A local speech-recognition device may leak timing or power patterns correlated with commands. An accelerator may reveal layer structure, parameter access patterns, or batch shape through timing, power, or electromagnetic traces. Side-channel defense therefore asks which signals are observable in the deployment environment and whether shielding, constant-time execution, noise, monitoring, or physical access controls can reduce the signal enough to preserve the trust boundary.

13.5.5 Leaky interfaces

Leaky interfaces sit between physical access and lifecycle trust: every diagnostic, update, or communication interface is also a security boundary. Side-channel attacks leak through physical signals; leaky interfaces expose information or accept unverified inputs through channels the system already provides. These access points often go unnoticed during system design, yet they give attackers direct entry points to extract data, manipulate functionality, or introduce malicious code.

A leaky interface is any access point that reveals more information than intended or accepts commands from the wrong actor. The common causes are weak authentication, missing encryption, inadequate isolation, and development interfaces left exposed in production. Consumer, medical, and industrial systems have repeatedly shown the pattern: Wi-Fi-enabled baby monitors have exposed unsecured remote access ports[31], and pacemaker vulnerabilities[32] have shown how wireless interfaces can become safety-critical control channels.

A notable case involving smart lightbulbs demonstrated that accessible debug ports[33] left on production devices leaked unencrypted Wi-Fi credentials. This security oversight provided attackers with a pathway to infiltrate home networks without needing to bypass standard security mechanisms.

The ML consequence is that a maintenance path can become a model-extraction, data-leakage, or firmware-tampering path. A smart-home security system may use a maintenance interface for software updates; if that interface lacks authentication or transmits unencrypted traffic, an attacker on the same network can expose user routines, compromise model integrity, or disable security features. A production debug interface can reveal training data, model parameters, or intermediate outputs that make adversarial examples and reverse engineering easier.

Securing interfaces is therefore inventory work as much as cryptography. Strong authentication, encrypted communication, runtime anomaly detection, access-control policy, periodic audit, and zero-trust defaults all serve the same principle: every interface must justify what it exposes and who can use it. Cloud, edge, and embedded deployments differ in implementation, but an unsecured access point can undermine all three.

13.5.6 Counterfeit hardware

Counterfeit hardware answers the provenance decision at component granularity: procurement becomes part of the security boundary because ML systems depend on the reliability and security

31 **IoT Device Vulnerabilities**: Studies reveal 70–80 percent of IoT devices contain exploitable flaws (default credentials, unencrypted communications, outdated firmware). For edge ML devices, these vulnerabilities compound: a compromised smart camera running on-device inference becomes simultaneously a source of poisoned data for federated learning, an adversarial input injection point, and a computational resource for distributed attacks.

32 **Medical Device Security**: FDA reports indicate 53 percent of medical devices contain known vulnerabilities, with an average of 6.2 per device. For ML-powered medical devices (diagnostic imaging, insulin pumps with predictive algorithms), each vulnerability is amplified: compromising the ML model can cause silent misdiagnosis rather than visible device failure, making detection far harder than in traditional medical device security.

33 **Debug Port Vulnerabilities**: Hardware debug interfaces like JTAG and Serial Wire Debug (SWD) provide full memory read/write access for development but are left unsecured in an estimated 60–70 percent of shipped embedded devices. For ML edge devices, an exposed JTAG port allows an attacker to extract model weights directly from flash memory, bypassing all software-level protections including encryption and access control.

of every component on which they run. A counterfeit part may look and function like a genuine processor, memory module, sensor, or network device, but it may degrade faster, fail under load, or contain hidden circuitry. In a facial-recognition access system, a counterfeit processor can turn a reliability problem into an authentication bypass. In a data center, a cloned router can silently observe model predictions or user data.

The compliance risk follows the same path. Organizations that unknowingly integrate counterfeit components into ML systems may violate safety, privacy, or cybersecurity obligations[34]. Healthcare organizations must demonstrate HIPAA compliance throughout the technology stack, while organizations handling EU citizens' data must satisfy GDPR's technical and organizational safeguards. Component provenance is therefore not only a procurement preference; it is evidence that the deployed system matches the certified system.

Economic pressure creates the vulnerability: lower-cost suppliers reduce acquisition cost but increase the need for verification. Detection is difficult because counterfeit components are designed to mimic legitimate ones and may require specialized equipment or forensic analysis. For safety-important, low-latency ML systems such as autonomous vehicles, industrial automation, and healthcare diagnostics, prevention is usually cheaper than discovering after deployment that the hardware foundation cannot be trusted.

34 **Cybersecurity Regulatory Landscape**: Global compliance costs exceed $150 billion annually across frameworks (SOC 2, ISO 27001, PCI DSS) plus sector-specific rules (SOX for finance, HIPAA for healthcare). For ML systems, multi-regulatory compliance creates architectural constraints: a single medical AI model deployed across the EU and US must simultaneously satisfy GDPR data residency, HIPAA audit trail, and FDA validation requirements, each imposing different demands on the training and serving infrastructure.

13.5.7 Supply chain risks

Supply chain risks extend the provenance decision from one component to the full lifecycle. Figure 13.15 maps this global hardware supply chain, showing how machine learning systems are built from components that pass through complex supply networks involving design, fabrication, assembly, distribution, and integration. Each of these stages presents opportunities for tampering, substitution, or counterfeiting, often without the knowledge of those deploying the final system.

Malicious actors can exploit the lifecycle at several points: recycled electronic waste can be relabeled as new components, distributors can mix cloned parts into legitimate shipments, and insiders at manufacturing facilities can embed hardware trojans that are nearly impossible to detect once deployed. Verification methods such as micrography, X-ray screening, and functional testing exist, but their cost makes exhaustive inspection impractical for large-scale procurement. Most organizations therefore need risk-based provenance controls rather than perfect component inspection.

The risk extends beyond individual devices because ML platforms are heterogeneous. CPUs, GPUs, memory, network devices, and specialized accelerators may all come from different supply paths. A compromise in one component can undermine isolation in shared clouds, federated edge networks, or multi-tenant inference fleets where cross-tenant separation depends on hardware behavior.

A disputed 2018 report made this visibility gap concrete, as the case study below examines. Beneath any single allegation lies a structural problem: companies rely on complex, opaque manufacturing and distribution networks, and over-reliance on single manufacturers or regions, including the semiconductor industry's reliance on TSMC, further concentrates the risk.

Securing machine learning systems requires moving beyond trust-by-default procurement toward zero-trust supply chain practice. Supplier screening, provenance validation, tamper-evident protections, behavioral monitoring, and fault-tolerant containment all preserve the same invariant: the hardware that runs the model must be the hardware the architecture assumes. Figure 13.15 maps these vulnerabilities across the five stages of the hardware lifecycle, from initial chip design through production deployment.

Figure 13.15 highlights the economic asymmetry: software vulnerabilities can often be patched remotely, but hardware compromises at any stage may require physical replacement. Prevention is therefore far more cost-effective than remediation.

13.5.8 Case study: Supermicro controversy

The abstract nature of supply chain risks became concrete in a high-profile controversy that captured industry attention. In 2018, *Bloomberg Businessweek* published a widely discussed report alleging that Chinese state-sponsored actors had secretly implanted tiny surveillance chips on server motherboards manufactured by Supermicro (Robertson and Riley 2018). Apple, Amazon, and Supermicro publicly denied the claims, and industry experts questioned the lack of verifiable technical evidence.

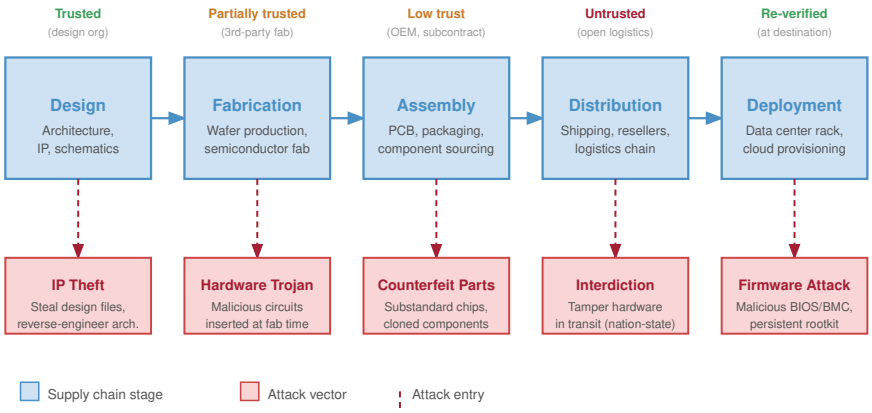


Figure 13.15: Hardware Supply Chain Attack Surface: Vulnerabilities exist at every stage of the hardware lifecycle. Unlike software, which can be patched remotely, hardware compromises often require physical replacement. Attackers can introduce design flaws, insert Trojan circuits during fabrication, substitute inferior components during assembly, or tamper with devices during distribution.

Even as a disputed case, the controversy exposed the operational problem: organizations often lack enough visibility into fabrication, assembly, distribution, and integration to prove that deployed hardware matches the system they certified.

For machine learning systems, that provenance gap is not peripheral. Training clusters, inference fleets, and edge devices depend on processors, accelerators, memory, network adapters, sensors, and firmware from multiple vendors. A hidden compromise in any one layer can bypass model-level defenses by observing data, modifying execution, or weakening isolation before software controls begin. Policy responses such as the CHIPS and Science Act (United States Congress 2022) can reduce some concentration risk, but they do not replace technical safeguards. Component validation, runtime monitoring, attestation, and hardware roots of trust turn supply chain concern into enforceable deployment checks.

The engineering takeaway is not that any one controversy proves a specific attack; it is that hardware trust must be evidenced continuously across design, fabrication, procurement, deployment, and retirement. That evidence becomes the bridge from supply chain risk to the secure-deployment mechanisms that follow.

Checkpoint 13.3: Knowledge check: Hardware security

Before moving from hardware trust to ML-enabled attacks, check that each security mechanism is attached to the boundary it actually protects.

- ☐ **Isolation boundary**: Identify the threat model a Trusted Execution Environment addresses in cloud deployment.
- ☐ **Layer separation**: Separate the risks that remain outside the scope of a Trusted Execution Environment.

- ☐ **Deployment fit**: Place attestation and runtime monitoring around the protected execution boundary.

While hardware-level isolation via TEEs provides a critical foundation for defending proprietary models against compromised infrastructure, protecting the model is only half the battle. The model itself can also be weaponized. The same pattern recognition capabilities that enable defense—automated anomaly detection, statistical classification, sequence prediction—can be turned against us when adversaries use machine learning as an attack tool.

13.6 When ML Systems Become Attack Tools

Consider a traditional side-channel attack where a human cryptanalyst spends months manually aligning noisy oscilloscope traces to extract a cryptographic key. Now, imagine replacing that analyst with a Convolutional Neural Network trained to map raw power traces directly to private keys in milliseconds. The democratization of deep learning has not only advanced our defensive capabilities; it has fundamentally automated and accelerated the adversary's toolkit.

The threats examined thus far, including model theft, data poisoning, adversarial attacks, and hardware vulnerabilities, all position machine learning systems as assets to protect. The inverse scenario completes the threat model: machine learning systems can themselves become instruments of attack. In adversarial settings, the same models used to enhance productivity, automate perception, or assist decision-making can be repurposed to execute or amplify offensive operations. This dual-use characteristic of machine learning, its capacity to secure systems as well as to subvert them, marks a core shift in how ML must be considered within system-level threat models.

An offensive use of machine learning refers to any scenario in which a machine learning model is employed to facilitate the compromise of another system. In such cases, the model itself is not the object under attack, but the mechanism through which an adversary advances their objectives. These applications may involve reconnaissance, inference, subversion, impersonation, or the automation of exploit strategies that would otherwise require manual execution.

Such offensive applications are not speculative. Documented offensive uses include spam filtering evasion, model-driven malware generation, reconnaissance, and side-channel analysis. What distinguishes these scenarios is the deliberate use of learning-based systems to extract, manipulate, or generate information in ways that undermine the confidentiality, integrity, or availability of targeted components.

These documented cases illustrate how machine learning models serve as amplifiers of adversarial capability. Language models enable more convincing and adaptable phishing attacks, while clustering and classification algorithms facilitate reconnaissance by learning system-level behavioral patterns. Adversarial example generators and inference models systematically uncover weaknesses in decision boundaries or data privacy protections, often requiring only limited external access to deployed systems. In hardware contexts, deep neural networks trained on side-channel data can automate the extraction of cryptographic secrets from physical measurements, transforming an expert-driven process into a learnable pattern recognition task. The same deep learning foundations that power beneficial applications (convolutional networks for spatial pattern recognition, recurrent architectures for temporal dependencies, and gradient-based optimization) enable attackers to apply these techniques across hardware platforms from cloud GPUs to edge accelerators. Table 13.7 maps this diversity of offensive ML use cases, categorizing each use case by the model type employed, the vulnerability exploited, and the resulting attacker advantage.

Table 13.7: Offensive ML Use Cases: This table categorizes how machine learning amplifies cyberattacks by enabling automated content generation, exploiting system vulnerabilities, and increasing attack sophistication; it details the typical ML model, targeted weakness, and resulting advantage for each offensive application. Understanding these use cases is important for developing effective defenses against learning-enabled threats.

Offensive Use Case	ML Model Type	Targeted System Vulnerability	Advantage of ML
Phishing and Social Engineering	Large Language Models (LLMs)	Human perception and communication systems	Personalized, context-aware message crafting

Offensive Use Case	ML Model Type	Targeted System Vulnerability	Advantage of ML
Reconnaissance and Fingerprinting	Supervised classifiers, clustering models	System configuration, network behavior	Scalable, automated profiling of system behavior
Exploit Generation	Code generation models, fine-tuned transformers	Software bugs, insecure code patterns	Automated discovery of candidate exploits
Data Extraction (Inference Attacks)	Classification models, inversion models	Privacy leakage through model outputs	Inference with limited or black-box access
Evasion of Detection Systems	Adversarial input generators	Detection boundaries in deployed ML systems	Crafting minimally perturbed inputs to evade filters
Hardware-Level Attacks	Deep learning models	Physical side-channels (for example, power, timing, EM)	Learning leakage patterns directly from raw signals

Although these applications differ in technical implementation, they share a common foundation: the adversary replaces a static exploit with a learned model capable of approximating or adapting to the target's vulnerable behavior. This shift increases flexibility, reduces manual overhead, and improves robustness in the face of evolving or partially obscured defenses.

What makes this class of threats particularly significant is their favorable scaling behavior. Just as accuracy in computer vision or language modeling improves with additional data, larger architectures, and greater compute resources, so too does the performance of attack-oriented machine learning models. A model trained on larger corpora of phishing attempts or power traces, for instance, may generalize more effectively, evade more detectors, or require fewer inputs to succeed. The same ecosystem that drives innovation in beneficial AI, including public datasets, open-source tooling, and scalable infrastructure, also lowers the barrier to developing effective offensive models.

The dynamic creates an asymmetry between attacker and defender. Defensive measures are bounded by deployment constraints, latency budgets, and regulatory requirements, while attackers can scale training pipelines with minimal marginal cost. The widespread availability of pretrained models and public ML platforms further reduces the expertise required to develop high-impact attacks.

Examining these offensive capabilities serves a crucial defensive purpose. Security professionals have long recognized that effective defense requires understanding attack methodologies. This principle underlies penetration testing[35], red team exercises[36], and threat modeling throughout the cybersecurity industry.

In the machine learning domain, this understanding becomes essential because ML amplifies both defensive and offensive capabilities. The same computational advantages that make ML effective for legitimate applications—pattern recognition, automation, and scalability—also enhance adversarial capabilities. By examining how machine learning can be weaponized, security professionals can anticipate attack vectors, design defenses, and develop detection mechanisms.

Any comprehensive treatment of machine learning system security must therefore consider both the vulnerabilities of ML systems themselves and the ways in which machine learning can be weaponized against other components: software, data, and hardware alike. The offensive potential of machine-learned systems directly informs the design of resilient defenses.

13.6.1 Case study: Deep learning for SCA

To illustrate these offensive capabilities concretely, we examine a specific case where machine learning transforms traditional attack methodologies. One of the most well-known and reproducible demonstrations of deep-learning-assisted SCA is the SCAAML framework (Side-Channel Attacks Assisted with Machine Learning) (Bursztein et al. 2024; Bursztein and Picod 2019). Developed by researchers at Google, SCAAML provides a practical implementation of the attack pipeline described earlier.

Figure 13.16 illustrates how cryptographic computations produce data-dependent power consumption signatures that reveal algorithmic state. These variations, while subtle, are measurable and reflect the internal state of the algorithm at specific points in time. The inset's three traces correspond to distinct processed values (the binary labels 0000, 1111, and 0101): because each value draws a measurably different amount of power, their amplitude separation is exactly the feature a model exploits to classify the secret bits a single intermediate computation depends on.

35 | **Penetration Testing** (from "penetration" of security perimeters): Authorized simulated attacks formalized in the 1960s for military systems and a $1.7 billion market in 2022. For ML systems, traditional pen testing is necessary but insufficient: it finds infrastructure vulnerabilities (SQL injection, misconfigurations) but misses ML-specific attack surfaces like adversarial inputs, model extraction via API queries, and training data poisoning that require specialized ML red teaming.

36 | **Red Team Exercises** (from Cold War military terminology for the adversary force): Unlike penetration testing's narrow technical scope, red teams simulate sophisticated multi-vector attacks over weeks or months, including social engineering and physical access. For ML systems, red-team exercises can encompass adversarial prompt engineering, training data poisoning simulations, and model extraction attempts; public safety-evaluation reports from OpenAI, Anthropic, and Google illustrate this pattern for pre-deployment safety evaluation.

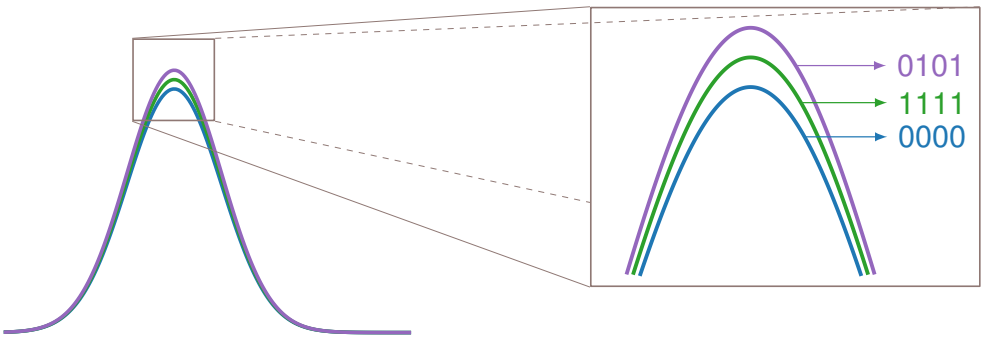


Figure 13.16: Power Traces: Cryptographic computations reveal subtle, data-dependent variations in power consumption that reflect internal states during specific operations.

In traditional side-channel attacks, experts rely on statistical techniques to extract these differences. However, a neural network can learn to associate the shape of these signals with the specific data values being processed, effectively learning to decode the signal in a manner that mimics expert-crafted models, yet with enhanced flexibility and generalization. The model is trained on labeled examples of power traces and their corresponding intermediate values (for example, output of an S-box operation). Over time, it learns to associate patterns in the trace with secret-dependent computational behavior. This transforms the key recovery task into a classification problem, where the goal is to infer the correct key byte based on trace shape alone.

In a SCAAML guide, Bursztein and Picod (2019) trained a convolutional neural network (CNN) to extract AES keys from power traces collected on an STM32F415 microcontroller running the open-source TinyAES implementation. The model was trained to predict intermediate values of the AES algorithm, such as the output of the S-box[37] in the first round, directly from raw power traces. The trained model recovered the full 128-bit key using only a small number of traces per byte.

Figure 13.17 shows the experimental hardware configuration: a ChipWhisperer setup with a custom STM32F target board that executes AES operations while allowing external equipment to monitor power consumption with high temporal precision. This setup demonstrates how even inexpensive, low-power embedded devices can leak information through side channels that machine learning models can learn to exploit.

Subsequent work expanded on this approach by introducing long-range models capable of exploiting broader temporal dependencies in the traces, improving performance even under noise and desynchronization (Bursztein et al. 2024). These developments highlight the potential for machine learning models to serve as offensive cryptanalysis tools, especially in the analysis of secure hardware.

The implications extend beyond academic interest. As deep learning models continue to scale, their application to side-channel contexts is likely to lower the cost, skill threshold, and trace requirements of hardware-level attacks, posing a growing challenge for the secure deployment of embedded machine learning systems, cryptographic modules, and trusted execution environments.

Adversarial machine learning dramatically lowers the skill threshold required to execute sophisticated side-channel and extraction attacks, rendering perimeter defenses insufficient. Surviving in an environment where attacks are automated and continuous demands comprehensive, multi-layered defense mechanisms that protect the data, the model, and the underlying infrastructure simultaneously.

[37] **S-box (Substitution Box)**: A nonlinear lookup table in block ciphers like AES that maps each input byte to a different output byte, providing the "confusion" property that makes ciphertext unpredictable from plaintext. S-box operations are the primary side-channel target because the output depends jointly on the plaintext and the secret key: a neural network trained on power traces during S-box computation can recover the key byte-by-byte, transforming cryptanalysis into a classification problem.

13.7 Comprehensive Defense Architectures

A determined adversary who bypasses API rate limits, defeats input sanitization, and begins querying a model with adversarial examples will not be stopped by any single defense. A single line of defense is a single point of failure. Secure ML systems therefore need defense in depth: overlapping layers that ensure a compromise in one mechanism does not grant full control of data, model behavior, runtime execution, and hardware trust.

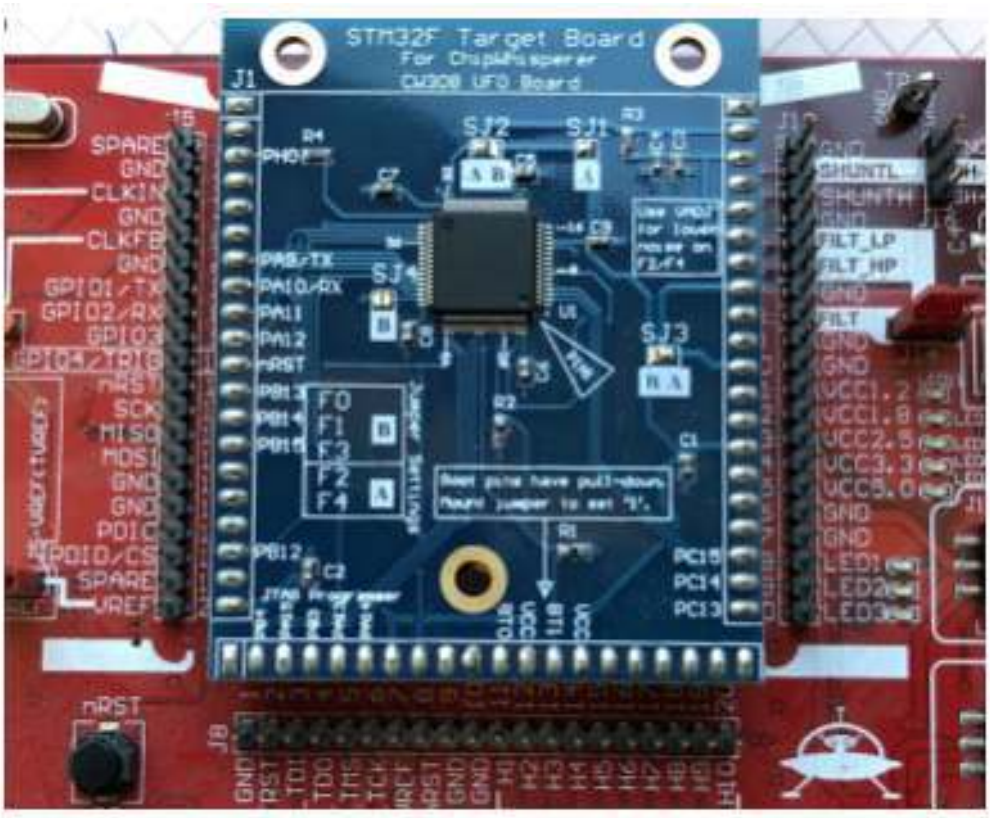


Figure 13.17: STM32F Target Board: Enables monitoring of power consumption during AES operations on the microcontroller, highlighting side-channel vulnerabilities that can be exploited by machine learning models.

Figure 13.18 visualizes the four layers. Hardware foundations such as trusted execution environments and secure boot anchor trust below the operating system. System-layer measures such as OS isolation and access control secure runtime operations. Model-layer defenses such as adversarial training[38], input validation, and output monitoring protect the learned function. Data-layer protections such as differential privacy, encryption, and provenance tracking reduce what the model and its operators can reveal. The ordering matters because each layer assumes the layer below has not already failed.

Layered defense (also known as defense in depth) makes this dependency explicit. ML systems are vulnerable to input manipulation, data leakage, model extraction, runtime abuse, and hardware compromise because data, model behavior, and infrastructure are tightly coupled. Data-level privacy techniques such as differential privacy and federated learning are only as secure as the runtime monitoring, model protections, and hardware trust anchors that support them; hardware trust anchors are only useful if the software layers above them preserve the intended policy.

38 **Robustness-Privacy Trade-off**: Improving adversarial robustness (for example, through adversarial training) can increase susceptibility to membership inference attacks. The robust model's decision boundary may fit the training distribution's "manifold" more tightly, making it easier for an attacker to distinguish training samples from unseen data (Song et al. 2019; Shokri et al. 2017). This trade-off forces fleet designers to explicitly prioritize between security and privacy invariants.

13.7.1 Privacy-preserving data techniques

The data layer forces the first privacy decision: keep raw records centralized, keep them on client devices, compute over encrypted values, or replace them with synthetic substitutes. That decision is architectural before it is algorithmic because it fixes the same system variables this chapter has tracked: information leakage, model utility, runtime cost, and the strength of the guarantee. ML workflows often depend on raw, high-fidelity data to train effective models, so privacy-preserving techniques should be read as different ways to move or bound that exposure rather than as interchangeable tools.

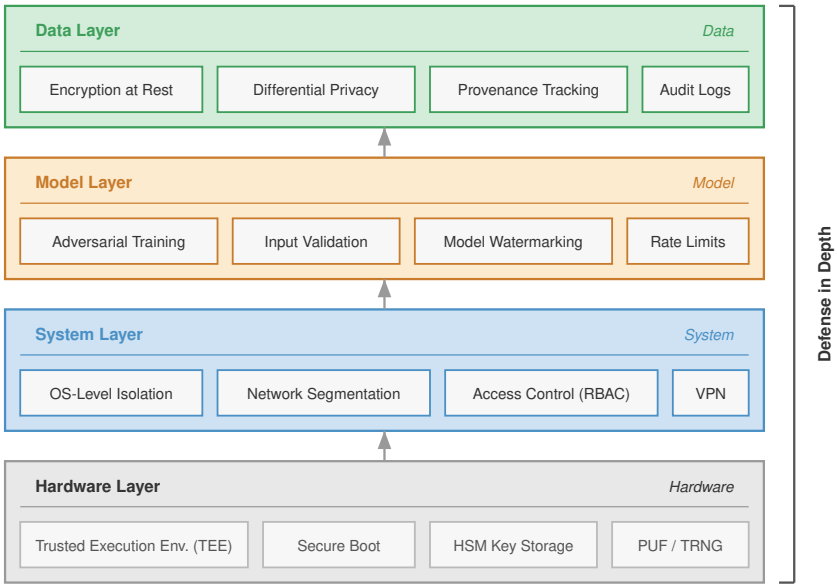


Figure 13.18: Layered Defense Stack: Machine learning systems require multi-faceted security strategies that progress from foundational hardware protections to data-centric privacy techniques, building trust across all layers. This architecture integrates safeguards at the hardware, system, model, and data levels to mitigate threats and ensure robust deployment in production environments.

13.7.1.1 Federated learning

Differential privacy adds noise to protect individual records, but some domains require a more structural solution: keeping raw data on the device entirely. In healthcare, finance, and other regulated domains, policy, consent, or data-use agreements may forbid centralizing raw records, making federated architectures a natural fit.

> **Lighthouse 13.1: Archetype C (Federated MobileNet): The need for privacy**
>
> Archetype C (Federated MobileNet) (section 1.6.1) represents the class of systems where privacy is a hard constraint, not an optimization. For a fleet of health monitors processing cardiac data, if policy or consent forbids centralizing raw signals, federated learning becomes a natural architectural choice for satisfying the privacy-utility trade-off: collective intelligence without data centralization.

While differential privacy adds mathematical guarantees to data processing, federated learning (FL) offers a complementary approach that reduces privacy risks by restructuring the learning process itself. Section 11.5 examines the FL protocols and weighted aggregation mechanisms that provide structural privacy protection by keeping data localized on client devices. However, FL alone does not guarantee privacy. The gradient updates that clients transmit can leak sensitive information, making FL a necessary but insufficient component of a comprehensive privacy strategy.

13.7.1.2 Gradient leakage and privacy-preserving computation

Gradient inversion attacks (such as Deep Leakage from Gradients) demonstrate that an adversary with access to a client's raw update can reconstruct the original training data (for example, images or

text) with high fidelity (Zhu et al. 2019; Nasr et al. 2019). The gradient vector encodes information about the data that produced it, creating a channel for information leakage even when raw data never leaves the device. The defense against this vulnerability is secure aggregation, a cryptographic protocol that ensures the server can compute the aggregate update without ever observing any individual client's contribution. Secure aggregation is one component of the broader layered defense framework summarized in figure 13.19, which places it alongside hardware-level primitives, system-level monitoring, model-level protections, and data-governance mechanisms such as differential privacy and federated learning.

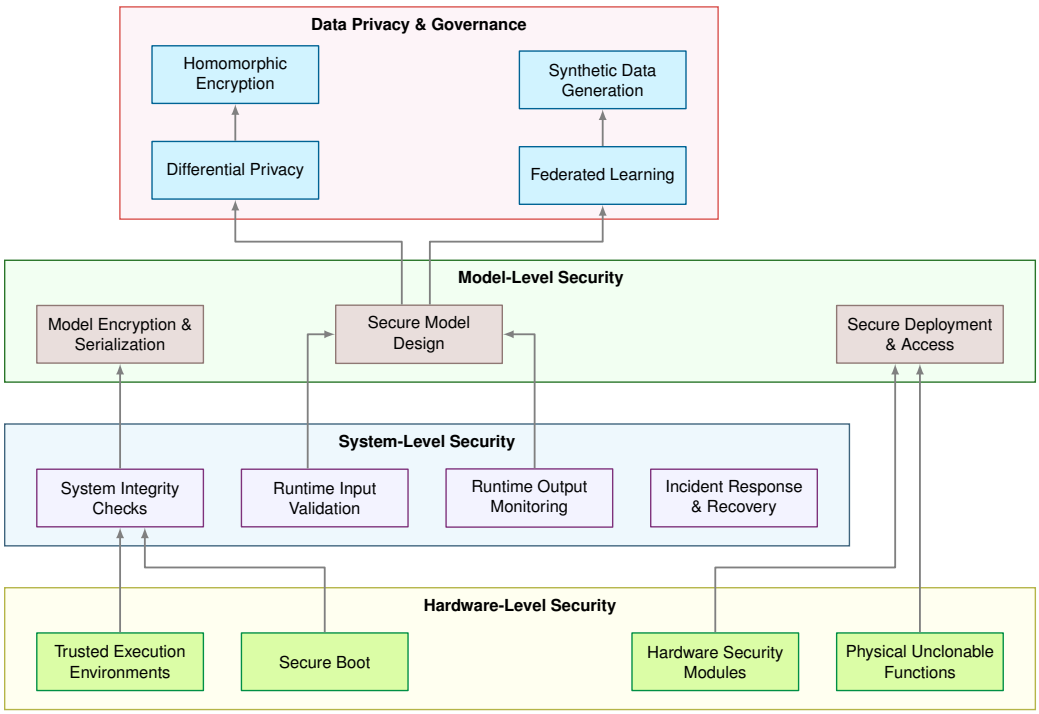


Figure 13.19: Layered Defenses Including Secure Aggregation: Four defense layers from hardware to data governance, with secure aggregation sitting in the data-governance layer alongside differential privacy, federated learning, and homomorphic encryption. Hardware-level security (TEEs, secure boot, HSMs) anchors trust in silicon; system-level security provides runtime validation and monitoring; model-level security protects weights and deployment; data privacy and governance mechanisms protect training data.

This vulnerability drives the need for secure aggregation protocols (Bonawitz et al. 2017) and differential privacy, but the trust boundary determines which version of DP applies. **Central differential privacy** assumes a trusted curator or aggregation service can see raw records or raw updates before releasing a noisy aggregate. **Local differential privacy** randomizes each client contribution before any server sees it, protecting against a stronger adversary but usually paying a larger utility cost. Secure aggregation sits between those regimes: the server sees an aggregate rather than individual updates, and DP noise can then be applied so the released aggregate reveals bounded information about any participant. Interestingly, the gradient compression techniques examined in section 6.6 for communication efficiency can also affect privacy properties: compressed gradients may leak less information than full-precision updates, though the interaction between compression and privacy guarantees requires careful analysis. Federated deployments therefore layer FL, secure aggregation, and the appropriate DP threat model to achieve formal privacy guarantees beyond structural protection alone. Google Gboard's federated learning deployment exemplifies this layered approach.

Lighthouse 13.2: Archetype C (Federated MobileNet): Google Gboard federated learning

For the Archetype C mobile/federated workload, Google's Gboard keyboard, illustrated in figure 11.2 as an on-device keyboard application, demonstrates how security mechanisms can layer atop the FL protocol. From a privacy perspective, the pattern combines three defense mechanisms: FL keeps raw typing data on-device; differential privacy can bound information leakage from gradient updates under an explicit ϵ; and secure aggregation lets servers observe aggregate updates rather than individual contributions. Published Gboard and federated-learning-at-scale reports show that this layered design can preserve useful model quality within mobile bandwidth budgets without centralizing raw typing data (Hard et al. 2018; Bonawitz et al. 2019).

Federated learning provides structural privacy by keeping raw data local, and differential privacy provides statistical privacy by bounding what outputs reveal about one record. A third family provides cryptographic privacy: **homomorphic encryption (HE)**[39] and **secure multiparty computation (SMPC)** allow models to perform inference or training over encrypted inputs (Gentry 2009; Yao 1982). The computational overhead of homomorphic operations often requires efficiency optimization techniques, including model compression (quantization reduces precision requirements for encrypted operations), architectural optimization (depthwise separable convolutions minimize encrypted multiplications), and hardware acceleration (specialized cryptographic accelerators), to maintain practical performance.

In the case of HE, operations on ciphertexts correspond to operations on plaintexts, enabling encrypted inference:

$$\text{Enc}(f(x)) = f(\text{Enc}(x))$$

This property supports privacy-preserving computation in untrusted environments, such as cloud inference over sensitive health or financial records. The computational cost of HE remains high, making it more suitable for fixed-function models and latency-tolerant batch tasks. SMPC[40], by contrast, distributes the computation across multiple parties such that no single party learns the complete input or output. This is particularly useful in joint training across institutions with strict data-use policies, such as hospitals or banks[41].

13.7.1.3 Synthetic data generation

Synthetic data changes the privacy decision from hiding raw records to replacing them with generated surrogates. Beyond cryptographic approaches like homomorphic encryption, a pragmatic alternative is **synthetic data generation**[42]. This approach offers an intuitive solution to privacy protection: if we can create artificial data that looks statistically similar to real data, we can train models without ever exposing sensitive information.

Synthetic data generation works by training a generative model (such as a GAN, variational autoencoder (VAE), or diffusion model) on the original sensitive dataset, then using this trained generator to produce new artificial samples. The key insight is that the generative model learns the underlying patterns and distributions in the data without memorizing specific individuals. When properly implemented, the synthetic data preserves statistical properties necessary for machine learning while removing personally identifiable information.

The generation typically follows three ordered stages:

1. **Distribution learning**: A generative model G_θ trains on real data $\mathcal{S}_{\text{real}} = \{x_1, x_2, \ldots, x_n\}$ to learn the data distribution $p(x)$.
2. **Synthetic sampling**: New samples $\mathcal{S}_{\text{synthetic}} = \{G_\theta(z_1), G_\theta(z_2), \ldots, G_\theta(z_m)\}$ are generated by sampling from random noise $z_i \sim \mathcal{N}(0, I)$.
3. **Validation**: The synthetic set is checked to ensure it maintains statistical fidelity to $\mathcal{S}_{\text{real}}$ while avoiding memorization of specific records.

By training generative models on real datasets and sampling new instances from the learned distribution, organizations can create datasets that approximate the statistical properties of the original data without retaining identifiable details (Goncalves et al. 2020).

39 **Homomorphic Encryption** (Greek *homos* "same" + *morphe* "form"): The name reflects the core property: operations on ciphertexts produce the "same form" of result as operations on plaintexts. Theoretical since the 1970s, fully homomorphic encryption became practical with Craig Gentry's 2009 PhD thesis. The systems cost remains steep: HE inference can run 1,000–10,000× slower than plaintext, limiting many ML applications to small models and low-throughput batch processing.

40 **SMPC Performance Overhead**: Secure multiparty computation can incur 1,000–10,000× computational overhead through cryptographic operations (secret sharing, garbled circuits), turning millisecond-scale inference into seconds, minutes, or longer depending on model size and protocol. The practical compromise is hybrid deployment: applying SMPC only to sensitive layers (final classification, embedding lookup) while running nonsensitive layers in plaintext, reducing overhead to 10–100× at the cost of partial information leakage from unprotected layers.

41 **SMPC (Secure Multi-Party Computation)**: Formalized in 1982 by Andrew Yao as the "Millionaires' Problem" (can two millionaires determine who is richer without revealing their wealth?). The framework enables hospitals to collaboratively train diagnostic ML models without sharing patient records. The key systems constraint is communication: SMPC requires each party to exchange encrypted shares for every operation, making network bandwidth rather than compute the dominant bottleneck in distributed ML training.

While appealing, synthetic data generation faces important limitations. Generative models can suffer from mode collapse, failing to capture rare but important patterns in the original data. More critically, sophisticated adversaries can potentially extract information about the original training data through generative model inversion attacks or membership inference. The privacy protection depends heavily on the generative model architecture, training procedure, and hyperparameter choices, making it difficult to provide formal privacy guarantees without additional mechanisms like differential privacy.

Consider a practical example where a hospital wants to share patient data for ML research while protecting privacy. They train a generative adversarial network (GAN) on 10,000 real patient records containing demographics, lab results, and diagnoses. The GAN learns to generate synthetic patients with realistic combinations of features (for example, diabetic patients typically have elevated glucose levels). The synthetic dataset of 50,000 artificial patients maintains clinical correlations necessary for training diagnostic models while containing no real patient information. However, the hospital also applies differential privacy during GAN training ($\epsilon = 1.0$) to prevent the model from memorizing specific patients, trading a 5 percent reduction in statistical fidelity for formal privacy guarantees.

Together, these techniques reflect a shift from isolating data as the sole path to privacy toward embedding privacy-preserving mechanisms into the learning process itself. Each method offers distinct guarantees and trade-offs depending on the application context, threat model, and regulatory constraints. Effective system design often combines multiple approaches, such as applying differential privacy within a federated learning setup, or employing homomorphic encryption for important inference stages, to build ML systems that are both useful and respectful of user privacy.

Across federated learning, differential privacy, homomorphic encryption, secure multi-party computation, and synthetic data, the privacy-preserving approaches differ in the guarantees they offer and in their system-level implications. The choice of mechanism therefore depends on computational constraints, deployment architecture, and regulatory requirements. A later comparison returns to these methods after the chapter introduces the deployment mechanisms needed to interpret their system costs. For now, the key point is that privacy guarantees, runtime overhead, operational maturity, and use case fit must be evaluated together rather than chosen as independent properties.

13.7.2 Case study: GPT-2 data extraction attack

Before turning to secure model design, the GPT-2 extraction case shows why model-level leakage cannot be solved by storage access control alone. In 2020, researchers demonstrated that large language models could leak sensitive training data through carefully crafted prompts (Carlini et al. 2021). The research team systematically queried OpenAI's GPT-2 model to extract verbatim content from its training dataset, revealing privacy vulnerabilities in large-scale language models.

The attack proved remarkably successful at extracting sensitive information directly from the model's outputs. By repeatedly querying the model with prompts like "My name is" followed by attempts to continue famous quotes or repeated phrases, researchers extracted personal information such as email addresses and phone numbers, verbatim passages from copyrighted books, and other identifiable snippets that had been memorized from the training data.

The technical approach exploited GPT-2's memorization of rare or repeated text sequences. The researchers combined four techniques:

- **Prompt engineering**: Crafted inputs triggered memorized sequences.
- **Continuation attacks**: Partial quotes or names elicited full sensitive information.
- **Statistical analysis**: Output patterns indicated verbatim memorization.
- **Verification**: Extracted data was cross-referenced with known public sources to confirm accuracy.

Generating 600,000 candidate samples, they confirmed 604 unique memorized training examples through manual verification.

The attack challenged assumptions about training data privacy. Large language models can act as unintentional databases, storing and retrieving sensitive information from their training data. The results violated privacy expectations that training data would be "forgotten" after model training, revealing that scale amplifies privacy risk: across the GPT-2 family (124M to 1.5B parameters), larger models memorized more training data than smaller models.

42 **Synthetic Data**: The market grew from $110 million (2019) to $1.1 billion (2023), driven by privacy regulations that make real data expensive to use. The fundamental trade-off is fidelity vs. privacy: synthetic datasets achieving 95 percent+ statistical fidelity to the original may still leak information about rare individuals through generative model memorization, which is why production systems combine synthetic generation with differential privacy guarantees.

The research revealed that simple data protection measures can be insufficient. Filtering, deduplication, and manual review reduce exposure, but models can still memorize rare or repeated sensitive strings, highlighting the tension between model utility and privacy protection. Techniques to prevent memorization such as differential privacy and aggressive data filtering can reduce model quality, creating challenging trade-offs for practitioners.

The result pushed practitioners toward layered mitigations: stronger PII filtering, deduplication, memorization audits using extraction-style probes, membership-inference defenses, research into machine unlearning that removes or bounds a record's learned influence after training, and policy discussions about training data rights and transparency. Differential privacy is one rigorous option when a formal privacy bound is required, but it must be evaluated against the utility and compute costs described by the privacy mechanisms in this chapter (Carlini et al. 2021).

13.7.3 Secure model design

The GPT-2 extraction attack demonstrates that even with strong data-level access controls, the model itself can leak information through its outputs, structure, or learned behavior. Differential privacy reduces this risk only when applied during training or analysis with an explicit privacy budget; it is not a post hoc patch for a model that has already memorized sensitive strings. This limitation motivates model-level security: architectural and design choices that complement data protections by reducing the model's capacity to serve as an information conduit.

Security must therefore begin at the design phase of a machine learning system. While downstream mechanisms such as access control and encryption protect models once deployed, many vulnerabilities can be mitigated earlier through architectural choices, defensive training strategies, and mechanisms that embed resilience directly into the model's structure or behavior. By considering security as a design constraint, system developers can reduce the model's exposure to attacks, limit its ability to leak sensitive information, and provide verifiable ownership protection.

One important design strategy is to build robust-by-construction models that reduce the risk of exploitation at inference time. For instance, models with confidence calibration or abstention mechanisms can be trained to avoid making predictions when input uncertainty is high. These techniques can help prevent overconfident misclassifications in response to adversarial or out-of-distribution inputs. Models may also employ output smoothing, regularizing the output distribution to reduce sharp decision boundaries that are especially susceptible to adversarial perturbations.

Certain application contexts may also benefit from choosing simpler or compressed architectures. Limiting model capacity can reduce opportunities for memorization of sensitive training data and complicate efforts to reverse-engineer the model from output behavior. For embedded or on-device settings, smaller models are also easier to secure, as they typically require less memory and compute, lowering the likelihood of side-channel leakage or runtime manipulation.

Another design-stage consideration is the use of model watermarking[43], a technique for embedding verifiable ownership signatures directly into the model's parameters or output behavior (Adi et al. 2018). A watermark might be implemented, for example, as a hidden response pattern triggered by specific inputs, or as a parameter-space perturbation that does not affect accuracy but is statistically identifiable.

For example, in a keyword spotting system deployed on embedded hardware for voice activation (for example, "Hey Alexa" or "OK Google"), a secure design might use a lightweight convolutional neural network with confidence calibration to avoid false activations on uncertain audio. The model might also include an abstention threshold, below which it produces no activation at all. To protect intellectual property, a designer could embed a watermark by training the model to respond with a unique label only when presented with a specific, unused audio trigger known only to the developer. These design choices improve robustness and accountability while supporting future verification in case of IP disputes or performance failures in the field.

In high-risk applications, such as medical diagnosis, autonomous vehicles, or financial decision systems, designers may also prioritize interpretable model architectures, such as decision trees, rule-based classifiers, or sparsified networks, to enhance system auditability. These models are often easier to understand and explain, making it simpler to identify potential vulnerabilities or biases. Using interpretable models allows developers to provide clearer insights into how the system arrived at a particular decision, which is important for building trust with users and regulators.

43 **Model Watermarking**: IP protection technique introduced by Adi et al. (2018), where a model is trained to produce a specific "signature" response on secret trigger inputs known only to the owner. In their experiments, trigger-set watermarks preserved normal test accuracy while making the secret trigger set reliably identify the model owner. The systems challenge is robustness: watermarks must survive model fine-tuning, pruning, and distillation, the same operations attackers use to strip ownership marks from stolen models.

Model design choices often reflect trade-offs between accuracy, robustness, transparency, and system complexity. When viewed from a systems perspective, early-stage design decisions yield the highest value for long-term security. They shape what the model can learn, how it behaves under uncertainty, and what guarantees can be made about its provenance, interpretability, and resilience.

13.7.4 Secure model deployment

Secure design establishes the model's intended behavior, but deployment determines whether that behavior survives contact with users, operators, and attackers. A model's vulnerability is not solely determined by its training procedure or architecture; it also depends on how the artifact is serialized, packaged, deployed, and accessed during inference. As models move into edge devices, public APIs, and multi-tenant platforms, deployment security must preserve the integrity, confidentiality, and availability of model behavior.

Design-stage controls shape model behavior; deployment-stage controls protect the packaged artifact and the inference interface. These controls also have to survive ordinary optimization work such as quantization, pruning, and knowledge distillation, where performance improvements must not compromise artifact integrity, ownership evidence, or access-control boundaries.

Once training is complete, the model must be securely packaged for deployment. Storing models in plaintext formats, including unencrypted ONNX or PyTorch checkpoint files, can expose internal structures and parameters to attackers with access to the file system or memory. To mitigate this risk, models should be encrypted, obfuscated, or wrapped in secure containers. Decryption keys should be made available only at runtime and only within trusted environments. Additional mechanisms, such as quantization-aware encryption or integrity-checking wrappers, can prevent tampering and offline model theft.

Deployment environments must also enforce strong access control policies to ensure that only authorized users and services can interact with inference endpoints. Authentication protocols, including OAuth[44] tokens, mutual TLS[45], or API keys[46], should be combined with role-based access control (RBAC)[47] to restrict access according to user roles and operational context. Hosted model APIs commonly authenticate requests with provider-issued API keys, service tokens, or mutually authenticated service identities. These controls are not merely billing or availability mechanisms: strong identity verification is a prerequisite for the rate-limiting and anomaly-detection defenses described in section 13.4.1.3. Without it, an attacker conducting model extraction can execute a Sybil attack—querying the endpoint through thousands of distinct, unauthenticated sessions—and defeat per-identity rate limits entirely. Every OAuth token, mTLS certificate, or API key converts an anonymous query stream into an attributable identity, making systematic extraction economically and operationally detectable.

This key authenticates the client and allows the backend to enforce usage policies, monitor for abuse, and log access patterns. Secure implementations retrieve API keys from environment variables rather than hardcoding them into source code, preventing credential exposure in version control systems or application logs. Such key-based access control mechanisms are simple to implement but require careful key management and monitoring to prevent misuse, unauthorized access, or model extraction. Additional security measures in production deployments often include model integrity verification through SHA-256 hash checking, rate limiting to prevent abuse, input validation for size and format constraints, and comprehensive logging for security event tracking.

The secure deployment patterns established here integrate naturally with development workflows, ensuring security becomes part of standard engineering practice rather than an afterthought. Runtime monitoring (section 13.7.5) extends these protections to operational environments.

13.7.5 Runtime system monitoring

Secure deployment controls who can load the model and call the endpoint, but runtime monitoring asks whether the live system is still behaving like the approved system. Attackers may craft inputs that pass static checks, exploit learned behavior, or target system-level infrastructure after deployment. Production ML systems span cloud services, edge devices, and embedded systems, so defensive strategies must connect real-time observation to threat detection and incident response across each environment.

44 **OAuth (Open Authorization)**: Standard developed in 2006 and used at internet scale, enabling API access without exposing credentials. For ML inference APIs, OAuth 2.0 token-based authentication adds 5–15 ms of latency per request for token validation, a negligible cost for batch inference but a meaningful overhead for real-time serving at thousands of requests per second where every millisecond of latency budget matters.

45 **Mutual TLS (mTLS)**: Enhanced Transport Layer Security (introduced 1999) where both client and server authenticate via certificates. The 15–30 ms handshake overhead is a one-time cost per connection, amortized across subsequent requests via connection pooling. For ML model serving, mTLS ensures that only authenticated services can submit inference requests, preventing unauthorized model extraction through API access.

46 **API Keys**: Simple authentication tokens popularized by Google Maps API (2005). Studies show 10–15 percent of GitHub repositories accidentally contain leaked API keys. For ML services, a leaked API key is not merely a billing risk: it provides the attacker unlimited query access for model extraction, effectively converting a credential management failure into complete model theft at the cost of API compute credits.

47 **RBAC (Role-Based Access Control)**: Formalized by NIST in the 1990s, RBAC assigns permissions to roles rather than individuals, reducing administration overhead by 90 percent+ compared to per-user policies. For ML platforms, RBAC maps naturally to the ML lifecycle: data engineers access training data but not model weights, ML engineers access models but not production serving keys, and monitoring systems access predictions but not raw inputs, enforcing least-privilege across the pipeline.

Runtime monitoring must decide whether an input should reach the model, whether the output is behaving normally, and whether the serving system is still the one we deployed. The corresponding mechanisms are input validation, output monitoring, and system integrity checks.

13.7.5.1 Input validation

Input validation is the first line of defense at runtime. It ensures that incoming data conforms to expected formats, statistical properties, or semantic constraints before it is passed to a machine learning model. Without these safeguards, models are vulnerable to adversarial inputs, which are crafted examples designed to trigger incorrect predictions, or to malformed inputs that cause unexpected behavior in preprocessing or inference.

Machine learning models, unlike traditional rule-based systems, often do not fail safely. Small, carefully chosen changes to input data can cause models to make high-confidence but incorrect predictions (Goodfellow et al. 2014). Input validation helps detect and reject such inputs early in the pipeline.

Validation techniques range from low-level checks (for example, input size, type, and value ranges) to semantic filters (for example, verifying whether an image contains a recognizable object or whether a voice recording includes speech). For example, a facial recognition system might validate that the uploaded image is within a certain resolution range (for example, 224×224 to 1024×1024 pixels), contains RGB channels, and passes a lightweight face detection filter. This prevents inputs like blank images, text screenshots, or synthetic adversarial patterns from reaching the model. Similarly, a voice assistant might require that incoming audio files be between 1 and 5 seconds long, have a valid sampling rate (for example, 16 kHz), and contain detectable human speech using a speech activity detector (SAD)[48]. This ensures that empty recordings, music clips, or noise bursts are filtered before model inference.

[48] **SAD (Speech Activity Detector)**: Algorithm distinguishing speech from silence, noise, or music, essential for voice interfaces since the 1990s. Modern neural SADs operate in less than 10 ms latency at 95 percent+ accuracy, serving as a lightweight security gate: by filtering nonspeech inputs before the expensive automatic speech recognition (ASR) model, SAD prevents adversarial audio attacks (noise-embedded commands, ultrasonic triggers) from reaching the speech recognition system.

In generative systems such as DALL·E, Stable Diffusion, or Sora, input validation often involves prompt filtering. This includes scanning the user's text prompt for banned terms, brand names, profanity, or misleading medical claims. For example, a user prompt like "Generate an image of a medication bottle labeled with Pfizer's logo" might be rejected or rewritten due to trademark concerns. Filters may operate using keyword lists, regular expressions, or lightweight classifiers that assess prompt intent. These filters prevent the generative model from producing harmful, illegal, or misleading content before sampling begins.

In some applications, distributional checks are also used. These assess whether the incoming data statistically resembles what the model saw during training. For instance, a computer vision pipeline might compare the color histogram of the input image to a baseline distribution, flagging outliers for manual review or rejection.

These validations can be lightweight (heuristics or threshold rules) or learned (small models trained to detect distribution shift or adversarial artifacts). In either case, input validation acts as a preinference firewall: it reduces exposure to adversarial behavior, improves system stability, and increases trust in downstream model decisions.

13.7.5.2 Output monitoring

Even when inputs pass validation, adversarial or unexpected behavior may still emerge at the model's output. Output monitoring is a containment mechanism, not a prevention mechanism: it does not stop exploitation but detects its symptoms in the model's predictions and triggers a fallback before a wrong output propagates. These mechanisms observe confidence, prediction entropy, class distribution, or response patterns to flag deviations from expected behavior.

The signal a monitor watches depends on the model's modality. For a discriminative classifier, prediction confidence is the key target: a model that begins assigning high confidence to low-frequency classes, or whose output entropy collapses in ambiguous contexts, is signaling adversarial inputs or distribution shift. The same idea generalizes by modality: content-moderation models are watched for a mismatch between predicted "safe" labels and auxiliary reference signals; time-series models such as fraud detectors are watched for sudden score drops during high-volume periods that suggest tampering or evasion. In each case the monitor compares the live prediction stream against an expected distribution and triggers a fallback (escalate to human review, revert to a conservative baseline) when the two diverge.

Generative models, such as text-to-image systems, introduce unique output monitoring challenges. These models can produce high-fidelity imagery that may inadvertently violate content safety policies, platform guidelines, or user expectations. To mitigate these risks, postgeneration classifiers are commonly employed to assess generated content for objectionable characteristics such as violence, nudity, or brand misuse. These classifiers operate downstream of the generative model and can suppress, blur, or reject outputs based on predefined thresholds. Some systems also inspect internal representations (for example, attention maps[49] or latent embeddings) to anticipate potential misuse before content is rendered.

However, prompt filtering alone is insufficient for safety. Research has shown that text-to-image systems can be manipulated through implicitly adversarial prompts, which are queries that appear benign but lead to policy-violating outputs. The Adversarial Nibbler project introduces an open red teaming methodology that identifies such prompts and demonstrates how models like Stable Diffusion can produce unintended content despite the absence of explicit trigger phrases (Quaye et al. 2024; Parrish et al. 2023). These failure cases often bypass prompt filters because their risk arises from model behavior during generation, not from syntactic or lexical cues.

Figure 13.20 provides concrete examples revealing how innocuous prompts can trigger unsafe generations. Such examples highlight the limitations of pregeneration safety checks and reinforce the necessity of output-based monitoring as a second line of defense. This two-stage pipeline consisting of prompt filtering followed by post-hoc content analysis is essential for ensuring the safe deployment of generative models in open-ended or user-facing environments.

49 **Attention Maps**: Visualization of attention weights, building on the neural attention mechanism introduced by Bahdanau et al. (2015). In monitoring contexts, attention-style visualizations and related attribution signals can help analysts inspect which input regions or tokens influenced a model output, but they should be treated as diagnostic evidence rather than a standalone security guarantee.



Figure 13.20: Adversarial Prompt Evasion: Implicitly adversarial prompts bypass typical content filters by triggering unintended generations, revealing limitations of solely relying on pregeneration safety checks. These examples underscore the necessity of post-hoc content analysis as a complementary defense layer for robust generative AI systems.

In the domain of language generation, output monitoring plays a different but equally important role. Here, the goal is often to detect toxicity, hallucinated claims, or off-distribution responses. For example, a customer support chatbot may be monitored for keyword presence, tonal alignment, or semantic coherence. If a response contains profanity, unsupported assertions, or syntactically malformed text, the system may trigger a rephrasing, initiate a fallback to scripted templates, or halt the response altogether.

Effective output monitoring combines rule-based heuristics with learned detectors trained on historical outputs. These detectors flag deviations in real time and feed alerts into incident response pipelines. Unlike model-centric defenses such as adversarial training, which aim to improve model robustness, monitoring forms a layer of operational resilience after the model is fixed: in safety-important or policy-sensitive applications, it is what bounds the damage of an exploit the upstream defenses missed.

These principles appear in output filtering frameworks. For example, Llama Guard is an LLM-based input-output safeguard that classifies human-AI conversations against policy categories (Inan et al. 2023). Similarly, ShieldGemma, developed as part of Google's open Gemma model release, applies configurable scoring functions to detect and filter undesired outputs during inference (Google 2024). Both systems exemplify how safety classifiers and output monitors can be integrated into the runtime stack to support scalable, policy-aligned deployment of generative language models.

13.7.5.3 Integrity checks

While input and output monitoring focus on model behavior, system integrity checks ensure that the underlying model files, execution environment, and serving infrastructure remain untampered

throughout deployment. These checks detect unauthorized modifications, verify that the model running in production is authentic, and alert operators to suspicious system-level activity.

One of the most common integrity mechanisms is cryptographic model verification. Before a model is loaded into memory, the system can compute a cryptographic hash (for example, SHA-256)[50] of the model file and compare it against a known-good signature.

Access control and audit logging complement cryptographic checks. ML systems should restrict access to model files using role-based permissions and monitor file access patterns. For instance, repeated attempts to read model checkpoints from a nonstandard path, or inference requests from unauthorized IP ranges, may indicate tampering, privilege escalation, or insider threats. In cloud environments, container- or VM-based isolation[51] helps enforce process and memory boundaries, but these protections can erode over time due to misconfiguration or supply chain vulnerabilities.

For example, in a regulated healthcare ML deployment[52], integrity checks might include three gates:

- **Model integrity**: Verify the model hash against a signed manifest.
- **Dependency integrity**: Validate that the runtime environment uses only approved Python packages.
- **Execution integrity**: Check that inference occurs inside a signed and attested virtual machine.

These checks ensure compliance with regulations like HIPAA[53]'s integrity requirements and GDPR's accountability principle, limit the risk of silent failures, and create a forensic trail in case of audit or breach.

Some systems also implement runtime memory verification, such as scanning for unexpected model parameter changes or checking that memory-mapped model weights remain unaltered during execution. While more common in high-assurance systems, such checks are becoming more feasible with the adoption of secure enclaves and trusted runtimes.

Taken together, system integrity checks play an important role in protecting machine learning systems from low-level attacks that bypass the model interface. When coupled with input/output monitoring, they provide layered assurance that both the model and its execution environment remain trustworthy under adversarial conditions.

Integrity checks become ML supply-chain controls only when they connect every deployable artifact. A signed model file is insufficient if the training dataset, dependency lockfile, preprocessing code, registry promotion event, container image, and deployment attestation cannot be tied to the same release record. That release record gives rollback and forensic analysis a concrete object: operators can ask which data snapshot produced the checkpoint, which dependencies built the serving image, which registry principal promoted it, and which production endpoints loaded it.

13.7.5.4 Response and rollback

When a security breach, anomaly, or performance degradation is detected in a deployed machine learning system, rapid and structured incident response is important to minimizing impact. The goal is not only to contain the issue but to restore system integrity and ensure that future deployments benefit from the insights gained. Unlike traditional software systems, ML responses may require handling model state, data drift, or inference behavior, making recovery more complex.

The first step is to define incident detection thresholds that trigger escalation. These thresholds may come from input validation (for example, invalid input rates), output monitoring (for example, drop in prediction confidence), or system integrity checks (for example, failed model signature verification). When a threshold is crossed, the system should initiate an automated or semi-automated response protocol.

One common strategy is **model rollback**, where the system reverts to a previously verified version of the model. For instance, if a newly deployed fraud detection model begins misclassifying transactions, the system may fall back to the last known-good checkpoint, restoring service while the affected version is quarantined. Rollback mechanisms require version-controlled model storage, typically supported by MLOps platforms such as MLflow, TFX, or SageMaker.

In high-availability environments, model isolation may be used to contain failures. The affected model instance can be removed from load balancers or shadowed in a canary deployment[54] setup.

50 **SHA-256** (Secure Hash Algorithm, NSA 2001): Produces 256-bit digests with no known practical collision attacks after 20+ years. For ML model integrity, computing a SHA-256 hash of a multi-gigabyte model file takes seconds and detects any modification, no matter how small. This makes cryptographic hashing the cheapest and most effective defense against model tampering during deployment, storage, and transfer between training and serving environments.

51 **Container vs. VM Isolation**: Linux containers (for example, Docker-style images) share the host kernel with 0-5 percent CPU overhead but weaker isolation; VMs provide hardware-level separation with 10–15 percent overhead. For ML serving, this is a security-performance tradeoff: containers enable rapid model scaling but share a kernel attack surface with cotenants, while VMs prevent cross-tenant model extraction at the cost of higher memory overhead per serving instance.

52 **Healthcare ML Compliance**: The FDA has approved 500+ AI-based medical devices since 2016 under 21 CFR Part 820, with approval cycles of 2–5 years costing $50+ million. The key systems constraint: any model update requires re-validation, creating tension between ML's iterative improvement cycle (retrain weekly) and regulatory approval timelines (validate over months), which drives the adoption of locked, versioned model deployment architectures.

This allows continued service with unaffected replicas while maintaining forensic access to the compromised model for analysis.

Traffic throttling is another immediate response tool. If an adversarial actor is probing a public inference API at high volume, the system can rate-limit or temporarily block offending IP ranges while continuing to serve trusted clients. This containment technique helps prevent abuse without requiring full system shutdown.

Once immediate containment is in place, investigation and recovery can begin. This may include forensic analysis of input logs, parameter deltas between model versions, or memory snapshots from inference containers. Generative and tool-using systems add five evidence requirements:

- **Prompts**: User and system messages show what the model was instructed to do.
- **Retrieved documents**: Retrieval context shows what external content shaped the response.
- **Tool calls**: Tool names, arguments, and results show which external side effects were attempted or completed.
- **Policy-classifier decisions**: Safety and compliance classifier outputs show whether guardrails detected the behavior.
- **Secret-access logs**: Credential and private-data access records show whether sensitive state was exposed.

These artifacts must preserve enough context to separate privacy breach, model-integrity failure, and ordinary model error. In regulated environments, organizations may also need to notify users or auditors, particularly if personal or safety-important data was affected.

Recovery typically involves retraining or patching the model. This must occur through a secure update process, using signed artifacts, trusted build pipelines, and validated data. To prevent recurrence, the incident should feed back into model evaluation pipelines by updating tests, refining monitoring thresholds, and hardening input defenses. For example, if a prompt injection attack[55] bypassed a content filter in a generative model, retraining might include adversarially crafted prompts, and the prompt validation logic would be updated to reflect newly discovered patterns.

Finally, organizations should establish postincident review practices. This includes documenting root causes, identifying gaps in detection or response, and updating policies and playbooks. Incident reviews help translate operational failures into actionable improvements across the design-deploy-monitor lifecycle.

13.7.6 Hardware security foundations

Input validation catches adversarial examples. Output monitoring detects anomalous predictions. Integrity checks verify model authenticity. However, all these software defenses share a critical assumption: the underlying execution environment is trustworthy. If an attacker compromises the operating system through a privilege escalation exploit, or gains physical access to an edge device's debug port, these careful software protections become meaningless since the attacker can simply disable them.

This limitation motivates hardware security mechanisms: protections that operate below the software layer and remain useful even when the operating system, runtime, or application code is compromised. Hardware-based security creates a root of trust anchored in silicon, which gives higher-layer defenses something to rely on. Machine learning systems deployed in edge devices, embedded systems, and untrusted cloud infrastructure often depend on that anchor because the attacker may have physical access, co-tenant access, or administrative access to parts of the stack.

The trust boundary becomes especially important when systems must satisfy regulatory requirements. Healthcare ML systems handling protected health information under HIPAA[56] must implement technical safeguards such as access control, audit controls, integrity protection, and transmission security. Systems processing EU citizens' data under GDPR[57] must also demonstrate appropriate technical and organizational measures, including privacy-by-design principles. Hardware security does not satisfy these requirements alone, but it provides the boot, runtime, key-management, and device-identity guarantees that software controls often assume.

Each hardware security primitive answers one trust question, as summarized in table 13.8. These hardware security mechanisms separate startup integrity, runtime isolation, key custody, and device identity. Secure Boot asks whether the system started from authentic code. Trusted Execution

53 **HIPAA ML Requirements**: The Health Insurance Portability and Accountability Act (1996) governs 600+ million US patient records. For ML systems specifically, the HIPAA Security Rule treats encryption at rest and in transit as addressable implementation specifications (covered entities must encrypt when reasonable and appropriate or document an equivalent alternative), and requires audit logs for all data access and Business Associate Agreements for cloud ML services (fines up to $1.5 million per incident). These requirements mean that training pipelines must log every data access, and model checkpoints containing patient-derived information must be encrypted, adding storage and I/O overhead.

54 **Canary Deployment** (named after coal mine canaries that detected gas before miners were affected): New model versions are rolled out to 1–5 percent of traffic before full deployment. For ML systems, canary deployments are essential because model failures are often statistical rather than binary: a backdoored model may pass unit tests while degrading accuracy on specific subpopulations, detectable only under real traffic patterns at canary scale.

55 **Prompt Injection** (by analogy with SQL injection): First widely documented in 2022, these attacks embed adversarial instructions within user input that override the model's system prompt. Unlike SQL injection, which exploits a syntactic boundary between code and data, prompt injection exploits the absence of any such boundary in LLMs: model instructions and user content share the same token stream, making robust detection fundamentally harder than traditional input sanitization.

Environments (TEEs) ask whether sensitive computation can run despite an untrusted host. Hardware Security Modules (HSMs) ask where cryptographic keys can be generated and used without exposing them to ordinary memory. Physical Unclonable Functions (PUFs) ask whether a device can prove a hardware-bound identity that cannot be cloned or extracted.

Table 13.8: Hardware Security Mechanisms: Each primitive answers a distinct trust question: startup integrity, runtime isolation, key custody, or device identity.

Mechanism	Trust Boundary	What It Protects
Secure Boot	Startup firmware and boot chain	Cryptographically verifies firmware and operating-system images before execution, preventing a tampered boot path from becoming the trusted base for the ML workload.
Trusted Execution Environments (TEEs)	Runtime code and data isolation	Isolates proprietary models, sensitive inputs, and intermediate state from an untrusted host operating system, hypervisor, or co-tenant process.
Hardware Security Modules (HSMs)	Key generation, storage, and use	Keeps cryptographic keys in tamper-resistant hardware rather than ordinary process memory, reducing exposure during model encryption, signing, and attestation workflows.
Physical Unclonable Functions (PUFs)	Device identity rooted in the device	Uses manufacturing variation to derive a hardware-bound identity, making device impersonation and key cloning harder than copying a software credential.

The mechanisms work best when they are composed across hardware, firmware, and software boundaries rather than selected as standalone products.

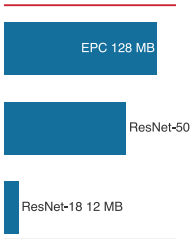
13.7.6.1 Hardware-software co-design

Modern ML systems require analysis of security trade-offs across the entire hardware-software stack, just as performance optimization analyzes compute, memory, and energy together. The interdependence between hardware security features and software defenses creates both opportunities and constraints that must be understood quantitatively.

Hardware security mechanisms introduce measurable overhead that must be factored into system design. ARM TrustZone world-switching adds approximately 300–1,000 cycles depending on processor generation and cache state (0.6–2 μs at 500 MHz) of latency per transition between secure and nonsecure worlds. Cryptographic operations in secure mode typically consume 15–30 percent additional power compared to normal execution, impacting battery life in mobile ML applications. Intel SGX context switching imposes 15–30 μs overhead per inference, representing 2 percent energy overhead for typical edge ML workloads.

Security features scale differently than computational resources. TEE memory limitations constrain model size regardless of available system memory. A ResNet-18 model is about 46.8 MB with FP32 weights or 11.7 MB with INT8 weights, while ResNet-50 is about 102.4 MB with FP32 weights or 25.6 MB with INT8 weights before activations and runtime buffers. These constraints create architectural decisions that must be made early in system design.

Different threat models and protection levels require quantitative trade-off analysis. For ML workloads requiring cryptographic verification, AES-256 operations add 0.1–0.5 ms per inference depending on model size and hardware acceleration availability.



Enclave memory caps which models can run securely.

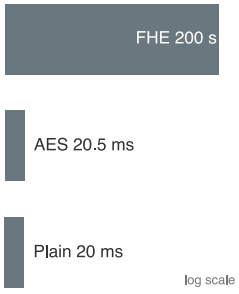
> Napkin Math 13.3: The tax of trusted compute
>
> **Problem**: A team deploying a health-monitoring model compares three security levels:
>
> 1. **Plaintext**: Standard inference.
> 2. **Encrypted transport (AES)**: Model and data are encrypted at rest and in transit.
> 3. **Encrypted compute (FHE)**: Inference performed on encrypted data.
>
> The comparison is whether each security level preserves real-time responsiveness.
>
> **Math**: Inference latency scales by the complexity of the security protocol.
>
> 1. **Plaintext**: 20 ms.

56 **HIPAA ML Requirements**: The Health Insurance Portability and Accountability Act (1996) governs 600+ million US patient records. For ML systems specifically, the HIPAA Security Rule treats encryption at rest and in transit as addressable implementation specifications (covered entities must encrypt when reasonable and appropriate or document an equivalent alternative), and requires audit logs for all data access and Business Associate Agreements for cloud ML services (fines up to $1.5 million per incident). These requirements mean that training pipelines must log every data access, and model checkpoints containing patient-derived information must be encrypted, adding storage and I/O overhead.

57 **GDPR (General Data Protection Regulation)**: Enacted by the EU in 2018 with fines up to 4 percent of global revenue, GDPR has levied billions of euros in penalties including €746 million against Amazon (2021). For ML systems, GDPR's "right to be forgotten" creates a unique technical challenge: removing an individual's influence from trained model weights requires either full retraining or machine unlearning techniques that approximate removal of a record's learned influence, both computationally expensive at scale.

2. **AES-256**: 20 ms + 0.5 ms = 20.5 ms (negligible tax).
3. **FHE**: 20 ms × 10,000× = 200 seconds.

Systems insight: Security is a latency-utility trade-off. Standard encryption (AES) is "nearly free" on hardware with cryptographic acceleration, but it only protects data *between computations*. Privacy-preserving compute (FHE) protects data *during computation* but costs 10,000× in performance. For a real-time monitor, this example makes FHE architecturally impractical. Confidential-compute offerings such as Intel SGX and NVIDIA H100 Confidential Computing illustrate a different design point: hardware isolation at much lower latency than FHE, with different trust assumptions.



Privacy-preserving computation costs orders more latency than encryption.

Homomorphic encryption operations can impose orders-of-magnitude computational overhead, with fully homomorphic encryption (FHE) usually at the higher end and somewhat homomorphic encryption (SHE) at the lower end. That makes them viable mainly for small models or offline scenarios where strong privacy guarantees justify the performance cost and explains why many production systems next consider hardware-enforced isolation.

13.7.6.2 Trusted execution environments

A TEE is first a trust-boundary decision: which parts of the model, key material, input data, or attestation path must remain protected even if the surrounding host is compromised. A **Trusted Execution Environment (TEE)**[58] is the hardware-isolated processor region that implements that boundary. TEEs enforce confidentiality, integrity, and runtime isolation, ensuring that even if the host operating system or application layer is attacked, sensitive operations within the TEE remain secure.

In the context of machine learning, TEEs are important for preserving the confidentiality of models, securing sensitive user data during inference, and ensuring that model outputs remain trustworthy. For example, a TEE can protect model parameters from being extracted by malicious software running on the same device, or ensure that computations involving biometric inputs, including facial data or fingerprint data, are performed securely. This capability is essential in applications where model integrity, user privacy, or regulatory compliance are nonnegotiable.

A well-documented example is Apple's Secure Enclave, which provides isolated execution and secure key storage for Apple devices. By separating cryptographic operations and biometric data from the main processor, the Secure Enclave is designed to keep user credentials and Face ID features protected even if the application processor kernel is compromised (Apple 2024b). The same isolation guarantee recurs across security-critical industries, from 5G control planes and mobile payments to wearable diagnostics and automotive ADAS, wherever model integrity, user privacy, or regulatory compliance is nonnegotiable.

Remote attestation closes the distributed trust loop: before two components exchange sensitive data or model updates, one component can verify cryptographically that the other is running approved code inside an expected hardware-protected boundary. What a TEE changes for an ML deployment is the trust boundary: inference and training execute inside an enclave, so intermediate activations, sensitive inputs, model weights, and update authenticity are shielded from system-level observation and tampering, and distributed components can exchange data over attested channels. What it costs is memory. Every protected byte must fit inside the enclave's bounded secure region, which forces a choice between compressing the model and partitioning inference so only the sensitive layers run inside the boundary.

The core security properties of a TEE are achieved through four mechanisms:

- **Isolated execution**: Code runs in a separate processor mode inaccessible to the normal-world operating system.
- **Secure storage**: Sensitive assets such as cryptographic keys or authentication tokens are stored in memory that only the TEE can access.
- **Integrity protection**: Code and data are verified before execution using hardware-anchored hashes or signatures.

58 **TEE (Trusted Execution Environment)**: Hardware-isolated processor regions that emerged from ARM's TrustZone (early 2000s), inspired by the military concept of compartmentalized information. For ML systems, TEEs create a fundamental memory constraint: secure enclaves (128 MB in SGX, expandable in TrustZone) must contain both model weights and intermediate activations, forcing either model compression or partitioned inference where only sensitive layers execute inside the enclave.

- **In-TEE data encryption**: Data processed inside the TEE is encrypted, so intermediate results remain inaccessible without appropriate keys managed internally by the TEE.

Together, these mechanisms let sensitive ML computation run even when the surrounding operating system or host environment is not fully trusted.

Several commercial platforms provide TEE functionality tailored for different deployment contexts. ARM TrustZone[59] offers secure and normal world execution on ARM-based systems and is widely used in mobile and IoT applications. Intel SGX[60] implements enclave-based security for cloud and desktop systems, enabling secure computation even on untrusted infrastructure. Qualcomm's Secure Execution Environment supports secure mobile transactions and user authentication. Apple's Secure Enclave remains a canonical example of a hardware-isolated security coprocessor for consumer devices (Apple 2024b).

Figure 13.21 illustrates a practical secure enclave architecture integrated into a system on chip (SoC) design. The enclave includes a dedicated processor, an AES engine, a true random number generator (TRNG), a public key accelerator (PKA), and a secure I2C interface to nonvolatile storage. These components operate in isolation from the main application processor and memory subsystem. A memory protection engine enforces access control, while cryptographic operations such as NAND flash encryption are handled internally using enclave-managed keys. By physically separating secure execution and key management from the main system, this architecture limits the impact of system-level compromises and establishes hardware-enforced trust (Apple 2024b).

This architecture underpins the secure deployment of machine learning applications on consumer devices. For example, Apple's Face ID system uses a Secure Enclave-protected biometric pipeline: TrueDepth camera data is transformed by a protected portion of the Neural Engine into a mathematical representation, then compared with enrolled facial data protected by the Secure Enclave. Face images captured during normal operation are discarded after the representation is computed, and Face ID data remains encrypted and device-local rather than being sent to Apple or included in backups (Apple 2024a). The ML systems lesson is that biometric models, templates, and comparison logic must be bound to hardware isolation and secure update paths, rather than treated as ordinary application data.

Despite their strengths, Trusted Execution Environments come with notable trade-offs. Implementing a TEE increases both direct hardware costs and indirect costs associated with developing and maintaining secure software. Integrating TEEs into existing systems may require architectural redesigns, especially for legacy infrastructure. Developers must adhere to strict protocols for isolation, attestation, and secure update management, which can extend development cycles and complicate testing workflows. TEEs can also introduce performance overhead, particularly when cryptographic operations are involved, or when context switching between trusted and untrusted modes is frequent.

Energy efficiency is another consideration, particularly in battery-constrained devices. TEEs typically consume additional power due to secure memory accesses, cryptographic computation, and hardware protection logic. In resource-limited embedded systems, these costs may limit their use. In terms of scalability and flexibility, the secure boundaries enforced by TEEs may complicate distributed training or federated inference workloads, where secure coordination between enclaves is required.

Market demand also varies. In some consumer applications, perceived threat levels may be too low to justify the integration of TEEs. Systems with TEEs may be subject to formal security certifications, such as Common Criteria or evaluation under the European Union Agency for Cybersecurity (ENISA), which can introduce additional time and expense. For this reason, TEEs best fit deployments where the expected threat model, including adversarial users, cloud tenants, and malicious insiders, justifies the investment.

Nonetheless, TEEs remain a critical hardware primitive in the machine learning security landscape. When paired with software- and system-level defenses, they provide a trusted foundation for executing ML models securely, privately, and verifiably, especially in scenarios where adversarial compromise of the host environment is a serious concern.

While TEEs provide runtime isolation once the system is running, they cannot protect against attacks that occur before the TEE is initialized. An attacker who compromises the boot process can modify firmware, inject malicious code, or disable security features before the TEE even be-

59 **ARM TrustZone**: Introduced in 2004, TrustZone partitions an ARM processor into secure and normal "worlds" with hardware-enforced isolation. For ML on mobile and edge devices, TrustZone can protect model weights and biometric data during inference, but the protection depends on whether the device vendor exposes and uses secure-world applications beyond key storage. The security lesson is architectural: hardware support alone does not protect an ML deployment unless the model, keys, and data path are actually placed inside the trusted boundary.

60 **Intel SGX Memory Constraints**: SGX enclaves are limited to approximately 128 MB of protected memory (EPC) on consumer processors, with EPC cache misses causing 100× performance penalties. A ResNet-50 requires approximately 102.4 MB for FP32 weights alone (25.6M parameters × 4 bytes), consuming 80 percent of EPC before activations. Inference latency jumps from 5 ms to 500 ms once the model exceeds EPC, making SGX practical only for small models under 10 MB or for protecting cryptographic keys.

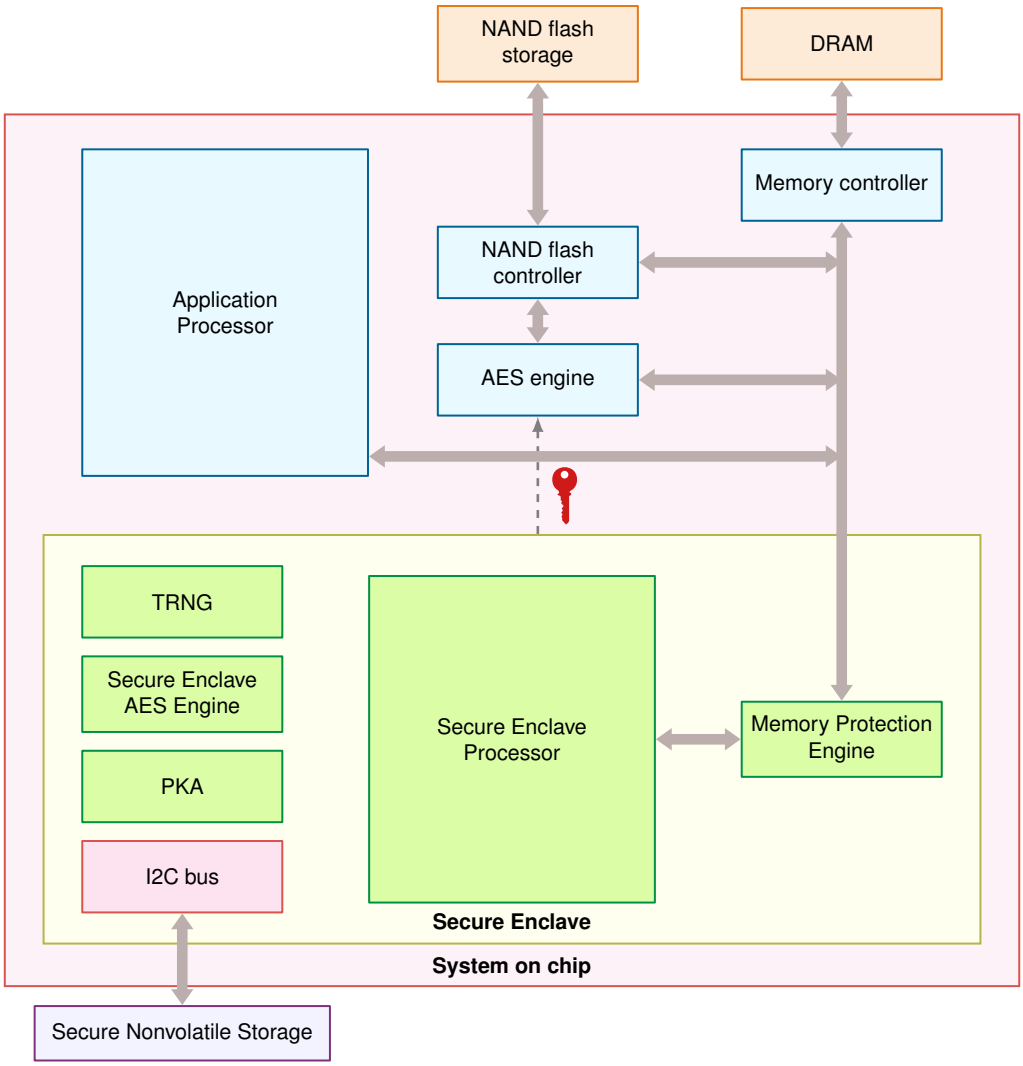


Figure 13.21: Secure Enclave Architecture: Hardware-isolated enclaves enhance system security by encapsulating sensitive data and cryptographic operations within a dedicated processor and memory. This design minimizes the attack surface and protects important keys even if the main application processor is compromised, providing a trusted execution environment for security-important tasks. Source: Apple.

gins executing. This temporal gap motivates Secure Boot, which establishes trust from the very first instruction the processor executes, creating a verified chain from power-on to secure enclave initialization.

13.7.6.3 Secure boot

Secure Boot is a mechanism that ensures a device only boots software components that are cryptographically verified and explicitly authorized by the manufacturer. At startup, each stage of the boot process, comprising the bootloader, kernel, and base operating system, is checked against a known-good digital signature. If any signature fails verification, the boot sequence is halted, preventing unauthorized or malicious code from executing. This chain-of-trust model establishes system integrity from the very first instruction executed.

In ML systems, especially those deployed on embedded or edge hardware, Secure Boot plays an important role. A compromised boot process may result in malicious software loading before the ML runtime begins, enabling attackers to intercept model weights, tamper with training data, or reroute

inference results. Such breaches can lead to incorrect or manipulated predictions, unauthorized data access, or device repurposing for botnets or crypto-mining.

For machine learning systems, Secure Boot offers several guarantees. First, it protects the integrity of the code path that handles model-related data during startup, preventing preruntime tampering before the ML runtime begins. Second, it ensures that only authenticated model binaries and supporting software are loaded, which helps guard against deployment-time model substitution. Third, Secure Boot can participate in secure update flows by ensuring that firmware or model-loading components verify signed changes before execution.

Secure Boot frequently works in tandem with hardware-based Trusted Execution Environments (TEEs) to create a more trusted execution stack. Figure 13.22 traces the layered verification sequence: platform firmware and boot components are verified before permitting execution of cryptographic operations or ML workloads (Regenscheid 2018). In embedded systems, this architecture improves resilience against preruntime compromise.

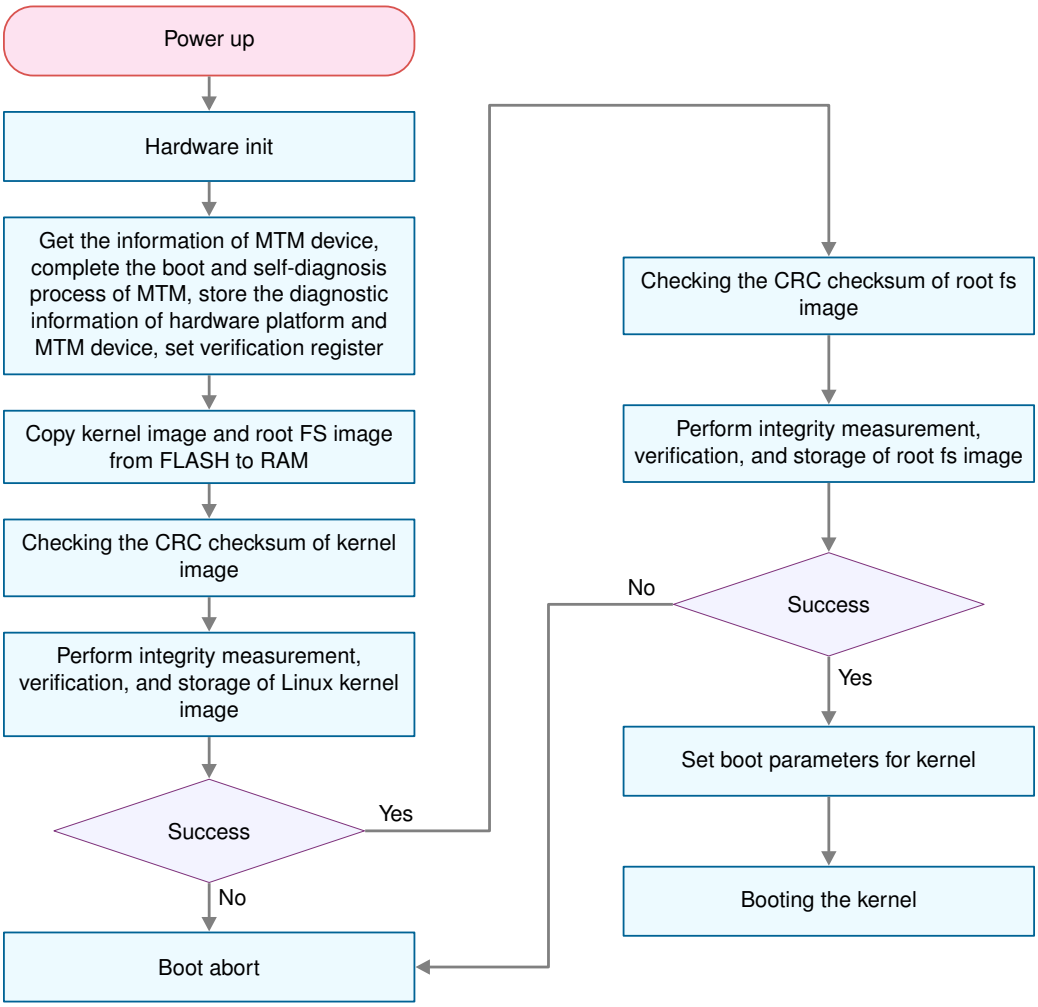


Figure 13.22: Secure Boot Sequence: Embedded systems employ a layered boot process to verify firmware and software integrity, establishing a root of trust before executing machine learning workloads and protecting against preruntime attacks. This architecture ensures only authenticated code runs, safeguarding model data and preventing unauthorized model substitution or modification during deployment.

A related real-world chain of trust appears in Apple devices that support Face ID, which uses machine learning for facial recognition. For Face ID to operate securely, the device stack from initial power-on through the biometric pipeline must be verifiably trusted.

Upon device startup, signed boot chains verify the application processor software and the Secure Enclave firmware before sensitive services are available. The firmware loaded onto the Secure Enclave is digitally signed by Apple, and unauthorized modification causes verification to fail. Once verified, the Secure Enclave participates in the broader trusted boot chain that protects biometric data and the cryptographic keys used by the device (Apple 2021a, 2024b).

After the device completes its secure boot sequence, the Face ID pipeline can authenticate the user. The TrueDepth camera projects and reads thousands of infrared dots to map a user's face, while the protected biometric pipeline computes a mathematical representation and compares it against enrolled facial data protected by the Secure Enclave. Face ID data is encrypted, remains on the device, and is not included in backups (Apple 2024a). The secure boot chain therefore protects not only the operating system, but also the trust assumptions under which biometric ML components execute.

To support continued integrity, Secure Boot also governs software updates. Apple devices use hardware-rooted authorization to install only Apple-signed system software and firmware versions that Apple is still signing, helping prevent tampered or downgraded components from entering the trusted stack (Apple 2021b). This process maintains a robust chain of trust over time, enabling secure evolution of the platform while preserving user privacy and device security.

While Secure Boot provides strong protection, its adoption presents technical and operational challenges. Managing the cryptographic keys used to sign and verify system components is complex, especially at scale. Enterprises must securely provision, rotate, and revoke keys, ensuring that no trusted root is compromised. Any such breach would undermine the entire security chain.

Performance is also a consideration. Verifying signatures during the boot process introduces latency, typically on the order of tens to hundreds of milliseconds per component. Although acceptable in many applications, these delays may be problematic for real-time or power-constrained systems. Developers must also ensure that all components, including bootloaders, firmware, kernels, drivers, and even ML models, are correctly signed. Integrating third-party software into a Secure Boot pipeline introduces additional complexity.

Some systems limit user control in favor of vendor-locked security models, restricting upgradability or customization. In response, open-source bootloaders such as U-Boot and coreboot have emerged, offering Secure Boot features while supporting extensibility and transparency[61]. To further scale trusted device deployments, device-identity standards such as the Device Identifier Composition Engine (DICE) (Trusted Computing Group 2018) and IEEE 802.1AR IDevID (IEEE 802.1 Working Group 2018) provide mechanisms for secure device identity, key provisioning, and cross-vendor trust assurance.

Secure Boot, when implemented carefully and complemented by trusted hardware and secure software update processes, forms the backbone of system integrity for embedded and distributed ML. It provides the assurance that the machine learning model running in production is not only the correct version, but is also executing in a known-good environment, anchored to hardware-level trust.

13.7.6.4 Hardware security modules

While TEEs and secure boot provide runtime isolation and integrity verification, **Hardware Security Modules (HSMs)** specialize in the cryptographic operations that underpin these protections. An HSM[62] is a tamper-resistant physical device designed to perform cryptographic operations and securely manage digital keys. HSMs are widely used across security-important industries such as finance, defense, and cloud infrastructure, and they are relevant for securing the machine learning pipeline—particularly in deployments where key confidentiality, model integrity, and regulatory compliance are important.

HSMs provide an isolated, hardened environment for performing sensitive operations such as key generation, digital signing, encryption, and decryption. Unlike general-purpose processors, they are engineered to withstand physical tampering and side-channel attacks, and they typically include protected storage, cryptographic accelerators, and internal audit logging. HSMs may be implemented as standalone appliances, plug-in modules, or integrated chips embedded within broader systems.

In machine learning systems, HSMs enhance security across several dimensions. They are commonly used to protect encryption keys associated with sensitive data that may be processed during

61 **Open-Source Secure Boot Stacks**: U-Boot (The U-Boot Project 2026a) is the de facto bootloader for embedded Linux systems, including Raspberry Pi-class devices used in edge ML deployments, and its UEFI documentation describes Secure Boot configuration (The U-Boot Project 2026b). coreboot (coreboot contributors 2026a) targets x86 server and workstation hardware, and its vboot documentation describes verified boot support, firmware measurement, and image signing (coreboot contributors 2026b). Both can participate in signature-verified or measured boot chains, but the trust root is platform-specific and may still be controlled by vendor hardware policy, firmware write protection, or platform keys. "Open-source" therefore applies to the bootloader code, not automatically to the root of trust that anchors the ML model's execution environment.

62 **HSM (Hardware Security Module)**: Enterprise HSMs perform 10,000+ RSA-2048 operations per second at $20,000–$100,000+ per unit, compared to 100,000+ operations per second on GPUs at $1,000+. The 10× throughput disadvantage is the price of tamper resistance: HSMs physically destroy keys when tampered with, a guarantee no software-only solution can provide and which certification or payment-security regimes may require for high-assurance key handling in production.

training or inference. These keys protect data at rest in model checkpoints and secure transmission of inference requests across networked environments. By ensuring that the keys are generated, stored, and used exclusively within the HSM, the system minimizes the risk of key leakage, unauthorized reuse, or tampering. In distributed ML deployments, HSMs serve as the root of trust for two additional workflows. First, they provision the unique device identities required to authenticate edge nodes in secure aggregation protocols for federated learning: each participating device proves membership with a certificate whose signing key was generated and stored in an HSM, preventing a compromised or spoofed node from injecting poisoned gradients into the aggregation. Second, they manage the key-wrapping operations necessary to deliver encrypted model weights directly into a Trusted Execution Environment—the HSM wraps the model-decryption key under the TEE's attestation-bound public key, ensuring that model weights are decrypted only inside the enclave and never appear in plaintext host memory.

HSMs also play a role in maintaining the integrity of machine learning models. In many production pipelines, models must be signed before deployment to ensure that only verified versions are accepted into runtime environments. The signing keys used to authenticate models can be stored and managed within the HSM, providing cryptographic assurance that the deployed artifact is authentic and untampered. Similarly, secure firmware updates and configuration changes, regardless of whether they pertain to models, hyperparameters, or supporting infrastructure, can be validated using signatures produced by the HSM.

In addition to protecting inference workloads, HSMs can be used to secure model training. During training, data may originate from distributed and potentially untrusted sources. HSM-backed protocols can help ensure that training pipelines perform encryption, integrity checks, and access control enforcement securely and in compliance with organizational or legal requirements. In regulated industries such as healthcare and finance, such protections are often necessary to satisfy required safeguards. For instance, the HIPAA Security Rule requires technical safeguards such as access control, audit controls, integrity protections, and transmission security, while treating encryption and some integrity controls as addressable implementation specifications. GDPR lists pseudonymization and encryption as examples of appropriate technical and organizational measures where they fit the risk.

These benefits are bought with a tamper-resistance tax. The dedicated, hardened silicon delivers roughly 10× lower cryptographic throughput than a general-purpose GPU at $20,000–$100,000+ per unit, and that single trade-off cascades into every constraint a resource-limited ML deployment cares about: the latency it adds to key exchange and on-the-fly decryption can break real-time inference budgets; its continuous secure operations draw power that shortens battery life on edge devices; its size and dedicated APIs force circuit-board and software redesign; and provisioning unique keys across a fleet of edge nodes turns identity management into a distributed-systems problem. Regulated deployments add a final cost: certification and compliance processes[63] such as FIPS 140-2[64] or Common Criteria can take longer than training the model the HSM protects.

Despite these operational complexities, HSMs remain a valuable option for machine learning systems that require high assurance of cryptographic integrity and access control. When paired with TEEs, secure boot, and software-based defenses, HSMs contribute to a multilayered security model that spans hardware, system software, and ML runtime.

HSMs provide robust cryptographic processing but require dedicated hardware modules and significant infrastructure investment. For resource-constrained embedded systems where adding external HSM hardware is impractical due to cost, size, or power constraints, an alternative approach derives cryptographic secrets directly from the chip's inherent physical properties. This capability is provided by Physical Unclonable Functions, which we examine next.

13.7.6.5 Physical unclonable functions

Physical Unclonable Functions (PUFs)[65] provide a hardware-intrinsic mechanism for cryptographic key generation and device authentication by exploiting physical randomness in semiconductor fabrication (Gassend et al. 2002). Unlike traditional keys stored in memory, a PUF generates secret values based on microscopic variations in a chip's physical properties. These variations are inherent to manufacturing processes and difficult to clone or predict, even by the manufacturer.

These variations arise from uncontrollable physical factors such as doping concentration, line edge roughness, and dielectric thickness. As a result, even chips fabricated with the same design masks

63 **HSM Certification**: FIPS 140-2 or Common Criteria certification takes 12–24 months and costs $500,000–$2 million per device family. Banking, government, and some healthcare procurement regimes may require Level 3+ certification for cryptographic operations. For ML systems in regulated industries, this certification timeline creates a deployment bottleneck: the HSM protecting model signing keys may take longer to certify than the model takes to train.

64 **FIPS 140-2** (Federal Information Processing Standard): Defines four security levels for cryptographic modules. Level 4 requires survival of physical attacks at -40 to +85 degrees Celsius with tamper detection that zeroizes keys within seconds. For ML systems handling classified data or operating under strict procurement requirements, FIPS 140-2 Level 3+ compliance may be mandatory, restricting HSM access to authorized personnel and slowing the rapid iteration cycles that ML development typically demands.

65 **PUF (Physical Unclonable Function)**: Named for the physical impossibility of cloning the microscopic manufacturing variations they exploit, PUFs generate device-unique cryptographic keys without storing secrets in memory. For edge ML devices, PUFs solve a critical deployment problem: each device can encrypt its local model with a hardware-derived key that exists nowhere else, ensuring that extracting a model from one device does not compromise the fleet.

exhibit small but measurable differences in timing, power consumption, or voltage behavior. PUF circuits amplify these variations to produce a device-unique digital output. When a specific input challenge is applied to a PUF, it generates a corresponding response based on the chip's physical fingerprint. Because these characteristics are effectively impossible to replicate, the same challenge will yield different responses across devices.

This challenge-response mechanism allows PUFs to serve several cryptographic purposes. They can be used to derive device-specific keys that never need to be stored externally, reducing the attack surface for key exfiltration. The same mechanism also supports secure authentication and attestation, where devices must prove their identity to trusted servers or hardware gateways. These properties make PUFs a natural fit for machine learning systems deployed in embedded and distributed environments.

In ML applications, PUFs offer unique advantages for securing resource-constrained systems. For example, consider a smart camera drone that uses onboard computer vision to track objects. A PUF embedded in the drone's processor can generate a private key to encrypt the model during boot. Even if the model were extracted, it would be unusable on another device lacking the same PUF response. That same PUF-derived key could also be used to watermark the model parameters, creating a cryptographically verifiable link between a deployed model and its origin hardware. If the model were leaked or pirated, the embedded watermark could help prove the source of the compromise.

PUFs also support authentication in distributed ML pipelines. If the drone offloads computation to a cloud server, the PUF can help verify that the drone has not been cloned or tampered with. The cloud backend can issue a challenge, verify the correct response from the device, and permit access only if the PUF proves device authenticity. These protections enhance trust not only in the model and data, but in the execution environment itself.

Figure 13.23 demonstrates how PUF operation depends on inherent physical variation. At a high level, a PUF accepts a challenge input and produces a unique response determined by the physical microstructure of the chip (Gao et al. 2020). Variants include optical PUFs, in which the challenge consists of a light pattern and the response is a speckle image, and electronic PUFs such as Arbiter PUFs (APUFs), where timing differences between circuit paths produce a binary output. Another common implementation is the SRAM PUF, which exploits the power-up state of uninitialized SRAM cells: due to threshold voltage mismatch, each cell tends to settle into a preferred value when power is first applied. These response patterns form a stable, reproducible hardware fingerprint.

Despite their promise, PUFs present several challenges in system design. Their outputs can be sensitive to environmental variation, such as changes in temperature or voltage, which can introduce instability or bit errors in the response. To ensure reliability, PUF systems must often incorporate error correction codes or helper data schemes. Managing large sets of challenge-response pairs also raises questions about storage, consistency, and revocation. Additionally, the unique statistical structure of PUF outputs may make them vulnerable to machine learning-based modeling attacks if not carefully shielded from external observation.

From a manufacturing perspective, incorporating PUF technology can increase device cost or require additional layout complexity. While PUFs eliminate the need for external key storage, thereby reducing long-term security risk and provisioning cost, they may require calibration and testing during fabrication to ensure consistent performance across environmental conditions and device aging.

Nevertheless, Physical Unclonable Functions remain a compelling building block for securing embedded machine learning systems. By embedding hardware identity directly into the chip, PUFs support lightweight cryptographic operations, reduce key management burden, and help establish root-of-trust anchors in distributed or resource-constrained environments. When integrated thoughtfully, they complement other hardware-assisted security mechanisms such as Secure Boot, TEEs, and HSMs to provide defense-in-depth across the ML system lifecycle.

13.7.6.6 Mechanisms comparison

The design choice is not whether hardware-backed protection is better than software-backed protection, but which primitive owns which trust boundary. While software-based defenses offer flexibility, they ultimately rely on the security of the hardware platform. As machine learning workloads

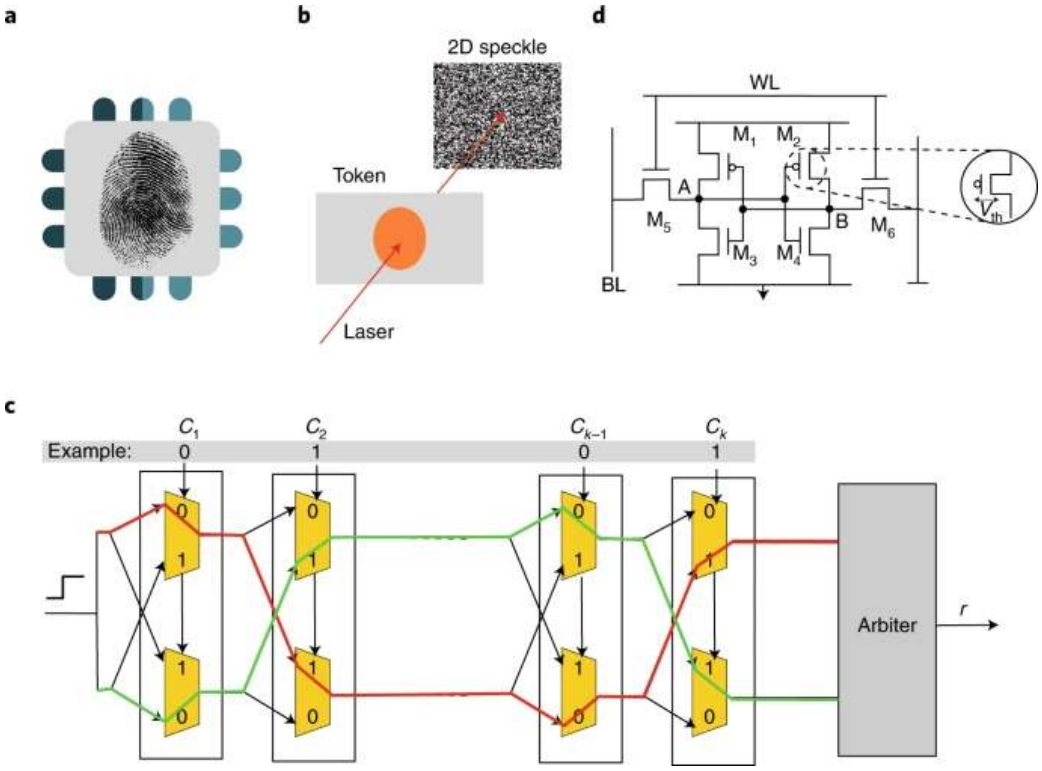


Figure 13.23: Physical Unclonable Functions: PUFs generate unique hardware fingerprints from inherent manufacturing variations. The figure shows several types: a chip-level fingerprint concept, optical PUFs that use laser speckle patterns, electronic arbiter PUFs that rely on timing differences in response to a challenge, and SRAM PUFs that exploit manufacturing variations in memory cells.

operate on edge devices, embedded platforms, and untrusted infrastructure, hardware-backed protections become important for maintaining system integrity, confidentiality, and trust.

Trusted Execution Environments (TEEs) provide runtime isolation for model inference and sensitive data handling. Secure Boot enforces integrity from power-on, ensuring that only verified software is executed. Hardware Security Modules (HSMs) offer tamper-resistant storage and cryptographic processing for secure key management, model signing, and firmware validation. Physical Unclonable Functions (PUFs) bind secrets and authentication to the physical characteristics of a specific device, enabling lightweight and unclonable identities.

These mechanisms address different layers of the system stack, ranging from initialization and attestation to runtime protection and identity binding, and complement one another when deployed together. Table 13.9 compares their roles, use cases, and trade-offs for machine learning system design.

Table 13.9: Hardware Security Primitives Compared: Machine learning systems use diverse hardware defenses (trusted execution environments, secure boot, hardware security modules, and physical unclonable functions) to establish trust and protect sensitive data across the system stack. This table compares how each mechanism addresses specific security challenges, from runtime isolation and integrity verification to key management and device identity, and emphasizes the associated trade-offs in performance and complexity.

Mechanism	Primary Function	Common Use in ML	Trade-offs
Trusted Execution Environment (TEE)	Isolated runtime environment for secure computation	Secure inference and on-device privacy for sensitive inputs and outputs	Added complexity, memory limits, performance cost; requires trusted code development

Mechanism	Primary Function	Common Use in ML	Trade-offs
Secure Boot	Verified boot sequence and firmware validation	Ensures only signed ML models and firmware execute on embedded devices	Key management complexity, vendor lock-in; startup performance impact
Hardware Security Module (HSM)	Secure key generation, storage, and cryptographic processing	Signing ML models, securing training pipelines, verifying firmware	High cost, integration overhead, limited I/O; requires infrastructure-level provisioning
Physical Unclonable Function (PUF)	Hardware-bound identity and key derivation	Model binding, device authentication, protecting IP in embedded deployments	Environmental sensitivity, modeling attacks; needs error correction and calibration

Together, these hardware primitives form the foundation of a defense-in-depth strategy for securing ML systems in adversarial environments. Their integration is especially important in domains that demand provable trust, such as autonomous vehicles, healthcare devices, federated learning systems, and important infrastructure.

Together with secure multi-party computation and gradient-compression choices from the privacy layer, the hardware mechanisms above form a menu rather than a recipe. The appropriate defense for any given deployment depends on three interacting factors: the threat model (who attacks and with what capabilities), the deployment context (what computational and latency budgets exist), and the regulatory environment (what legal mandates constrain design). A healthcare system training federated diagnostic models faces fundamentally different threats than a public-facing LLM chatbot, and each demands a distinct combination of the mechanisms surveyed in this chapter. Table 13.10 provides a defense selection framework that maps seven common deployment contexts to their primary threats, recommended defense combinations, and the quantified trade-offs that each combination imposes. The rows that prescribe differential privacy include specific privacy-budget values (ϵ); section 13.8 defines what ϵ bounds and why smaller values cost accuracy, so the prescriptions here can be read as deployment targets and revisited once that machinery is in hand.

Table 13.10: Defense Selection Framework: Maps deployment contexts to threat-specific defensive strategies with quantified trade-offs. The framework provides starting points for security architecture design, highlighting primary threats, recommended defense combinations, and key implementation trade-offs across seven common ML system deployment scenarios.

Deployment Context	Primary Threats	Recommended Defenses	Key Trade-offs
Healthcare ML (Federated diagnostic models)	Data leakage (HIPAA violation) Membership inference Unauthorized access	• Differential Privacy ($\epsilon \leq 4$) for training • Federated Learning across hospitals • TEE for inference on sensitive data • Audit logging and access control (RBAC)	2–5% accuracy loss acceptable for compliance; 50–100 ms inference latency from TEE overhead
Financial ML (Fraud detection API)	Model theft (IP loss) Adversarial evasion Data poisoning	• Model encryption (AES-256) at rest • HSM for cryptographic key management • Adversarial training on projected-gradient perturbations • Input validation + rate limiting (100 req/min) • Output confidence monitoring	HSM adds \$10–50K capital cost; rate limiting may impact legitimate high-frequency users
Edge ML (Mobile/IoT devices)	Physical access Side-channel attacks Model extraction	• Secure Boot (verified firmware) • ARM TrustZone or similar TEE • Model quantization + obfuscation • Encrypted model storage • Anti-tampering hardware (PUF)	TEE memory limits constrain model size < 50 MB; quantization required for large models; 15–30% power overhead from encryption
Cloud ML Training (Multi-tenant platform)	Data poisoning Backdoor injection Gradient leakage	• Secure data pipelines (provenance tracking) • Differential Privacy (DP-SGD, $\epsilon \approx 1$–10) • Gradient verification and anomaly detection • Secure aggregation (if federated) • Model watermarking for IP protection	Training time increases 30–120% with DP; gradient verification adds 10–15% compute overhead; federated aggregation requires secure communication protocols
Public-Facing LLM (Chatbot/API)	Prompt injection Data extraction (training leakage) Abuse/overuse	• Input sanitization (prompt filtering) • Output monitoring (PII detection) • Rate limiting (per-user quotas) • Response watermarking • Confidence thresholding (abstention)	Aggressive filtering may block 5–10% of legitimate requests; response time increases 50–100 ms for content filtering; watermarking may be detectable by sophisticated users
Multi-Party ML (Cross-organizational training)	Data sharing restrictions Honest-but-curious participants Privacy compliance (GDPR)	• Federated Learning (no raw data sharing) • SMPC for secure aggregation • Differential Privacy ($\epsilon \leq 1$) • Homomorphic Encryption (for sensitive operations)	Communication overhead: 10–100$\times$ more rounds than centralized tr SMPC adds 1,000$\times$+ compute cost; accuracy may degrade 5–15%; requires legal agreements for liability

Deployment Context	Primary Threats	Recommended Defenses	Key Trade-offs
Critical Infrastructure (Autonomous vehicles, power grids)	Supply chain compromise Real-time adversarial attacks Safety-critical failures	• Hardware attestation (Trusted Platform Module, or TPM, and PUF) • Secure Boot + runtime integrity checks • Redundant model validation • Fault injection detection • Fail-safe fallback mechanisms	Development cost +18 months additional engi 20–40% higher hardware costs; latency constraints limit cryptographic defenses requires certified hardware

Checkpoint 13.4: Mapping hardware threats to defenses

The chapter introduced seven hardware-level attack categories, four hardware-security primitives, and a framework mapping deployment contexts to defenses (table 13.10). Test whether you can apply that mapping.

- ☐ **Primitive matching**: Match fault injection on model weights and side-channel extraction from the inference pipeline to the most relevant hardware-security primitives.
- ☐ **Edge defense stack**: Select a combination of primitives for an edge inference deployment facing physical access, side-channel attacks, and model extraction.
- ☐ **Supply-chain gap**: Explain why supply-chain compromise is the hardest hardware threat to defend against with the four primitives, and connect it to the deployment context that treats it as a primary threat.
- ☐ **TEE vs. attestation**: Distinguish runtime TEE detection from boot-time hardware attestation in a deployment facing both fault injection and counterfeit hardware attacks.

A recurring theme across table 13.10 is the cost of privacy: differential privacy appears in three of seven deployment contexts, and each such row exposes an accuracy or systems cost that must be weighed against regulatory mandates and risk tolerance. The table should therefore be read as a deployment triage tool rather than as a universal recipe: serialization, artifact integrity, access control, runtime isolation, and privacy budgets each defend a different boundary. Even if the training process is perfectly secure and the collaborative architecture leak-free, the final published model may still inadvertently memorize and regurgitate the sensitive information it was trained on. To bound what a model's outputs can reveal about individual records, we must enforce a rigorous mathematical standard known as differential privacy.

13.8 Differential Privacy

Differential privacy is the next engineering move because encryption and access control protect data at rest or in transit, while memorization and inference attacks exploit the trained model itself. Suppose an adversary queries a medical diagnosis model using the exact attributes of a known patient. If the model's prediction changes significantly depending on whether that specific patient was included in the training dataset, the patient's privacy has been mathematically compromised.

Differential privacy addresses this by injecting carefully calibrated noise during training or analysis, providing a formal guarantee that the model's behavior is bounded whether any single individual opts in or out of the dataset. The mechanism converts privacy into a budget: lower privacy loss means more noise, more training cost, and usually lower utility. That budget is why DP belongs in the deployment architecture rather than only in a legal checklist.

Definition 13.4: Differential privacy

Differential Privacy is a mathematical guarantee that the output distribution of a randomized algorithm changes by no more than a bounded factor when any single individual's record is added to or removed from the training dataset, formalized by the (ϵ, δ) bound $\Pr[\mathcal{A}(D) \in S] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in S] + \delta$ for adjacent datasets D and D' and any measurable output set S (Dwork et al. 2006; Dwork and Roth 2014).

1. **Significance**: The privacy budget ϵ converts an algorithmic property into an engineering currency that can be allocated, composed, and depleted. Production reports and benchmark studies show that privacy budgets are deployment-specific: Apple describes local-DP telemetry deployments with per-event ϵ values chosen by use case, while DP-SGD studies show accuracy and compute trade-offs across benchmark tasks (Apple Differential Privacy Team 2017; Abadi et al. 2016; Bu et al. 2020; De et al. 2022). Smaller ϵ raises the noise floor and usually degrades model accuracy (figure 13.25); DP-SGD can also reduce training throughput because per-sample gradient clipping works against the batch-level parallelism that makes accelerator training efficient.
2. **Distinction**: Unlike de-identification, which removes identifiers from records and fails against re-identification through auxiliary data, and unlike statistical disclosure control, which depends on the attacker's model, differential privacy is a property of the algorithm, not of the data: the DP guarantee holds against any adversary, including one with arbitrary side information about other records in the dataset.
3. **Common pitfall**: A frequent misconception is that ϵ is a per-query parameter rather than a budget across the entire interaction lifetime. Sequential composition of k queries each at ϵ_0 yields a total privacy loss of $k\epsilon_0$ under basic composition, tighter under advanced and Renyi accountants (section 13.8.2); a system that runs many DP-SGD training steps or answers many DP queries must therefore track cumulative loss and stop, refuse further queries, or refresh keys when the allocated budget is exhausted.

The central technical problem of differential privacy is quantifying privacy loss when learning from data. Traditional privacy approaches focus on removing identifying information (names, addresses, social security numbers) or applying statistical disclosure controls. However, these methods fail against sophisticated adversaries who can re-identify individuals through auxiliary data, statistical correlation attacks, or inference from model outputs.

Differential privacy takes a different approach by focusing on algorithmic behavior rather than data content. The key insight is that privacy protection should be measurable and should limit what can be learned about any individual, regardless of what external information an adversary possesses.

The salary-average scenario introduced in section 13.1.2 already computed the noise this mechanism adds; the formal definition gives that calculation its guarantee. To rebuild the intuition: an analyst computes the average salary of a group of people, where no one wants to reveal their actual salary. With differential privacy, each person writes their salary on a piece of paper, but before handing it in, adds or subtracts a random number from a known distribution. Averaging many papers reduces the random noise in expectation, giving an accurate estimate of the true average. Pulling out any single piece of paper, however, does not reveal the exact salary because the random offset is unknown. This is the core idea: learn aggregate patterns while making it difficult to be sure about any single individual.

Differential privacy formalizes this intuition through a comparison of algorithm behavior on similar datasets, as figure 13.24 depicts. Consider two adjacent datasets that differ only in the presence or absence of a single individual's record. Differential privacy ensures that the probability distributions of algorithm outputs remain statistically similar regardless of whether that individual's data is included. This protection is achieved through carefully calibrated noise that masks individual contributions while preserving the aggregate statistical patterns necessary for machine learning.

To make this intuition mathematically precise, differential privacy introduces a quantitative measure of privacy loss. The mathematical framework uses probability ratios to bound how much an algorithm's behavior can change when a single individual's data is added or removed. This approach proves privacy guarantees rather than simply assuming them.

A randomized algorithm $\mathcal{A}$ is said to be ϵ-differentially private if, for all adjacent datasets D and D' differing in one record, and for all outputs $S \subseteq \text{Range}(\mathcal{A})$, the following holds (Dwork et al. 2006; Dwork and Roth 2014):

$$\Pr[\mathcal{A}(D) \in S] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in S]$$

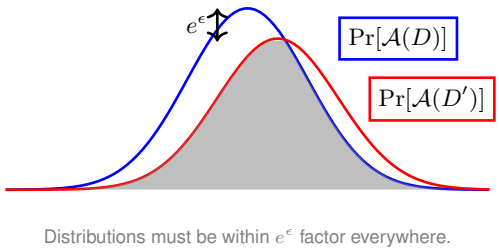


Figure 13.24: Differential Privacy Indistinguishability: Differential privacy ensures that the probability distribution of an algorithm's output on dataset D is nearly identical to its output on an adjacent dataset D'. This statistical indistinguishability (controlled by the privacy budget ϵ) prevents an observer from inferring whether any single individual's data was included in the training set.

The parameter ϵ quantifies the privacy budget, representing the maximum allowable privacy loss. Smaller values of ϵ provide stronger privacy guarantees through increased noise injection, but may reduce model utility. Typical values include $\epsilon = 0.1$ for strong privacy protection, $\epsilon = 1.0$ for moderate protection, and $\epsilon = 10$ for weaker but utility-preserving guarantees. The multiplicative factor e^ϵ bounds the likelihood ratio between algorithm outputs on adjacent datasets, constraining how much an individual's participation can influence any particular result.

Figure 13.25 quantifies this cost using published empirical results across two benchmark datasets. The pragmatic sweet spot lies between $\epsilon \approx 1$ and $\epsilon \approx 10$, where meaningful privacy guarantees coexist with acceptable accuracy loss. MNIST remains comparatively accurate under single-digit privacy budgets, while CIFAR-10 trained from scratch without extra data remains much harder: De et al. report 81.4 percent accuracy at $(\epsilon, \delta) = (8, 10^{-5})$, even as pretraining and extra public data can substantially improve private image-classification accuracy (De et al. 2022).

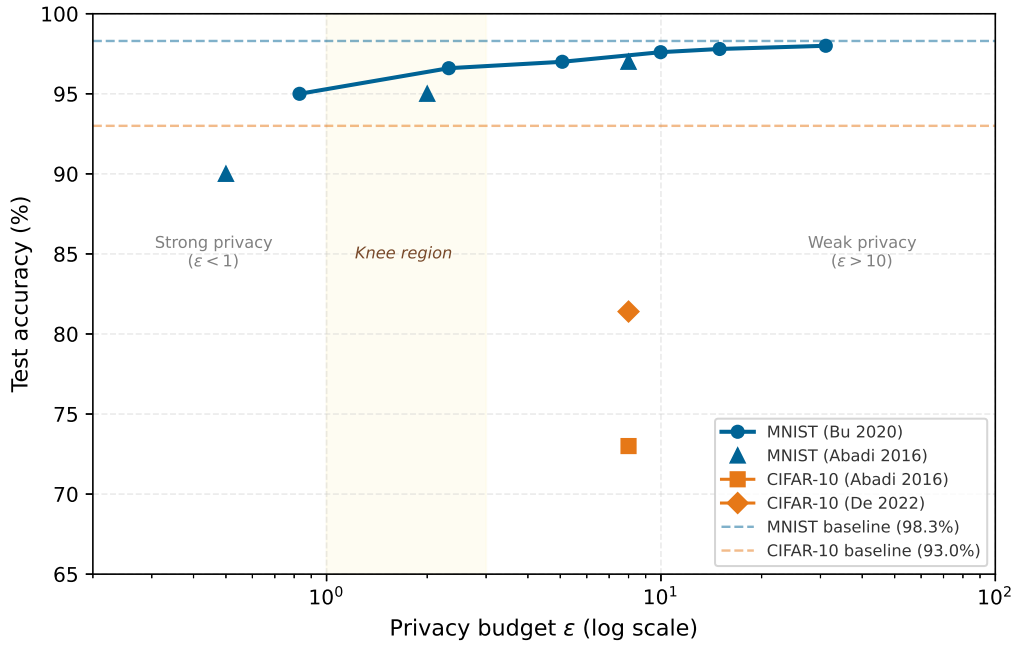


Figure 13.25: The Privacy-Utility Frontier: Published accuracy at various privacy budgets from (Abadi et al. 2016; Bu et al. 2020; De et al. 2022). MNIST stays comparatively accurate under single-digit epsilon budgets, while CIFAR-10 trained from scratch without extra data remains harder, with De et al. reporting 81.4 percent at epsilon = 8. The knee region, here shown between an epsilon of 1 and 3, marks the transition from practical privacy to severe utility loss in this illustrative curve.

This bound ensures that the algorithm's behavior remains statistically indistinguishable regardless of whether any individual's data is present, thereby limiting the information that can be inferred about that individual. In practice, DP is implemented by adding calibrated noise to model updates or query responses, using mechanisms such as the Laplace or Gaussian mechanism. Training techniques like differentially private stochastic gradient descent[66] integrate calibrated noise into training computations, ensuring that individual data points cannot be distinguished from the model's learned behavior.

[66] **DP-SGD (Differentially Private SGD)**: Introduced by Abadi et al. (2016), DP-SGD clips per-sample gradients and adds calibrated Gaussian noise during training. Apple reported a large-scale local differential privacy deployment in 2017 for telemetry-style use cases; that deployment is evidence of production DP, not DP-SGD specifically (Apple Differential Privacy Team 2017). The systems cost of DP-SGD can be significant: per-sample gradient clipping prevents the batch-level parallelism that makes accelerator training efficient, reducing throughput relative to standard SGD.

13.8.1 Mathematical foundations and privacy parameters

Production deployments fail when a privacy budget is treated as a one-time label rather than an accounting system. The mathematical foundations specify which guarantee is being claimed, how noise is calibrated, and how privacy loss composes across repeated training steps or queries. Systems that manage sensitive datasets need this machinery before they can state that an ϵ value still holds after deployment.

13.8.1.1 The (ϵ, δ)-differential privacy formulation

The pure ϵ-DP definition provides strong guarantees but can be overly restrictive for practical ML applications. A relaxed formulation, (ϵ, δ)-differential privacy, allows a small probability δ of privacy loss exceeding ϵ. This relaxation proves essential for deep learning, where Gaussian noise (which has unbounded support) is preferred over Laplace noise for gradient perturbation.

Formally, a randomized algorithm $\mathcal{A}$ satisfies (ϵ, δ)-differential privacy if for all adjacent datasets D, D' and all measurable output sets S:

$$\Pr[\mathcal{A}(D) \in S] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in S] + \delta$$

The parameter δ represents the probability that the privacy guarantee fails catastrophically. In practice, δ is commonly chosen smaller than the inverse dataset size, often $\delta < 1/n_{\text{records}}$ or smaller depending on policy and threat model; smaller δ strengthens the failure-probability bound at additional utility cost. For a dataset with 1 million records, $\delta = 10^{-12}$ ensures the probability of exceeding the ϵ bound remains vanishingly small.

13.8.1.2 Noise mechanisms for achieving differential privacy

ML systems operationalize differential privacy through carefully calibrated noise injection. The two primary mechanisms used in ML systems differ in their noise distributions and applicability. The first is the Laplace mechanism. For pure ϵ-DP, it adds noise drawn from $\text{Lap}(\Delta f/\epsilon)$, where Δf is the global sensitivity of the function (the maximum change in output when one record changes). For a query f, the privatized output is:

$$\tilde{f}(D) = f(D) + \text{Lap}\left(\frac{\Delta f}{\epsilon}\right)$$

The Laplace distribution has density $p(x) = \frac{\epsilon}{2\Delta f} \exp\left(-\frac{\epsilon|x|}{\Delta f}\right)$, with scale parameter $b = \Delta f/\epsilon$. The noise magnitude scales inversely with ϵ: stronger privacy $(\epsilon = 0.1)$ requires $10\times$ more noise than moderate privacy $(\epsilon = 1.0)$.

The Laplace derivation shows why the noise scale must match sensitivity. To prove that the Laplace mechanism achieves ϵ-DP, consider the ratio of output probabilities on adjacent datasets D and D'. Let $f(D) = v$ and $f(D') = v'$, where $|v - v'| \leq \Delta f$ by the sensitivity bound. For any output y, the probability ratio is:

$$\frac{p(y|D)}{p(y|D')} = \frac{\exp(-\epsilon|y - v|/\Delta f)}{\exp(-\epsilon|y - v'|/\Delta f)} = \exp\left(\frac{\epsilon(|y - v'| - |y - v|)}{\Delta f}\right)$$

By the triangle inequality, $|y - v'| - |y - v| \leq |v - v'| \leq \Delta f$. Therefore:

$$\frac{p(y|D)}{p(y|D')} \leq \exp\left(\frac{\epsilon \cdot \Delta f}{\Delta f}\right) = e^{\epsilon}$$

This establishes that the Laplace mechanism satisfies ϵ-differential privacy. The derivation reveals why sensitivity calibration is essential: the noise scale must match the maximum possible change in the query output to mask individual contributions.

The Gaussian mechanism serves the ML case where the relaxed (ϵ, δ) guarantee is acceptable. It adds noise with standard deviation σ calibrated based on ϵ, δ, and the ℓ_2 sensitivity $\Delta_2 f$:

$$\tilde{f}(D) = f(D) + \mathcal{N}\left(0, \sigma^2 I\right)$$

Unlike the Laplace mechanism, which achieves pure ϵ-DP, the Gaussian mechanism generally requires the relaxed (ϵ, δ)-DP formulation because shifted Gaussian densities can have arbitrarily large likelihood ratios in the tails; the δ term bounds the probability of those tail events. The derivation proceeds by analyzing the privacy loss random variable.

For adjacent datasets with ℓ_2 sensitivity $\Delta_2 f$, the privacy loss at output y follows:

$$L_{\text{priv}}(y) = \ln \frac{p(y|D)}{p(y|D')} = \frac{\|y - f(D')\|_2^2 - \|y - f(D)\|_2^2}{2\sigma^2}$$

When $y = f(D) + z$ for noise $z \sim \mathcal{N}(0, \sigma^2 I)$, the privacy loss becomes a shifted Gaussian with mean $\frac{\Delta_2 f^2}{2\sigma^2}$ and variance $\frac{\Delta_2 f^2}{\sigma^2}$. Using tail bounds on this distribution, the probability that $L_{\text{priv}}(y) > \epsilon$ can be bounded by δ when:

$$\sigma \geq \frac{\Delta_2 f}{\epsilon} \sqrt{2 \ln \left(\frac{1.25}{\delta} \right)}$$

This formula reveals the three-way trade-off: achieving smaller ϵ (stronger privacy) or smaller δ (higher confidence) requires proportionally larger noise σ, which degrades model utility. The factor $\sqrt{2 \ln(1.25/\delta)}$ grows slowly with $1/\delta$, so cryptographically small δ (for example, 10^{-8}) only moderately increases the required noise compared to $\delta = 10^{-5}$. The resulting noise scale makes the accuracy cost concrete.

Example 13.2: The privacy-accuracy tax

Trade-off: Stronger privacy requires adding more noise to gradients during training. This noise acts like a "tax" on model accuracy.

Formula: For (ϵ, δ)-DP with gradient clipping C, the required noise standard deviation σ is:

$$\sigma \geq \frac{C\sqrt{2 \ln(1.25/\delta)}}{\epsilon}$$

Scenario:

- **Gradient Norm Limit** (C): 1
- **Failure Probability** (δ): 10^{-5}

Result (σ):

- **Strong Privacy** $(\epsilon = 1)$: $\sigma \approx 1 \times \sqrt{2 \times 11.7}/1 \approx 4.8$
- **Weak Privacy** $(\epsilon = 10)$: $\sigma \approx 1 \times \sqrt{2 \times 11.7}/10 \approx 0.48$

Systems insight: Achieving $\epsilon = 1$ requires adding noise nearly 4.8× larger than the signal (gradient norm 1). This degrades accuracy by 5–10 percent unless the model trains for significantly longer or uses much larger batch sizes to average out the noise.

The noise scale must satisfy $\sigma \geq \frac{\Delta_2 f}{\epsilon} \sqrt{2 \ln(1.25/\delta)}$ to achieve (ϵ, δ)-DP. For typical ML hyperparameters $(\epsilon = 1, \delta = 10^{-7})$, this requires $\sigma \approx 5.72 \cdot \Delta_2 f$.

Gaussian noise is preferred in deep learning because gradient norms are naturally bounded in ℓ_2 space, making sensitivity analysis tractable. The Gaussian mechanism also composes more tightly under Rényi Differential Privacy accounting.

The next subsections explain the privacy-accounting ladder. Simple composition adds privacy loss pessimistically across accesses to the data. Privacy loss random variables and moments accounting track the distribution of loss more tightly. Rényi Differential Privacy then gives a convenient way to compose many noisy SGD steps and convert the result back to an (ϵ, δ) budget.

13.8.1.3 Privacy loss random variable and moments accountant

A more refined approach to privacy analysis tracks the privacy loss random variable (PLRV), which quantifies how much information about an individual leaks from a single observation. For adjacent datasets D, D' and algorithm output o, the privacy loss is:

$$L^{(o)}_{\text{priv},(D,D')} = \ln \frac{\Pr[\mathcal{A}(D) = o]}{\Pr[\mathcal{A}(D') = o]}$$

The PLRV characterizes the log-likelihood ratio between outputs on adjacent datasets. For (ϵ, δ)-DP, the tail probability must satisfy $\Pr[L_{\text{priv}} > \epsilon] \leq \delta$. This formulation enables composition analysis through moment-generating functions.

The moments accountant technique, introduced for DP-SGD by Abadi et al. (2016), tracks higher-order moments of the privacy loss distribution. Rather than computing worst-case composition, it analyzes the actual privacy loss distribution across training iterations. For mechanism $\mathcal{M}$ with privacy loss L_{priv}, the moments accountant computes:

$$\alpha_{\mathcal{M}}(\lambda_{\text{mom}}) = \max_{D,D'} \ln \mathbb{E}_{o \sim \mathcal{M}(D)} \left[\left(\frac{\Pr[\mathcal{M}(D) = o]}{\Pr[\mathcal{M}(D') = o]} \right)^{\lambda_{\text{mom}}} \right]$$

for moment order λ_{mom}. After k compositions, the accumulated moment is $k \cdot \alpha_{\mathcal{M}}(\lambda_{\text{mom}})$. Applying the Markov inequality then yields (ϵ, δ) bounds:

$$\epsilon(\delta) = \min_{\lambda_{\text{mom}}} \left[\frac{k \cdot \alpha_{\mathcal{M}}(\lambda_{\text{mom}}) + \ln(1/\delta)}{\lambda_{\text{mom}} - 1} \right]$$

For DP-SGD training a ResNet-20 on CIFAR-10 with 100 epochs, batch size 256, and clipping norm $C = 1$, the moments accountant yields $\epsilon \approx 2.3$ (for $\delta = 10^{-5}$), compared to $\epsilon \approx 23$ under naive composition. This tighter accounting makes private deep learning practically feasible.

13.8.1.4 Renyi differential privacy and composition

Rényi Differential Privacy (RDP), introduced by Mironov (2017), provides even tighter composition bounds by tracking Rényi divergence rather than KL divergence. A mechanism $\mathcal{M}$ satisfies (α, ε)-RDP if for all adjacent D, D':

$$\mathcal{D}_\alpha(\mathcal{M}(D) \| \mathcal{M}(D')) = \frac{1}{\alpha - 1} \ln \mathbb{E}_{x \sim \mathcal{M}(D')} \left[\left(\frac{\Pr[\mathcal{M}(D) = x]}{\Pr[\mathcal{M}(D') = x]} \right)^{\alpha} \right] \leq \varepsilon$$

where $\alpha > 1$ is the Rényi order. RDP composition is remarkably simple: if mechanism $\mathcal{M}_i$ satisfies (α, ε_i)-RDP, then their composition satisfies $(\alpha, \sum_i \varepsilon_i)$-RDP. This linearity contrasts with the suboptimal $\sqrt{k}$ scaling of advanced composition for (ϵ, δ)-DP.

For Gaussian noise with scale σ, the RDP guarantee is:

$$\varepsilon(\alpha) = \frac{\alpha}{2\sigma^2}$$

After k iterations of DP-SGD with noise σ and sampling rate q, the RDP guarantee is approximately:

$$\varepsilon_{\text{total}}(\alpha) \approx \frac{k \cdot q^2 \cdot \alpha}{2\sigma^2}$$

This RDP guarantee can be converted to (ϵ, δ)-DP using:

$$\epsilon = \varepsilon + \frac{\ln(1/\delta)}{\alpha - 1}$$

optimized over α. For typical ML workloads, RDP provides 2–3$\times$ tighter bounds than moments accountant, enabling longer training with fixed privacy budget.

13.8.1.5 Practical privacy budget example: DP-SGD for image classification

Consider training a CNN on 50,000 CIFAR-10 images with $(\epsilon, \delta) = (3, 10^{-5})$ target privacy. Using DP-SGD with:

- Batch size $B = 4{,}000$ (sampling rate $q = B/n_{\text{records}} = 0.08$)
- Gradient clipping norm $C = 1$ (sensitivity $\Delta_2 = 2C/B = 0.0005$)
- Noise multiplier $\sigma = 1.3$ (noise scale relative to clipping)
- Training for $N_{\text{epochs}} = 60$ epochs ($k = N_{\text{epochs}} \cdot n_{\text{records}}/B = 750$ steps)

The RDP analysis proceeds in three steps:

1. Per-step RDP: $\varepsilon_{\text{step}}(\alpha) \approx \frac{q^2\alpha}{2\sigma^2} = \frac{(0.08)^2\alpha}{2(1.3)^2} \approx 0.00189\alpha$
2. Total RDP after 750 steps: $\varepsilon_{\text{total}}(\alpha) = 750 \times 0.00189\alpha = 1.42\alpha$
3. Converting to (ϵ, δ)-DP means optimizing over α. For $\alpha = 10$, $\epsilon = 1.42 \times 10 + \ln(10^5)/9 = 14.2 + 1.28 = 15.48$, which is too high. For $\alpha = 4$, $\epsilon = 1.42 \times 4 + \ln(10^5)/3 = 5.68 + 3.84 = 9.52$. For $\alpha = 3$, $\epsilon = 1.42 \times 3 + \ln(10^5)/2 = 4.26 + 5.76 = 10.02$. The optimum is $\alpha \approx 3.8$, yielding $\epsilon \approx 9.5$.

The single-knob changes are partial remediations, not complete solutions under this accountant. Training for $N_{\text{epochs}} = 25$ lowers the estimate to $\epsilon \approx 5.8$, while reducing the sampling rate to $B = 2{,}000$ ($q = 0.04$) lowers it to $\epsilon \approx 6.4$. Reaching the target $\epsilon = 3$ therefore requires a stronger combination, typically increasing σ substantially while accepting lower accuracy, fewer training passes, or smaller batches. These adjustments demonstrate the fundamental privacy-utility-computational resource trade-off[67] in production ML systems: differential privacy offers strong theoretical assurances, but tightening the privacy bound consumes accuracy and compute budget in measurable ways.

Practical DP deployment requires careful consideration of computational trade-offs, privacy budget management, and implementation challenges. Table 13.11 compares privacy-preserving technique trade-offs across five approaches, including federated learning. The maturity labels are a deployment snapshot for the regimes discussed here, not a permanent ranking of the underlying techniques.

[67] **Privacy-Utility Tension**: Formalized by Dwork and McSherry, who proved that perfect privacy (infinite noise) yields zero utility, while perfect utility (zero noise) provides zero privacy. The "privacy budget" (ϵ) is a finite resource: each query or training epoch consumes a portion, and once exhausted, no further computation on that data is permitted without degrading the guarantee. This makes privacy accounting a first-class constraint in ML system design, alongside compute and memory budgets.

Table 13.11: Privacy-Preserving Technique Comparison: Data privacy techniques impose varying computational costs and offer different levels of formal privacy guarantees, requiring practitioners to balance privacy strength with model utility and deployment constraints. The table summarizes key properties (privacy guarantees, computational overhead, maturity, typical use cases, and trade-offs) to guide informed decisions when designing privacy-aware machine learning systems.

Technique	Privacy Guarantee	Computational Overhead	Deployment Snapshot	Typical Use Case	Trade-offs
Differential Privacy	Formal (ϵ-DP)	Moderate to High	Production	Training with sensitive or regulated data	Reduced accuracy; careful tuning of ϵ/noise required to balance utility and protection
Federated Learning	Structural	Moderate	Production	Cross-device or cross-org collaborative learning	Gradient leakage risk; requires secure aggregation and orchestration infrastructure
Homomorphic Encryption	Strong (Encrypted)	High	Experimental	Inference in untrusted cloud environments	High latency and memory usage; suitable for limited-scope inference on fixed-function models
Secure MPC	Strong (Distributed)	Very High	Experimental	Joint training across mutually untrusted parties	Expensive communication; challenging to scale to many participants or deep models
Synthetic Data	Weak (if standalone)	Low to Moderate	Emerging	Data sharing, benchmarking without direct access to raw data	May leak sensitive patterns if training process is not differentially private or audited for fidelity

Increasing the noise to reduce ϵ may degrade model accuracy, especially in low-data regimes or fine-grained classification tasks. Consequently, DP is often applied selectively (either during training on sensitive datasets or at inference when returning aggregate statistics) to balance privacy with performance goals (Dwork and Roth 2014).

13.8.2 Privacy budget composition

A critical aspect of differential privacy is that privacy loss accumulates. Every time a mechanism accesses the sensitive data, it consumes a portion of the privacy budget ϵ. If an organization trains 10 models on the same dataset, each with $\epsilon = 1$, the total privacy loss is not $\epsilon = 1$ but closer to $\epsilon = 10$ (under simple composition).

The privacy ledger depends on three composition tools:

- **Simple Composition**: Running k mechanisms with ϵ_i guarantees $\sum \epsilon_i$ privacy. This is a loose bound.
- **Advanced Composition**: Provides tighter bounds, showing that privacy loss grows roughly with $\sqrt{k}$.
- **Rényi Differential Privacy (RDP)**: A framework used in deep learning (for example, DP-SGD) that offers even tighter composition tracking, essential for training neural networks over thousands of iterations.



Each query spends part of the finite ϵ privacy budget.

The practical implication is that organizations must manage a **global privacy budget** for each dataset, halting access once the budget is exhausted.

13.8.3 Quantifying the privacy-utility trade-off

The theoretical framework of differential privacy translates into measurable accuracy degradation in practice. Empirical studies show that the trade-off depends strongly on task complexity, architecture, privacy accountant, pretraining, and whether extra public data are allowed. Table 13.12 therefore lists source-backed comparison points rather than treating a single set of benchmark rows as universal.

Table 13.12: Privacy-Accuracy Trade-Offs: Published DP-SGD accuracy points are setting-specific. Older DP-SGD experiments showed strong privacy-utility costs on MNIST and CIFAR-10 (Abadi et al. 2016), while later image-classification work showed that careful tuning, larger models, and pretraining or extra public data can substantially improve private accuracy (De et al. 2022).

Source/setting	Dataset	Training regime	Privacy budget	Reported private accuracy
Abadi 2016	MNIST	DP-SGD neural-network experiment	$(8, 10^{-5})$-DP	97%
Abadi 2016	CIFAR-10	DP-SGD CIFAR-10 experiment	$(8, 10^{-5})$-DP	73%
De 2022	CIFAR-10	Wide-ResNet, no extra data	$(8, 10^{-5})$-DP	81.4%
De 2022	ImageNet	NFNet-F3 with JFT-4B pretraining	$(8, 8 \cdot 10^{-7})$-DP	86.7%

Several patterns emerge from these empirical results. First, simpler tasks tolerate DP better: MNIST remains substantially easier than CIFAR-10 in the original DP-SGD experiments. Second, dataset size and training regime matter critically: larger datasets and public pretraining can raise the signal-to-noise ratio and change the apparent privacy-utility frontier. Third, the privacy-accuracy curve is nonlinear: tightening ϵ usually costs more utility as the privacy budget becomes smaller, so every deployment needs its own task-specific sweep rather than relying on a generic table.

13.8.4 Decision framework for differential privacy deployment trade-offs

The quantified trade-offs turn differential privacy from a mathematical guarantee into a deployment decision. The criteria below synthesize regulatory requirements, threat models, and operational constraints into one question: whether a formal privacy budget is worth the accuracy, latency, and compute it consumes.

Start with the regulatory and legal requirement. DP becomes especially valuable when policy, contracts, audits, or risk assessments require demonstrable privacy bounds. Regulations rarely mandate DP by name, but GDPR's data-protection-by-design principle (European Parliament and Council of the European Union 2016) and privacy-by-design framing (Cavoukian 2012) motivate privacy-preserving system design, while HIPAA's technical safeguards require controls such as access control, audit controls, integrity protections, authentication, and transmission security (U.S.

Department of Health and Human Services 2005). DP can complement those controls by documenting a mathematical privacy-loss bound, but it does not replace HIPAA safeguard implementation. Organizations handling EU health data, US medical records, or California personal data subject to CCPA/CPRA should evaluate DP as one possible compliance-supporting mechanism.

Next, match the threat model to the mechanism. DP protects against membership inference (determining if a specific individual's data was used in training) and training data extraction attacks by bounding the privacy loss associated with any one record. When the threat model includes sophisticated adversaries attempting these attacks (competitors, nation-state actors, or malicious insiders), DP provides a quantifiable guarantee that other techniques lack. When the primary threats are data breaches of stored data rather than inference attacks on deployed models, encryption and access controls may be more appropriate defenses.

The dataset then determines whether the guarantee is usable. Effective DP training depends on whether the dataset can absorb added gradient noise without destroying the task signal:

- **Size threshold**: Datasets with fewer than 50,000 samples rarely achieve acceptable utility with meaningful privacy ($\epsilon < 10$). For small datasets, consider federated learning with secure aggregation as an alternative.
- **Task complexity**: Simple classification tasks (binary or few-class) tolerate noise better than fine-grained recognition or generation tasks.
- **Data sensitivity distribution**: If sensitive attributes are concentrated in rare subgroups, DP may disproportionately degrade performance on those subgroups, raising fairness concerns.

These constraints decide whether DP is a deployable guarantee or only a formal property that destroys task utility.



Small sensitive datasets often cannot absorb DP noise without losing utility.

The utility budget must also be explicit. The maximum acceptable accuracy degradation must be quantified before deployment. For safety-critical applications (medical diagnosis, autonomous vehicles), even 5 percent accuracy loss may be unacceptable. For recommendation systems or content personalization, 10–15 percent degradation might be tolerable given the privacy benefits. Table 13.12 provides a starting point, followed by experiments on the specific task.

Finally, the computational budget decides whether the design can ship. DP training typically requires 2–5$\times$ more compute than nonprivate training due to per-sample gradient computation and larger batch sizes needed to overcome noise. If computational resources are constrained, this overhead may be prohibitive. Cloud deployments can scale compute, but edge training scenarios may find DP impractical.

13.8.4.1 Decision matrix summary

Use DP when: (1) a formal privacy bound is required by policy, contract, audit, or risk assessment, (2) membership inference is a credible threat, (3) dataset exceeds 50,000 samples, (4) task can tolerate 5–15 percent accuracy loss, and (5) computational budget supports increased training costs. In contrast, consider alternatives when: (1) primary threats are data breaches (use encryption), (2) dataset is small (use federated learning), (3) task requires maximum accuracy (use access controls and audit logging), or (4) deployment is resource-constrained (use differential privacy at inference only for aggregate queries).

Checkpoint 13.5: Applying the DP decision framework

A wearable cardiac monitor classifies arrhythmias from raw ECG signals collected on-device for 5,000 enrolled patients. The product is sold in the United States and is HIPAA-regulated; the training pipeline runs on a cloud cluster of 8 GPUs, and a competing vendor has incentive to extract the model. Apply each of the five decision criteria to determine whether DP-SGD belongs in the training pipeline.

- ☐ **Criterion 1 (regulatory)**: Decide whether HIPAA supports a DP guarantee here and identify the documentation DP supplies that ad-hoc de-identification cannot.
- ☐ **Criterion 2 (threat model)**: Identify the adversary populations that matter for a cardiac-monitor model and evaluate membership inference as a credible attack.

- ☐ **Criterion 3 (dataset characteristics)**: Compare the 5,000-patient dataset with the 50,000-sample size threshold and select the appropriate alternative when DP utility collapses on a small dataset.
- ☐ **Criterion 4 (acceptable utility loss)**: Use the safety-critical misclassification risk and 5 percent accuracy-loss ceiling to constrain the privacy budget this deployment can afford.
- ☐ **Criterion 5 (compute)**: Weigh the 8-GPU cloud training run and 2–5× DP-SGD overhead against the criteria that pull toward and away from DP, then identify the dominant deployment signal.

This framework treats DP as one tool in a privacy-preserving toolkit rather than a universal solution. The most effective deployments often combine DP with complementary techniques: federated learning for data minimization, secure aggregation for gradient protection, and access controls for deployment security. A concrete DP-SGD calculation shows how the decision criteria translate into an operational privacy budget.

Napkin Math 13.4: DP-SGD privacy budget example

Scenario: Train a sentiment classifier on 100,000 customer reviews with target privacy $\epsilon \leq 8$ and $\delta = 10^{-6}$.

Step 1: Configure DP-SGD parameters.

- Dataset size: $n_{\text{records}} = 100{,}000$
- Batch size: $B = 2{,}000$ (sampling rate $q = B/n_{\text{records}} = 0.02$)
- Gradient clipping norm: $C = 1$
- Training epochs: $N_{\text{epochs}} = 10$ (total steps $k = N_{\text{epochs}} \times n_{\text{records}}/B = 500$)
- Target: $\epsilon = 8$, $\delta = 10^{-6}$

Step 2: Calculate required noise multiplier. Using the Gaussian mechanism formula and RDP accounting, we need noise multiplier σ such that after $k = 500$ iterations with sampling rate $q = 0.02$, the total privacy loss is $\epsilon \leq 8$.

The per-step RDP guarantee for Gaussian mechanism with subsampling is approximately:

$$\varepsilon_{\text{step}}(\alpha) \approx \frac{q^2 \alpha}{2\sigma^2}$$

For $\sigma = 0.8$ (a typical starting point):

$\varepsilon_{\text{step}}(\alpha) = \frac{(0.02)^2 \alpha}{2(0.8)^2} = \frac{0.0004\alpha}{1.28} \approx 0.000312\alpha$

After 500 steps: $\varepsilon_{\text{total}}(\alpha) = 500 \times 0.000312\alpha = 0.156\alpha$

Step 3: Convert RDP to DP. Converting to (ϵ, δ)-DP by optimizing over α:

$\epsilon = \varepsilon_{\text{total}}(\alpha) + \frac{\ln(1/\delta)}{\alpha - 1} = 0.156\alpha + \frac{\ln(10^6)}{\alpha - 1}$

Sweeping the bound across α (table 13.13) shows the slope and tail terms trading off, with a minimum near $\alpha \approx 10.4$. Optimizing the displayed approximation gives $\epsilon \approx 3.1$, which satisfies the target of $\epsilon \leq 8$ with margin.

Step 4: Expected accuracy impact. With $\sigma = 0.8$ and $\epsilon \approx 3.1$ under this simplified accountant, expect a nontrivial accuracy degradation compared to nonprivate training. For a sentiment classifier achieving 92 percent accuracy without DP, the private version might land several points lower depending on model capacity, clipping norm, optimizer, and data distribution.

Systems insight: The implementation is accountable only if the mechanism and the accountant agree: clipping bounds sensitivity, noise sets the privacy-utility trade-off, and accumulated epsilon becomes a release gate for the trained artifact.

The α-sweep behind Step 3 makes the optimization concrete: each Rényi order yields a different slope-plus-tail bound, and the budget is the minimum across orders.

Table 13.13: RDP-to-DP conversion across Rényi orders: The privacy bound ϵ is the sum of a slope term that grows with α and a tail term that shrinks with α; the released budget is the minimum over orders, near $\alpha \approx 10.4$.

α	Slope term	Tail term	ϵ
8	1.25	1.97	3.22
10	1.56	1.54	3.10
12	1.87	1.26	3.13

The guarantee itself comes from clipping, calibrated noise, and the privacy accountant; the library call in listing 13.1 is only an implementation path for those mechanism choices.

Listing 13.1: DP-SGD training step: Per-step clipping, noise addition, and accountant update that together enforce the privacy budget.

```
clip each per-example gradient to norm C
add Gaussian noise with multiplier sigma to the averaged clipped gradient
take one optimizer step with the privatized gradient
accumulate privacy loss in the accountant after every step
stop, retune, or reject the run if accumulated epsilon exceeds the target budget
```

13.9 Security and Privacy Maturity Model

Moving from isolated mathematical proofs to a fully fortified production environment requires more than setting an epsilon value; it requires a structured, multi-phase organizational strategy for deploying security controls. A common mistake when securing a new ML platform is attempting to deploy every defense at once: differential privacy, trusted execution environments, adversarial training, model watermarking, and red-team automation. The resulting friction can paralyze the engineering team while basic controls remain weak. A better maturity model asks which system property is currently unprotected: access boundary, data privacy, model integrity, adversarial robustness, or governance evidence. The dependency structure in table 13.14 keeps that choice tied to the system boundary being defended.

Table 13.14: ML Security and Privacy Maturity Model: Security controls should be added according to the threat model and deployment context. The order is not a calendar; it is a dependency structure from basic access boundaries to stronger privacy, integrity, adversarial, and governance controls.

Maturity layer	Primary question	Control family
Access and configuration boundary	Who can touch data, weights, deployments, and logs?	Least privilege, service identity, encrypted transport, audit logging
Data and privacy boundary	What can be learned from training, fine-tuning, or output?	Privacy accounting, output limiting, secure aggregation, sensitive-data isolation
Model integrity boundary	How do we know this is the validated model and data path?	Signed artifacts, registry controls, hash checks, release gates, rollback paths
Adversarial and abuse boundary	What can an adaptive attacker extract, infer, or induce?	Query monitoring, rate limiting, robustness evaluation, red-team exercises
Governance and evidence boundary	Which obligations must be proven after deployment?	Compliance mapping, incident records, retention policy, reproducible control checks

The maturity layers are cumulative. Access controls and encrypted channels are prerequisites because a model-extraction defense does not matter if the registry is open or a serving node can load unverified weights. Privacy controls become necessary when user data enters fine-tuning, personalization, or telemetry feedback loops; the privacy budget then becomes a first-class system

resource, tracked across retraining, hyperparameter searches, and A/B tests that touch the same sensitive dataset. Model-integrity controls protect the artifact path itself: signed manifests, key management, hash checks, and release gates ensure that the model being served is the one that was validated.

Adversarial defenses enter when the attacker is adaptive rather than accidental. Query monitoring and output limiting reduce the information available for extraction attacks, while robustness evaluation and red-team exercises test whether the model can be induced into unsafe behavior. Stronger mechanisms such as trusted execution environments or certified defenses that provide formal robustness guarantees under a specified perturbation bound should be selected only when the threat model justifies their latency, hardware, and operational cost. Governance then closes the loop: audit logs, compliance mappings, incident records, and reproducible control checks provide evidence that the deployed system still satisfies its privacy and security obligations after the model, data, and traffic change.

Different domains traverse the model differently. A healthcare deployment prioritizes privacy accounting and governance evidence before broad patient-facing use. A financial fraud system emphasizes access control, model integrity, and abuse monitoring because extraction and evasion directly change loss exposure. An autonomous system emphasizes adversarial robustness and incident evidence because safety depends on behavior under distribution shift and physical-world perturbations. The point is not to follow a universal implementation calendar; it is to make the threat model determine which control boundary must mature next.

The maturity model moves security from reactive patching toward proactive threat control. Even well-designed systems can still fail when teams mistake a local mechanism for a complete guarantee.

13.10 Fallacies and Pitfalls

A common ML security failure is treating a local mechanism as a system guarantee. Obscurity, differential privacy libraries, federated learning, or encrypted storage each solve one part of the threat model, but none substitutes for lifecycle-wide reasoning about data, models, interfaces, hardware, and distributed scale.

Fallacy: *Security through obscurity provides adequate protection for machine learning models.*

Hiding architectures or parameters provides no meaningful security when black-box attacks can succeed without internal knowledge. As detailed in section 13.4.1.3, the cited extraction examples reconstruct functionality with 90 percent accuracy using 10,000–100,000 queries; the 100,000-query daily limit is a scenario assumption for showing why simple rate limits may not be enough. Adversarial examples transfer across architectures with 60–80 percent success rates in the evaluated settings, exploiting shared geometric properties rather than architectural details. Organizations relying on secrecy discover this weakness when "proprietary" models are reconstructed through patient querying. Effective ML security requires robust defenses functioning under Kerckhoffs's principle: assume attackers have complete knowledge and build protections through query limiting, output perturbation, watermarking, and anomaly detection rather than architectural secrecy.

Pitfall: *Assuming that differential privacy automatically ensures privacy without considering implementation details.*

Many practitioners treat differential privacy as a universal solution without understanding parameter selection or budget tracking. As established in section 13.8.1, privacy strength varies nonlinearly: $\epsilon = 0.1$ provides strong privacy but degrades accuracy by 10–15 percent, $\epsilon = 1.0$ offers moderate protection with 5-10 percent degradation, while $\epsilon = 10$ gives weak guarantees with minimal utility loss. Privacy budgets compound across operations: training 10 models at $\epsilon = 1.0$ each consumes total $\epsilon = 10$, not $\epsilon = 1.0$. A production system retraining monthly for two years accumulates $\epsilon = 24$ even if targeting $\epsilon = 1.0$ per run. Organizations failing to track cumulative privacy loss across retraining, hyperparameter tuning, and A/B testing exceed guarantees by orders of magnitude, violating regulations despite using DP libraries.

Fallacy: *Federated learning inherently provides privacy protection without additional safeguards.*

This misconception assumes decentralization automatically ensures privacy, but gradient updates transmitted during training leak significant information. Membership inference attacks achieve 70–90 percent accuracy on production federated models, determining whether specific data points

were used in training by exploiting behavioral differences. Gradient inversion attacks can reconstruct original training data (images, text) from gradient vectors with high fidelity. As examined in section 13.7.1.1, effective federated privacy requires layered defenses: secure aggregation protocols prevent the server from seeing individual contributions, differential privacy with $\epsilon \approx 6$ adds calibrated noise to updates, and cryptographic protection prevents gradient inversion. Organizations deploying federated learning without these safeguards discover vulnerability when researchers demonstrate gradient inversion or compliance audits reveal regulatory violations despite data never leaving devices.

Pitfall: *Planning poisoning defenses only for large training-data compromise.*

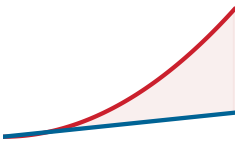
This misconception leads organizations to focus on large-scale breach prevention while underestimating surgical poisoning. Data poisoning exhibits extreme leverage: poisoning just 0.1 percent of training data (1,000 examples in 1 million) can reduce model accuracy by 10–50 percent. Backdoor attacks prove even more efficient—inserting trigger patterns into 0.01 percent of data creates models with 95 percent+ clean accuracy but 90 percent+ attack success on triggered inputs. These backdoors persist through retraining and transfer learning. Attack economics favor adversaries: creating 1,000 poisoned examples costs dramatically less than collecting millions of legitimate examples. For systems incorporating user-generated content, attackers inject poisoned data through normal channels (fake accounts, crafted ratings) representing 0.1 percent of inputs but shifting recommendations significantly. Organizations assuming "clean enough" data yields "safe enough" models discover otherwise when adversaries achieve disproportionate impact through minimal corruption.

Fallacy: *Security can be isolated from the rest of the ML system.*

Organizations often add defenses to individual components (encrypted storage, API authentication, model watermarking) without considering system-level attack vectors spanning multiple boundaries. A production system implementing strong API defenses (rate limiting (1,000 queries per day), output perturbation, top-k filtering) appears robust in isolation, but attackers bypass these through alternative batch processing endpoints, extract query-response pairs from unsecured monitoring logs, or directly download model weights from public registries without access controls. As established in section 13.3.1, effective ML security requires holistic threat modeling across the entire lifecycle: data collection, training infrastructure, model storage, deployment, and monitoring. A single weak link (unencrypted snapshots, unauthenticated endpoints, permissive CORS policies) compromises otherwise secure systems. Organizations discover this through costly incidents: API defenses bypassed, models leaked through CI/CD pipelines, or privacy-preserving training compromised by verbose logging.

Pitfall: *Underestimating the attack surface expansion in distributed ML systems.*

Organizations secure single-node ML systems effectively but fail to recognize distributed architectures multiply attack surfaces geometrically, not linearly. The shift from one machine to n_{nodes} machines increases vulnerabilities by approximately n^2_{nodes} due to inter-node communication channels creating $\mathcal{O}(n^2_{\text{nodes}})$ attack vectors in all-to-all topologies or $\mathcal{O}(n_{\text{nodes}} \log n_{\text{nodes}})$ in ring topologies.



The attack surface grows faster than the node count.

Distributed training across 128 GPUs in 16 machines means compromising one node injects poisoned gradients propagating to the global model. Centralized poisoning requiring 0.1 percent corruption achieves the same effect with 0.01 percent when targeting specific distributed nodes. Edge deployment exacerbates this: federated learning with 10,000 clients creates 10,000 compromise points—if 1 percent are compromised (100 devices), coordinated poisoning degrades accuracy by 10–50 percent while remaining below individual-device anomaly thresholds. Effective distributed ML security requires threat modeling acknowledging superlinear growth: securing inter-node channels, managing identity across security domains, and coordinating policies across heterogeneous infrastructure.

Recognizing that the shift from a single node to a distributed cluster multiplies the attack surface geometrically rules out perimeter-only thinking. Security for the machine learning fleet must be layered, measured, and tied to the full lifecycle.

13.11 Summary

Privacy and security are the defensive layers of the Machine Learning Fleet. They protect the model, data, hardware, and service boundary from adversaries who seek to steal intelligence, poison memory, or hijack decision-making logic.

The multi-layered defense architecture spans from silicon trust anchors (TEEs, HSMs) up to the linguistic safeguards required for generative AI. The fundamental shift from traditional cybersecurity to ML security makes "learned" decision boundaries the primary attack surface. Rigorous mathematical frameworks, particularly differential privacy, allow engineers to derive utility from sensitive data without compromising individual confidentiality. The breach case studies make that architecture concrete: Stuxnet turns supply-chain trust into a model-provenance problem, the Jeep Cherokee hack turns isolation boundaries into safety controls, and Mirai turns weak endpoints into fleet-scale risk.

Machine learning systems present a threat surface that traditional cybersecurity was never designed to address. A conventional application server can be hardened by patching known vulnerabilities, encrypting data at rest and in transit, and enforcing access controls. An ML system shares all of these requirements, but adds entirely new attack classes that exploit the learned decision boundary itself. An adversary who poisons training data does not need to breach a firewall; the model internalizes the corruption and carries it through every subsequent deployment. A prompt injection does not exploit a buffer overflow; it exploits the model's inability to distinguish instruction from content. These threats make security and privacy first-class engineering concerns rather than afterthoughts delegated to a separate team.

The practitioner who internalizes this chapter's layered defense architecture gains a decisive advantage: the ability to reason about where an ML system is most vulnerable at each stage of its lifecycle. From hardware root of trust through differential privacy budgets to generative AI guardrails, each layer addresses a threat that the layers below cannot catch alone. No single mechanism suffices, but their composition creates a defense posture that degrades gracefully rather than failing catastrophically. Building this discipline into the engineering workflow from the earliest design phases, rather than bolting it on after deployment, is what separates production-grade systems from prototypes that survive only until the first determined adversary arrives.

Key Takeaways: Defend the model, not just the server

- **The model is the attack surface**: ML systems inherit ordinary server threats but add attacks against the learned boundary itself. Poisoned data, model extraction, adversarial inputs, and prompt injection can compromise behavior without exploiting a traditional software vulnerability.
- **Privacy is a spent budget**: Because models can encode shadows of sensitive data, privacy cannot be treated as anonymization at the perimeter. Differential privacy offers bounded (ϵ, δ) claims, but only if budgets are tracked across training, release, and repeated query access.
- **Generative systems need semantic defenses**: LLM failures exploit language and context: prompt injection, tool misuse, and PII leakage pass through normal text channels. Output monitoring, content isolation, policy enforcement, and guardrails must sit in the serving path, not only in network controls.
- **Provenance enables recovery**: Signed datasets, locked dependencies, registry permissions, and deployment attestations let operators answer which data, code, principal, and checkpoint produced a suspect model. Without that chain, rollback and forensics become guesswork at fleet speed.
- **Trust starts below software**: TEEs, secure boot, HSMs, and hardware roots of trust protect multi-tenant and confidential workloads from layers software alone cannot isolate. The defense posture is layered because each level catches failures the others cannot see.

The deepest of this chapter's threats is not an intrusion at all. It is the model's own competence. The same fit to training data that makes a model useful is what lets an attacker read that data back

out of it, and what lets a few poisoned examples bend its behavior from the inside, because a learned decision boundary cannot be patched the way a vulnerable function can. This is the information leakage invariant (principle 17): a model trained on data always carries a shadow of that data, so privacy stops being a wall to build once and becomes a budget spent with every query answered. Security here cannot be bolted on at the perimeter; it has to be designed into the model from the first training run, because the asset and the attack surface are the same object.

What's Next: From security to resilience

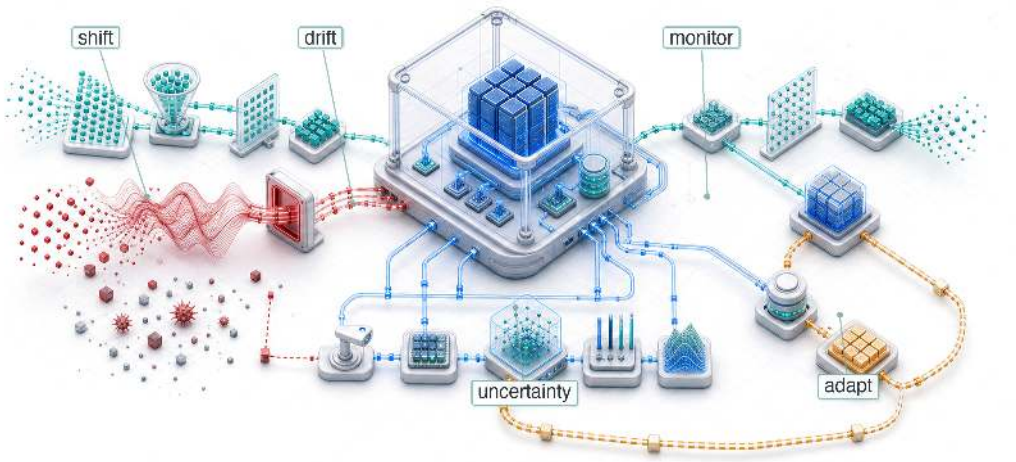
The defensive perimeter of the ML fleet protects against adversarial manipulation and unauthorized inference. Security, however, addresses only intentional threats. A model that is perfectly protected from attackers can still fail catastrophically when the real world shifts beneath it. Chapter 14 shifts the focus from *malicious* threats to *operational* stress: building systems that maintain reliability in the face of distribution drift, hardware faults, and the compounding failures that define production environments.

Research Questions: For further inquiry

- How should an ML team build a layered threat model when the model itself is both asset and attack surface?
- When does differential privacy provide enough protection to justify its accuracy, compute, and budget-composition costs?
- How can a public inference API make model extraction economically irrational without destroying legitimate utility?
- Under what threat models are TEEs, secure boot, HSMs, or PUFs worth their latency and operational overhead?

14

Robust AI

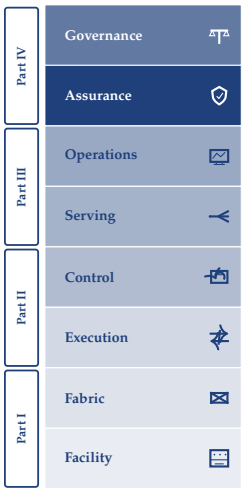


14.1 The Silent Failure Problem
14.2 Real-World Robustness Failures
14.3 A Unified Framework for Robust AI
14.4 Environmental Shifts
14.5 Input-Level Attacks and Model Robustness
14.6 Adversarial Defenses
14.7 Data Poisoning Defenses
14.8 Fallacies and Pitfalls
14.9 Summary

Purpose

Why do machine learning systems fail silently in ways that traditional software cannot?

Traditional software fails loudly: exceptions crash processes, type errors halt compilation, assertion failures stop execution. These failures are annoying but discoverable because the system signals that something is wrong. Machine learning systems fail silently. A model confronting out-of-distribution inputs continues producing outputs with full confidence, never signaling that those outputs are unreliable. A system experiencing adversarial attack serves manipulated predictions indistinguishable from legitimate ones. A model degrading under distribution drift maintains stable latency and uptime while its accuracy quietly erodes. This silence makes ML failures uniquely dangerous. By the time degradation becomes visible in business metrics, the damage has been accumulating for weeks. By the time an adversarial attack is detected, it may have influenced thousands of decisions. Robustness engineering exists to make the invisible visible: to build systems that detect when they are operating outside their competence, resist manipulation, and degrade gracefully rather than produce confidently wrong outputs. In C^3 terms, that visibility is bought with a deliberate compute penalty: continuous verification is the coordination work that catches silent, statistical decay.



Learning Objectives

- Classify robustness challenges into environmental shifts, input-level attacks, and system-level faults
- Explain how software faults amplify or masquerade as model failures using quantitative reliability metrics
- Evaluate adversarial attack techniques and select defenses such as adversarial training, certification, and input sanitization
- Construct data-poisoning defenses using anomaly detection, statistical validation, and robust training
- Apply statistical drift metrics to choose monitoring, investigation, or retraining responses
- Integrate robustness across model and system dimensions while budgeting accuracy, compute, energy, and resilience trade-offs

14.1 The Silent Failure Problem

Robust AI sits in the Governance Layer of the fleet stack. Security and privacy define the adversarial boundary: who can manipulate, extract from, or infer through the model, and which controls contain that access. Robustness asks the next systems question: when inputs, data distributions, hardware, or software no longer match the conditions assumed during training and validation, does the fleet still produce bounded, recoverable behavior? The same adversarial examples that security treats as abuse become, here, model-level perturbations to measure, defend against, and certify; nonadversarial drift and faults receive the same engineering treatment. A system that is secure but fragile is operationally useless, so robustness engineering keeps the fleet functioning under perturbation and degraded conditions.

An autonomous vehicle's vision system can operate perfectly on a sunny day in California and fail silently when a blizzard in Colorado changes the visual distribution. It will not throw an unhandled exception or print a stack trace; it may classify a snow-covered stop sign as a speed-limit sign with full confidence. Robustness is the engineering discipline that bounds this behavior under operational stress: distribution shift, sensor noise, adversarially crafted inputs, and hardware or software faults that leave the service apparently healthy while the model's answers become unreliable.

The silence is what distinguishes the discipline. A self-driving car's perception system does not crash when it misclassifies a truck as the sky; a demand forecasting model does not error out when it produces wildly inaccurate predictions; a medical diagnosis system does not shut down when it quietly provides incorrect classifications that could endanger patient lives. This **silent failure mode** makes robustness a unique and critical challenge in AI systems: engineers must defend against a world that refuses to conform to training data, not merely against bugs in code.

The silent failure challenge grows more severe as ML systems expand across diverse deployment contexts. In cloud-based services, edge devices, and embedded systems, hardware and software faults directly impact performance and reliability. The increasing complexity of these systems and their deployment in safety-critical applications[1] makes robust and fault-tolerant designs essential for maintaining system integrity.

Checkpointing and recovery keep training jobs alive, and access control enforces authentication at the system boundary. Neither addresses what happens when a deployed model receives an adversarial input indistinguishable from a legitimate one, when the data distribution drifts so far that predictions become meaningless, or when a software fault in the preprocessing pipeline silently corrupts every inference. These failure modes span the complete ML lifecycle and demand techniques for fault detection, isolation, and recovery that go beyond any single defense. The consequences of ignoring them range from economic disruption to life-threatening situations in safety-critical domains.

These failure modes motivate a precise definition of Robust AI:

[1] **Safety-Critical Applications**: Systems classified at Safety Integrity Level (SIL) 3–4 or Automotive Safety Integrity Level (ASIL) D impose very low dangerous-failure targets, but the thresholds are standard-specific. IEC 61508 high-demand SIL 3 is typically 10^{-8} to $< 10^{-7}$ dangerous failures per hour, SIL 4 is 10^{-9} to $< 10^{-8}$ per hour, and ISO 26262 ASIL D probabilistic metric for random hardware failures (PMHF) targets are commonly below 10^{-8} per hour. ML deployment in these domains faces a fundamental tension: neural networks lack the formal verifiability that regulators require, forcing multi-year certification processes that lag the model iteration cycle by orders of magnitude.

Definition 14.1: Robust AI

Robust AI is the measurable systems property that a model's predictions remain valid (within specified error bounds) under distribution shift, adversarial perturbation, and hardware or software faults, as opposed to the average-case accuracy achieved under ideal i.i.d. conditions.

1. **Significance**: Robustness is quantified by worst-case guarantees: a certified robust classifier proves that its prediction cannot change for any input within a specified perturbation set, such as an ℓ_∞ ball of radius ϵ around a test point. For image classification, $\epsilon = 8/255$ (a perturbation invisible to humans) typically drives a nonrobust model's accuracy to near zero under strong attacks such as projected gradient descent. Distribution shift compounds this: a clinical NLP model trained on 2019 records and deployed in 2021 without retraining can see accuracy drop 15–25 percent as medical coding practices and terminology evolve.
2. **Distinction**: Unlike standard generalization (which measures *average-case accuracy* on held-out i.i.d. test data drawn from the same distribution as training), robustness measures *worst-case performance* on adversarial or out-of-distribution inputs, a distinction that matters because a model can achieve 95 percent i.i.d. test accuracy while failing completely on inputs that differ from training by amounts imperceptible to humans.
3. **Common pitfall**: A frequent misconception is that robustness can be added as a post-hoc monitoring layer to any existing model. A model's robustness properties are determined primarily during training—models trained without adversarial examples or robustness objectives *cannot* achieve certified robustness through inference-time filtering alone, because the vulnerability is in the learned decision boundary, not in which inputs reach the model.

Three categories of threat produce these silent failures, and each demands distinct engineering responses. The first and most pervasive is environmental change: distribution shifts, concept drift, and evolving operational contexts challenge the core assumptions underlying model training. A model trained on last year's transaction patterns quietly becomes unreliable as customer behavior evolves, requiring continuous monitoring and adaptation strategies that go beyond standard operational practices.

The second category, malicious manipulation, targets model behavior directly. Adversarial attacks, data poisoning attempts, and prompt injection vulnerabilities cause models to misclassify inputs or produce unreliable outputs—failures that authentication and access control (Chapter 13) cannot prevent because the attacker operates within the model's own input space.

The third category is system-level fault: hardware faults, software bugs, dependency failures, and runtime errors that corrupt the machinery around the model. These faults can also amplify, mask, or mimic the other robustness failures. Bugs, design flaws, and implementation errors within algorithms, libraries, and frameworks propagate through the system, creating systemic vulnerabilities[2] that transcend individual component failures. A preprocessing bug might create artificial distribution shifts; a numerical error might corrupt model behavior in ways indistinguishable from adversarial attack; a race condition might corrupt learned representations. Because these faults originate in the systems layer, their detailed taxonomy and mitigation strategies are covered in Chapter 7; here, we focus on how system-level faults interact with environmental shifts and input-level attacks.

The appropriate defense depends on where the system runs. Large-scale cloud environments can afford redundancy and sophisticated error detection mechanisms that would overwhelm an edge device's power and memory budgets. Edge devices (Chapter 11) must instead rely on targeted hardening strategies[3] that protect the most critical inference paths while accepting weaker guarantees elsewhere.

Despite these contextual differences, a robust ML system requires fault tolerance, error resilience, and sustained performance across all deployment environments, and those guarantees are not free. Error correction adds memory-bandwidth overhead, redundant processing multiplies energy draw, and continuous monitoring claims a share of compute, and each also generates additional heat that

2 | **Systemic Vulnerability**: Architectural weaknesses that cascade across layers rather than isolating to one component. Log4Shell (CVE-2021-44228) affected hundreds of millions of devices through a single logging library. ML pipelines face analogous risk: a single CUDA or PyTorch version pinned across thousands of models means one vulnerability compromises the entire fleet simultaneously, turning dependency management into a reliability-critical function.

3 | **Hardening Strategy**: Defense-in-depth applied to ML pipelines: model loading (signature verification), input processing (adversarial filtering), and output validation (confidence thresholds). On resource-constrained edge devices, selective hardening prioritizes critical paths—protecting the inference engine while accepting weaker guarantees on logging—because full redundancy would exceed the power and memory budgets that make edge deployment viable.

exacerbates the thermal management challenges constraining deployment density. The robustness question is where this additional resource cost provides enough reliability value to justify itself.

Robustness, then, is not an afterthought to be bolted onto a finished system. It is an architectural constraint that shapes every layer of the ML pipeline, from input validation and adversarial training through drift detection and software fault isolation, and the engineering cost of ignoring it compounds silently until the system fails in production. Silent failures have caused significant damage to production systems across cloud, edge, and embedded deployments.

14.2 Real-World Robustness Failures

Across cloud, edge, and embedded environments, ML systems fail when an assumption hidden in the stack becomes a dependency the system does not monitor. The incidents below differ in scale and domain, but each shows the same pattern: the system continues to operate while an unobserved condition corrupts the result.

War Story 14.1: The label that exposed the test gap

Context: In June 2015, Google Photos launched automatic image labeling to organize user photos at consumer scale (Kasperkevic 2015).

Failure mode: Brooklyn programmer Jacky Alciné tweeted screenshots showing the system had tagged photos of him and a Black friend under the album label "Gorillas." The misclassification reflected a vision model and evaluation pipeline that had not surfaced harmful slice-level failures before launch.

Consequence: Yonatan Zunger, then Google's Chief Architect for Social, responded publicly on Twitter within hours, called the result unacceptable, and apologized. Google's interim fix removed the labels "gorilla," "chimpanzee," and "monkey" from Photos entirely—a category-level deletion rather than a model correction. Reporting years after the incident described the labels as still blocked rather than restored: the trade-off taken under incident pressure became long-lived because the team could not be confident the underlying classifier would not fail the same way on the same population again.

Systems lesson: Robustness is not aggregate accuracy. Production vision systems need slice-level evaluation, harmful-label tests, and safe fallback behavior for labels whose errors carry high social cost—and the "temporary" mitigation often outlives the fix it was meant to bridge.

That test-gap failure is the model-facing version of a broader reliability problem. In cloud infrastructure, the hidden assumption is often that a shared dependency remains available and correct.

14.2.1 Cloud infrastructure failures

Robust ML systems inherit a reliability tradition that predates machine learning. Loud, non-ML infrastructure failures, such as the 2017 AWS S3 outage[4] in which a mistyped maintenance command removed too much capacity and cascaded through every service that treated regional object storage as an availability invariant (Amazon Web Services 2017), are the kind of dependency and fault failure whose detection and recovery mechanics this chapter defers to Chapter 7. They are loud rather than silent, and they motivate the discipline this chapter assumes rather than the silent, model-level failures it focuses on. The economics of large-scale training amplify these consequences: an S3 outage starves thousands of accelerators of data shards simultaneously, and any checkpoint writes that fail during the outage window mean that when preemption eventually returns the cluster to the scheduler, hours or days of gradient updates are unrecoverable. The genuinely new and uniquely ML problem appears when the failure is silent: the system keeps serving while an unobserved corruption propagates.

In another case (Dixit et al. 2021), Facebook encountered a silent data corruption (SDC)[5] issue in its distributed querying infrastructure (figure 14.1). SDC refers to undetected errors during computation or data transfer that propagate silently through system layers. Facebook's system processed SQL-like queries across datasets and supported a compression application designed to reduce data storage footprints. Files were compressed when not in use and decompressed upon read

4 **AWS S3 Outage (2017)**: A mistyped command during routine maintenance removed too much capacity from S3's index and placement subsystems in US-East-1. While those subsystems restarted, S3 could not service requests and dependent AWS services experienced elevated errors or impaired functionality. The incident exposed a single-region dependency pattern: systems that assume regional storage availability as an invariant can fail even when their own application code and model-serving logic remain unchanged.

5 **Silent Data Corruption (SDC)**: Hardware errors that corrupt data without triggering any detection mechanism. Meta reported six to eight machines per million experiencing SDC daily—rates "orders of magnitude higher than soft-error predictions." In ML systems, SDC is uniquely dangerous because corrupted weights or activations produce plausible but incorrect outputs that pass all health checks, evading the monitoring that catches loud failures.

requests. A size check was performed before decompression to ensure the file was valid. However, an unexpected fault occasionally returned a file size of zero for valid files, leading to decompression failures and missing entries in the output database. The issue appeared sporadically, with some computations returning correct file sizes, making diagnosis particularly difficult.

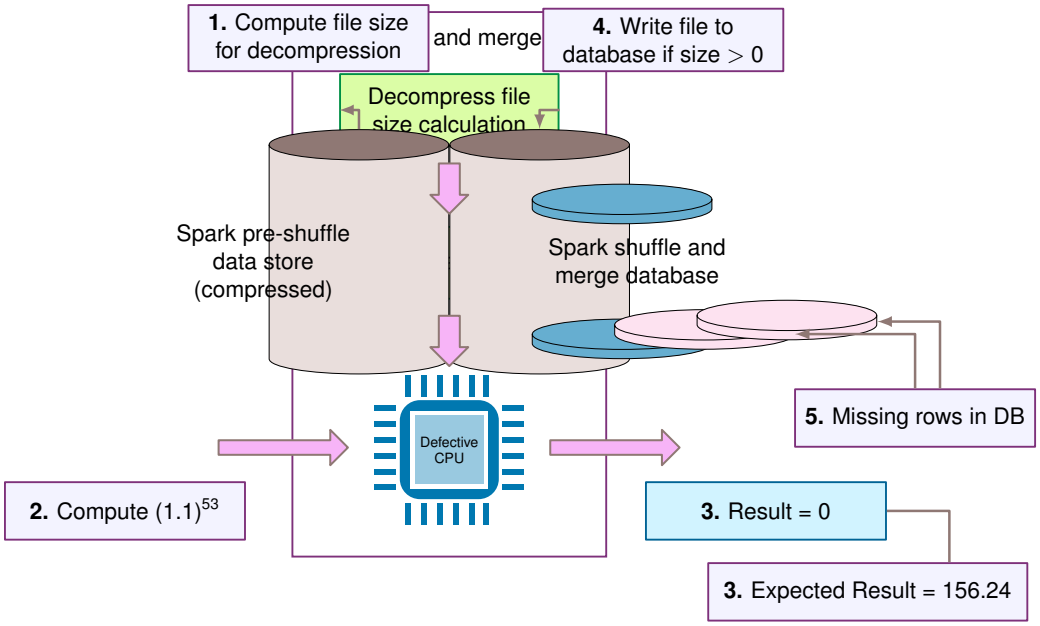


Figure 14.1: Silent Data Corruption: Unexpected faults can return incorrect file sizes, leading to data loss during decompression and propagating errors through distributed querying systems despite apparent operational success. This example from Facebook emphasizes the challenge of undetected errors, silent data corruption, and the importance of robust error detection mechanisms in large-scale data processing pipelines. Source: Facebook (Dixit et al. 2021).

In distributed ML training, silent data corruption is qualitatively more destructive than in conventional data-processing systems: a corrupted gradient or activation can perturb optimizer state and affect later steps rather than remaining bounded to one query result. A dropped row in a database query is a localized data loss bounded to that query's output—a point fix. Where the database repair is a point fix, the ML remedy is often a rollback to the last clean checkpoint and a restart of the affected training run. SDC can therefore compromise model accuracy without triggering any alert, and the blast radius grows with the synchronization and checkpointing design rather than remaining localized. Meta's production report shows SDCs as a systemic fleet issue across CPUs and software layers (Dixit et al. 2021), and recent LLM-training work shows that real-world SDCs can alter submodule outputs, optimizer steps, loss spikes, and final model weights (Ma et al. 2025). Dean's MLSys 2024 invited talk is included here as an industry-scale visual example of the same reliability concern in AI systems (Dean 2024) (figure 14.2).

14.2.2 Edge device vulnerabilities

Distributed edge deployments[6] expose the fragility of ML systems where compute, power, and connectivity are severely constrained. Self-driving vehicles serve as the canonical example of this vulnerability, as they operate in open-world environments with hard real-time latency requirements and zero tolerance for failure.

In May 2016, a fatal crash involving a Tesla Model S in Autopilot mode[7] demonstrated the catastrophic potential of perception failures (National Transportation Safety Board 2017). Traveling at 74 mph in a 65 mph zone, the vehicle's Mobileye EyeQ3 camera system failed to distinguish the white side of a tractor-trailer against a brightly lit sky. The radar, designed to ignore overhead road signs to prevent false braking events, tuned out the high-riding trailer as a stationary object. The multimodal failure resulted in a high-speed underride collision without autonomous braking intervention: both optical and radar systems received valid raw data, but the fusion logic discarded it (figure 14.3).

6 **Edge Computing**: Processing data locally rather than in centralized clouds, reducing inference latency from ~100 ms (cloud round-trip) to <10 ms. The robustness trade-off is stark: edge devices gain latency but lose the redundancy, elastic scaling, and centralized monitoring that make cloud systems resilient. A failing edge model cannot fail over to a secondary cluster—it must degrade gracefully within its own power and memory envelope or fail safely within milliseconds.

7 **Autopilot**: The 2016 crash involved Tesla's then-current SAE Level 2 driver-assistance system and Mobileye-era perception stack, not the later 8-camera full-self-driving hardware or dual FSD-chip computer. Later Tesla vehicles introduced expanded camera coverage and dedicated FSD compute, but the robustness lesson is the same: fleet-scale data collection does not automatically cover rare scenarios such as a white trailer against a bright sky.

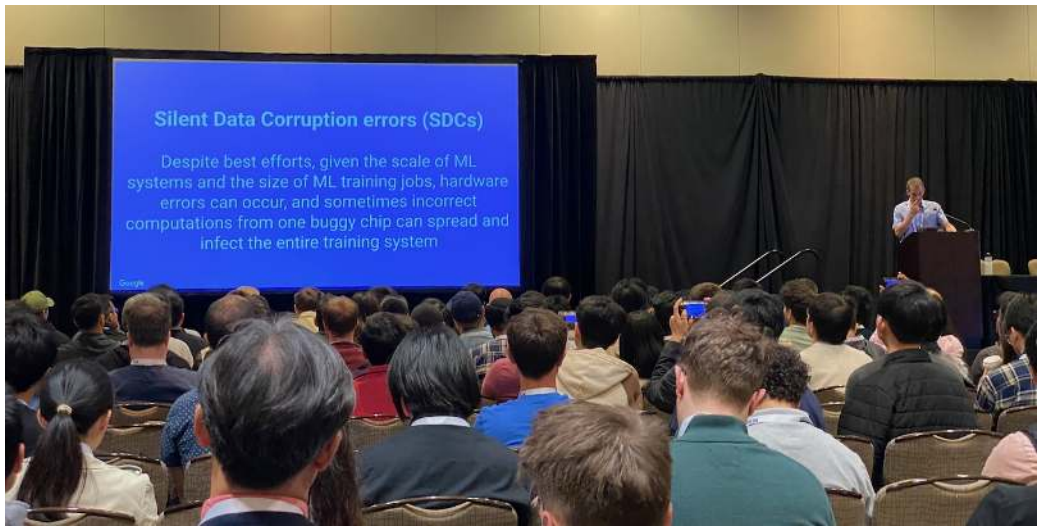


Figure 14.2: Silent Data Corruption: Jeff Dean presents on silent data corruption (SDC) at MLSys 2024, arguing that at the scale of large ML training jobs, hardware errors occur routinely and that incorrect computations from one buggy chip can propagate and infect an entire training run. Source: Jeff Dean at MLSys 2024, Keynote (Google).

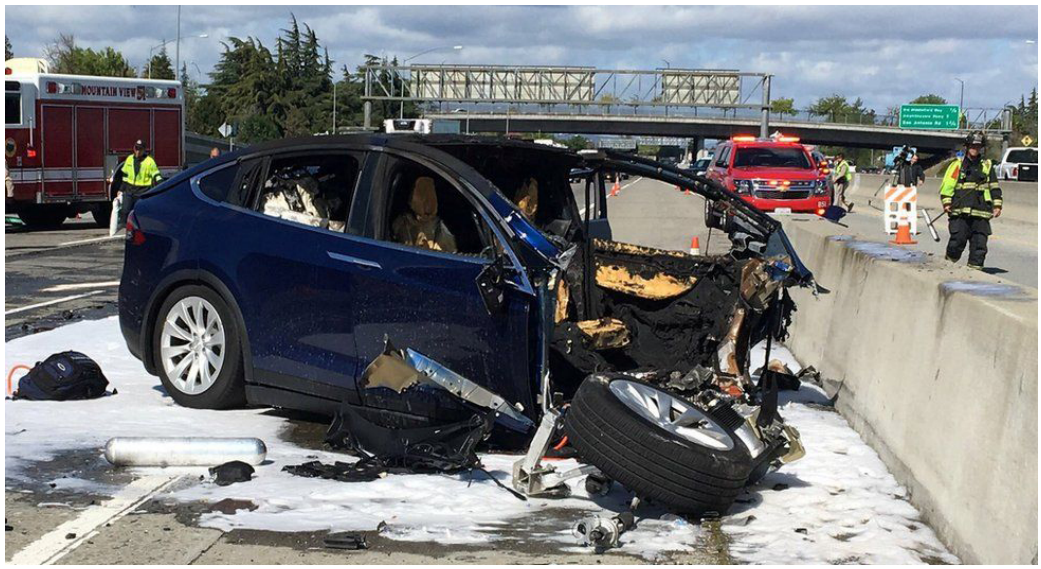


Figure 14.3: Autopilot Perception Failure: This crash reveals the critical safety risks of relying on machine learning for perception in autonomous systems, where failures to correctly classify objects can lead to catastrophic outcomes. The incident underscores the need for robust validation, redundancy, and failsafe mechanisms in self-driving vehicle designs to mitigate the impact of imperfect AI models. Source: BBC News.

A similarly tragic failure occurred in March 2018 in Tempe, Arizona, when an Uber self-driving test vehicle struck and killed a pedestrian (National Transportation Safety Board 2019). The perception system detected the victim six seconds prior to impact but fundamentally failed in **object classification stability**. As the pedestrian crossed the road, the system toggled its classification from "unknown object" to "vehicle" and then to "bicycle," resetting its trajectory prediction history with each change. Because the system lacked a persistent object track, it failed to predict a collision path until 1.3 seconds before impact—too late for the safety driver to intervene.

Beyond automotive, industrial edge deployments face similar perils. An inspection drone surveying high-voltage power lines may rely on visual odometry for stabilization; a sudden change in

lighting or a repetitive texture can cause the localization algorithm to diverge, leading to a collision or fly-away event. Edge devices lack **fallback redundancy**: no secondary cluster exists to route traffic to when the primary inference engine becomes uncertain. The system must degrade gracefully or fail safely within milliseconds. The absence of resource elasticity makes edge AI uniquely fragile to environmental variance that a data center would handle through massive over-provisioning.

14.2.3 Embedded system constraints

Embedded systems[8] operate under even tighter constraints than edge devices, often in safety-critical environments where recovery from failure is impossible. These are also the domains where ML inherits the most demanding part of the pre-ML reliability tradition: the classic embedded software faults below are loud, non-ML failures whose mechanics belong to Chapter 7, but they set the validation bar that any ML component in the decision loop must also clear.

The loss of NASA's Mars Polar Lander in 1999, attributed by the review board to premature touchdown detection that likely shut the engines off before landing (NASA Mars Program Independent Assessment Team 2000), is the canonical example: where recovery is impossible, rigorous software validation is a prerequisite, not a luxury, and the same rigor applies to any ML component in the decision loop (figure 14.4).



Figure 14.4: Robotic Lander on Mars: An artist's rendering of NASA's InSight lander deployed on the Martian surface. Remote robotic missions exemplify the class of systems in which software or sensor faults cannot be recovered by human intervention, motivating the rigorous validation and failure-mode analysis that ML components in the decision loop must also satisfy. Source: Slashgear.

Commercial aviation shows the same inherited hazard: a 2015 FAA airworthiness directive followed Boeing's discovery that a 787 powered continuously for 248 days could lose all AC power if all four generator control units entered failsafe mode at once[9], so that uptime itself became the risk factor. Safety-critical systems[10] demand stringent reliability requirements precisely because of latent hazards like this.

> "If the four main generator control units (associated with the engine-mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase."—Federal Aviation Administration directive (Federal Aviation Administration 2015)

8 | **Embedded Systems**: Dedicated processors ranging from 8-bit microcontrollers (kilobytes of RAM) to complex SoCs, with 30+ billion shipping annually. Real-time constraints (microsecond to millisecond deadlines) and unattended operation (years without maintenance) make ML deployment uniquely challenging: models cannot be easily updated, over-the-air (OTA) patches risk bricking devices, and there is no human in the loop to catch silent degradation.

9 | **Failsafe Mechanism**: A system that shifts to a safe state on fault detection, following the circuit-breaker pattern (closed/open/half-open). In ML serving, failsafes include confidence-based rejection (deferring predictions below a threshold to humans), fallback to simpler models, and automatic rollback when drift monitors fire. The trade-off is availability: aggressive confidence thresholds reject 5–15 percent of legitimate traffic, so tuning the rejection boundary becomes a reliability-vs.-throughput optimization.

10 | **ASIL (Automotive Safety Integrity Levels)**: ISO 26262 classifies automotive systems from ASIL A (lowest risk) to ASIL D (highest), where D demands 99.999 percent reliability with redundant sensors, fail-safe behaviors, and formal verification. ML-based perception systems face a certification paradox: the standard requires deterministic failure analysis, but neural networks are stochastic—their failure modes depend on input distribution, making exhaustive testing infeasible and forcing reliance on statistical safety arguments.

When AI is applied in aviation, including tasks such as autonomous flight control and predictive maintenance, the robustness of embedded systems affects passenger safety. These pre-ML failures set the validation bar that any ML component sharing those environments must also clear. A neural network running visual odometry on a planetary rover must handle cosmic-ray bit flips in its weight tensors, because hardware ECC is unavailable at those radiation levels and a corrupted layer activation can cause the localization algorithm to diverge, driving the rover into terrain it would otherwise avoid. An edge ML flight controller must implement a deterministic failsafe triggered by the model's own epistemic uncertainty: when the network's confidence falls below a specified threshold, control authority transfers to a conventional rule-based system before the neural component can make a safety-critical error. These requirements are not additions bolted onto the pre-ML validation tradition; they are the same rigor applied to a class of failure mode that traditional embedded software never encountered.

The stakes become even higher when we consider implantable medical devices. A smart pacemaker that experiences a fault or unexpected behavior due to software or hardware failure could place a patient's life at risk (BBC Future 2022). As AI systems take on perception, decision-making, and control roles in such applications, new sources of vulnerability emerge, including data-related errors, model uncertainty[11], and unpredictable behaviors in rare edge cases. The opaque nature of some AI models complicates fault diagnosis and recovery.

Each failure reveals common patterns that demand systematic approaches to robustness evaluation and mitigation: the AWS outage disrupted S3-dependent cloud services, autonomous vehicle perception errors led to fatal crashes, and spacecraft software bugs caused mission loss. The structural patterns cut across deployment environments, and a unified framework for robustness must capture how different failure modes interact and compound at system scale.

14.3 A Unified Framework for Robust AI

A flipped bit in a GPU memory module can cause a language model to generate toxic text. A gradual change in user demographics can trigger a sudden spike in recommendation latency. Production ML systems cannot treat these as isolated bugs. A unified framework must map how low-level hardware faults, software bugs, data drift, and adversarial inputs cascade upward to destroy the integrity of the model's output.

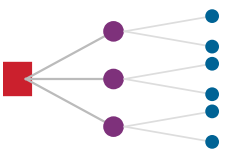
14.3.1 Connections to previous concepts

The fault tolerance mechanisms from Chapter 7, originally designed to recover training jobs from hardware crashes, serve a second role in robustness: inference-time availability. Training recovery focuses on checkpoint restoration, but robustness extends this to graceful degradation, ensuring a serving system remains operational even when inputs are adversarial or components degrade. The distributed training architectures from Chapter 5 introduce unique vulnerabilities: a single node transmitting corrupted gradients during an AllReduce operation can poison the global model weights, necessitating Byzantine fault tolerance protocols that validate peer updates before aggregation.

The security frameworks from Chapter 13 provide threat modeling principles that inform adversarial defense strategies. Operational monitoring systems from Chapter 12 provide the infrastructure foundation for detecting robustness threats in production. The serving infrastructure from Chapter 10 creates new attack surfaces: batching, model routing, and pipeline parallelism expose scheduling logic and individual pipeline stages to adversarial queries.

Large dense models amplify these risks. A GPT-3-class 175B-parameter model is too large for a single accelerator under typical FP16/BF16 serving precision, so deployments shard weights and activations across multiple devices. Each additional pipeline or tensor-parallel stage increases the fault surface compared with a monolithic deployment: a single bit flip, network partition, or adversarial input targeting one stage can bring down the entire inference request. Efficiency techniques such as INT8 quantization and aggressive pruning compound this problem by reducing the model's **robustness margin**: the amount of input perturbation, numerical error, or representation change the model can absorb before its prediction changes. Robustness engineering is therefore a constant negotiation with the efficiency and scalability constraints established in previous chapters.

11 | **Model Uncertainty (Epistemic Uncertainty)**: The reducible gap between a model's learned representation and the true data-generating process, as distinct from aleatoric uncertainty (irreducible data noise). Quantifying epistemic uncertainty enables a critical robustness mechanism: safety-critical systems can defer to human operators when predictions fall outside the training distribution. The systems cost is significant—Bayesian approximations or Monte Carlo dropout, which runs multiple dropout-perturbed forward passes at inference time, require 10–100× more inference compute, creating a direct trade-off between uncertainty awareness and serving latency.



In sharded inference, one failed stage fails the whole request.

14.3.2 From ML performance to system reliability

Once silent failure becomes a systems property, accuracy, latency, and throughput no longer describe the full reliability envelope. The deployed model also depends on the computational substrate that executes it, and that substrate can corrupt a correct model without producing a visible service failure.

Consider how hardware reliability directly impacts ML performance. As figure 14.5 illustrates, a single bit flip in a critical neural network weight can degrade ResNet-50 classification accuracy from 76 percent (top-1) to 11 percent on ImageNet, while memory subsystem failures during training corrupt gradient updates and prevent model convergence. Modern transformer models such as GPT-3 with 175B parameters execute enormous numbers of floating-point operations and create many opportunities for hardware faults during a forward pass. GPU memory systems operating at up to 900 GB/s bandwidth (such as V100 HBM2) process about 7.2×10^{12} bits per second. At a base error rate of 1.0×10^{-17} errors per bit processed, sustained peak bandwidth would yield about 0.26 errors/hour per device; at fleet scale, those low per-device rates compound into operationally visible fault rates.

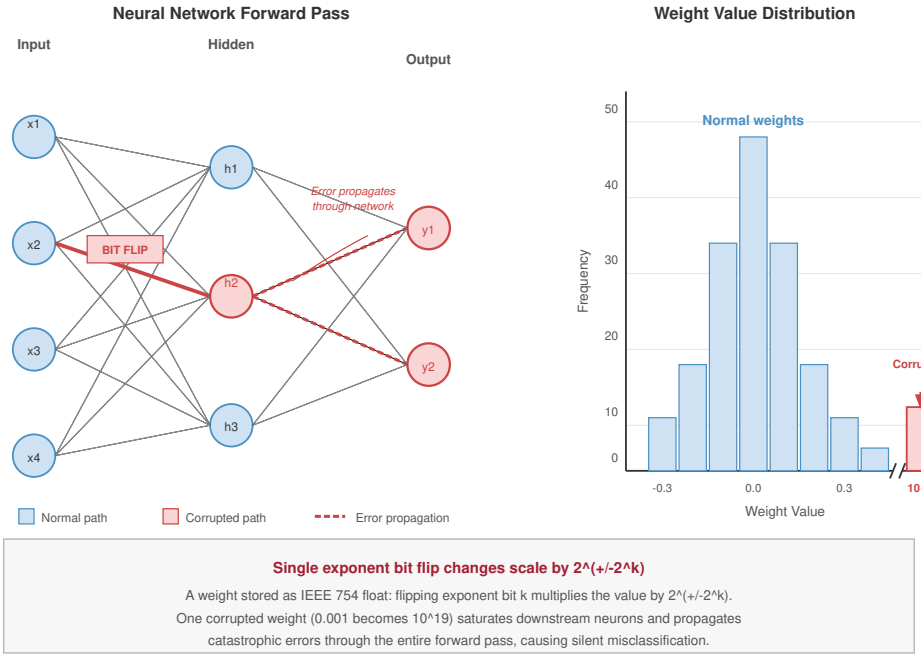


Figure 14.5: Weight Corruption via Bit Flip: A neural-network forward-pass diagram (left) shows a BIT FLIP on one weight, turning a normal path into a corrupted path whose error propagates to the outputs. The weight-value histogram (right) shows a single corrupted weight landing at the extreme 10^k tail, far outside the normal distribution. Flipping an exponent bit with place value 2^k in IEEE 754 changes the value by a factor of $2^{\pm 2^k}$, saturating downstream neurons and causing silent misclassification.

The connection between hardware reliability and ML performance demands concepts from reliability engineering[12]: fault models that describe how failures occur, error detection mechanisms that identify problems before they impact results, and recovery strategies that restore system operation. These reliability concepts complement performance optimization techniques such as quantization, pruning, and knowledge distillation by ensuring that optimized systems continue to operate correctly under real-world conditions.

Chapter 7 establishes that per-device silent corruption compounds across a fleet of N devices as $\Pr(\geq 1) = 1-(1-p)^N$, the same arithmetic that makes a single bit flip a near-certain event at training scale. The robustness consequence is what figure 14.6 extends: sweeping the per-device rate shows how steeply the cluster-level probability climbs once a model is sharded across thousands of devices.

12 **Reliability Engineering**: Originated in 1950s aerospace with MTBF analysis and failure-mode analysis; quantifies system reliability as $R_{\text{system}}(t) = e^{-N\lambda t}$ for N independent components with exponential failure distributions. ML systems inherit these methods but add failure modes that traditional reliability never anticipated: model drift (the system degrades without any hardware fault), adversarial robustness (the system is correct on the test set but fails on crafted inputs), and epistemic uncertainty (the system cannot distinguish what it knows from what it does not).

The curve uses an illustrative stress-test rate of 0.01 percent per device-hour. At that rate, a 10,000-device cluster is more likely than not to see an hourly silent error (63.2 percent probability), and the probability crosses 95 percent at about 29,956 devices. Meta's SDC report confirms corruption at observable fleet scale (Dixit et al. 2021).

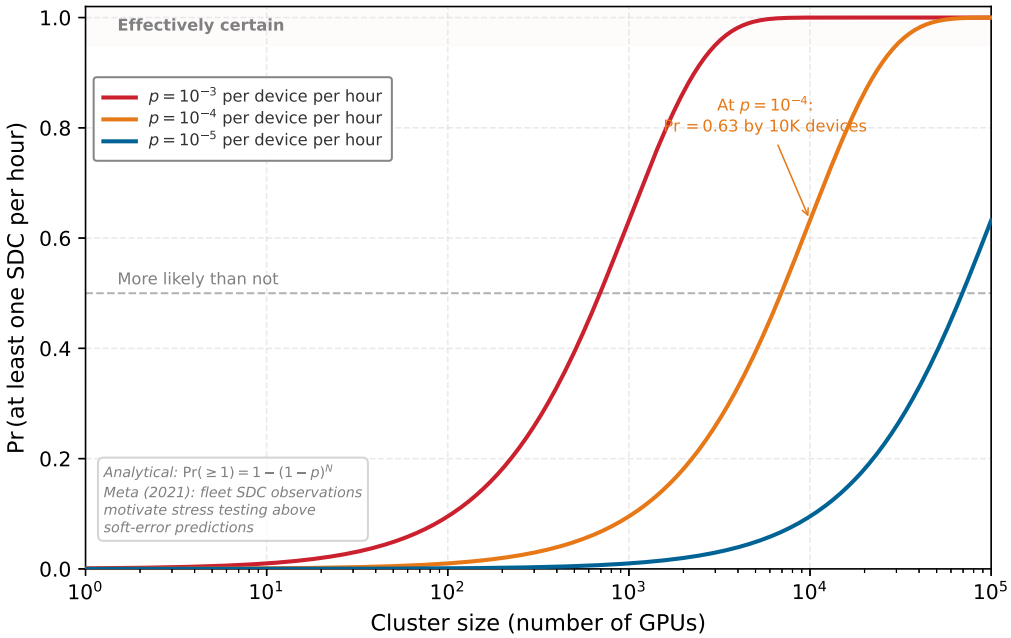


Figure 14.6: Silent Error Probability at Scale: Probability of at least one silent data corruption event per hour as a function of cluster size, for three illustrative per-device error rates. The curves show how quickly cluster-scale probabilities compound as per-device rates increase.

The compounding effect at cluster scale motivates a unified framework for robustness that spans all dimensions of ML systems. Faults originating from hardware, adversarial inputs, and software defects share common characteristics and yield to systematic approaches.

14.3.3 The three pillars of robust AI

The unified framework helps engineers decide which failure signal they are seeing before they choose a defense. Environmental shifts, input-level attacks, and system-level faults produce different evidence and require different responses; software faults cut across all three because they can amplify or masquerade as any of them. The **Three Pillars Framework** in figure 14.7 organizes these threats as interconnected vulnerabilities that require complementary defense strategies.

What the figure adds to the three categories introduced earlier is the evidence each pillar produces and the defense family it demands. Environmental shifts produce a statistical signal: the input or label distribution moves continuously against a reference, so the evidence is distributional distance and the response family is monitoring, recalibration, and retraining.

Input-level attacks produce an adversarial signal: a crafted input or poisoned sample is engineered to maximize error, so the evidence is gradient-aligned perturbation or anomalous training samples, and the response family is adversarial training, certification, and input sanitization. Because the attacker operates within the model's own input space, authentication and access control (Chapter 13) do not reach this failure.

System-level faults encompass failures originating from the hardware, code, frameworks, and deployment infrastructure that support ML systems: numerical instability in gradient computations, data pipeline corruption from preprocessing bugs, race conditions in distributed training, memory leaks that degrade long-running services, dependency failures from version mismatches, and hardware faults such as bit flips or power events. The fault mechanics themselves, and their detection

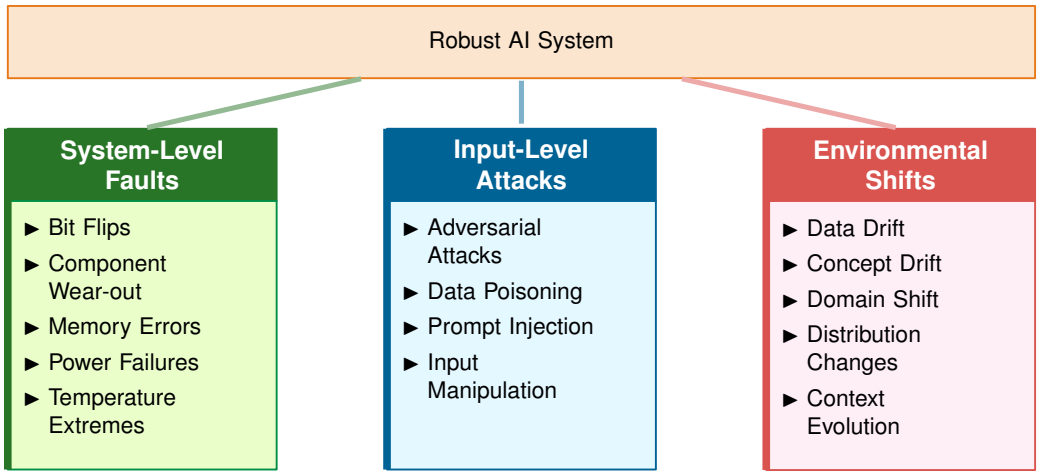


Figure 14.7: Three Pillars Framework: The three core categories of robustness challenges that AI systems must address to ensure reliable operation in real-world deployments. A robust AI system is built upon effectively handling these three challenge areas.

and recovery, belong to Chapter 7. What makes the third pillar a robustness problem rather than a pure reliability problem is that these faults rarely announce themselves as faults: they masquerade as the other two pillars, and an engineer who misreads the disguise spends the wrong defense budget.

14.3.3.1 Diagnosing the masquerade

Consider an operator who sees the same surface symptom from all three pillars: model accuracy is falling while latency and uptime hold steady. A preprocessing bug that silently rescales a feature looks exactly like covariate shift to a drift monitor, because the feature statistics have genuinely moved. A numerical overflow that corrupts a layer's activations produces confident misclassifications that look exactly like an adversarial example, because the decision flipped without a visible input cause. The disguise is the whole difficulty: the cheap pillar-specific detectors fire on the symptom, not the cause.

Three signals separate the cases:

- **Boundary of change**: Genuine environmental shift moves continuously and affects whole populations of inputs as the world evolves; a pipeline fault appears as a step discontinuity synchronized with a deploy, a dependency bump, or a schema change, and it can move features that no real-world process would move together.
- **Fixed-input reproducibility**: Drift and adversarial perturbation are properties of the input distribution, so replaying a saved input through the current pipeline reproduces the correct historical output; a hardware or numerical fault is a property of the computation, so the same saved input now yields a different answer. Replaying a golden input set is the single most discriminating test, and it is why pipeline parity and preprocessing-version checks belong in the robustness toolkit alongside drift monitors.
- **Cross-layer correlation**: A real adversarial campaign correlates with input-space anomalies such as unusual query patterns or gradient-aligned perturbations; a masquerading software fault correlates instead with system-level signals, including a code release, an ECC counter, an SDC checker, or a memory-pressure alarm, that Chapter 7 already instruments.

The diagnostic discipline is therefore to read the system-level evidence before accepting the drift or attack hypothesis the surface symptom suggests, because the response each pillar demands is different and only one of them is correct.

14.3.4 Common robustness principles

Across all three categories, the shared engineering problem is deciding which signal triggers which response. Robust systems need a detection threshold, a degradation path, and an adaptation mechanism, each with an explicit cost budget.

Detection and monitoring form the foundation of that strategy. Each pillar asks for a different signal. System-level monitoring samples hardware and runtime metrics to catch temperature anomalies, voltage fluctuations, memory errors, or silent data corruption before they corrupt model state. Input-level attack monitoring uses statistical or activation-space tests to flag adversarial inputs and poisoning attempts before they reach the decision boundary or training loop. Environmental-shift monitoring compares production traffic with reference distributions using tools such as Maximum Mean Discrepancy (MMD),[13] Population Stability Index (PSI), or Kolmogorov-Smirnov (K-S) tests. The same quantitative discipline applies to defense cost: robustness mechanisms must be budgeted, not merely enabled, because every detector trades sensitivity against false positives, latency, and compute overhead.

[13] **Maximum Mean Discrepancy (MMD)**: A kernel-based statistical test measuring distance between two distributions in a reproducing kernel Hilbert space, without parametric assumptions. Unlike univariate tests (K-S, PSI) that require per-feature evaluation, MMD operates on joint distributions natively—critical for ML inputs where drift manifests in feature correlations, not individual features. The trade-off is compute: MMD scales $\mathcal{O}(n^2)$ in sample size, making it impractical for real-time monitoring without subsampling or random feature approximations.

> Napkin Math 14.1: The cost of defense
>
> **Problem**: A team is training a robust classifier for an autonomous vehicle, using adversarial training with a seven-step projected-gradient descent attack (PGD-7) that searches for a worst-case perturbation for every training batch. How much does this extra attack-generation work slow down the training run?
>
> **Math**: Generating an adversarial example requires K additional gradient steps per training sample.
>
> 1. **Forward/backward passes**: 1 (Standard) + 7 (Attack Generation) = 8 total passes.
> 2. **Training Slowdown**: 8× slower.
> 3. **Utility Cost**: Accuracy against the worst-case attack is 70 percent, compared with 95 percent clean-data accuracy for the standard model.
>
> **Systems insight**: Robustness is an efficiency-utility trade-off. In this PGD-7 example, the team pays 8× the training cost and exposes a 25 percentage-point accuracy gap between standard clean-data accuracy and robust accuracy under the specified worst-case digital perturbation threat model. This notebook measures a training-cost scenario; the ResNet-50 robustness-tax example in section 14.6.1 measures the separate clean-accuracy tax of building adversarial robustness into the model weights. In the Machine Learning Fleet, "Robustness" is not a setting that can be flipped on; it is a budget the team spends. This budget pressure often makes detection attractive for lower-risk components, while certification and robust training are easier to justify for safety-critical paths.

Graceful degradation turns detection into a bounded operating mode instead of a crash. Robust systems exhibit predictable performance reduction that preserves critical capabilities. ECC memory systems recover from single-bit errors with 99.9 percent success rates while adding 12.5 percent bandwidth overhead. Model quantization from FP32 to INT8 reduces memory requirements by 75 percent and inference time by 2–4×, trading 1–3 percent accuracy for continued operation under resource constraints. Ensemble fallback systems trade peak accuracy for continuity, holding most of peak performance when a primary model fails and switching over fast enough to stay within a real-time serving budget.

Adaptive response completes the loop by changing system behavior when the signal persists. Adaptation might involve activating error correction mechanisms, applying input preprocessing techniques, or dynamically adjusting model parameters. The key principle is that robustness is not static but requires ongoing adjustment to maintain effectiveness.

Detection, degradation, and adaptation extend beyond fault recovery to form a systematic performance adaptation strategy that appears throughout ML system design. Figure 14.8 expands the same three pillars from figure 14.7 into concrete failure subtypes, then attaches the shared response pattern to each subtype: detection strategies form the foundation for monitoring systems, graceful

degradation guides fallback mechanisms when components fail, and adaptive response enables systems to evolve with changing conditions.

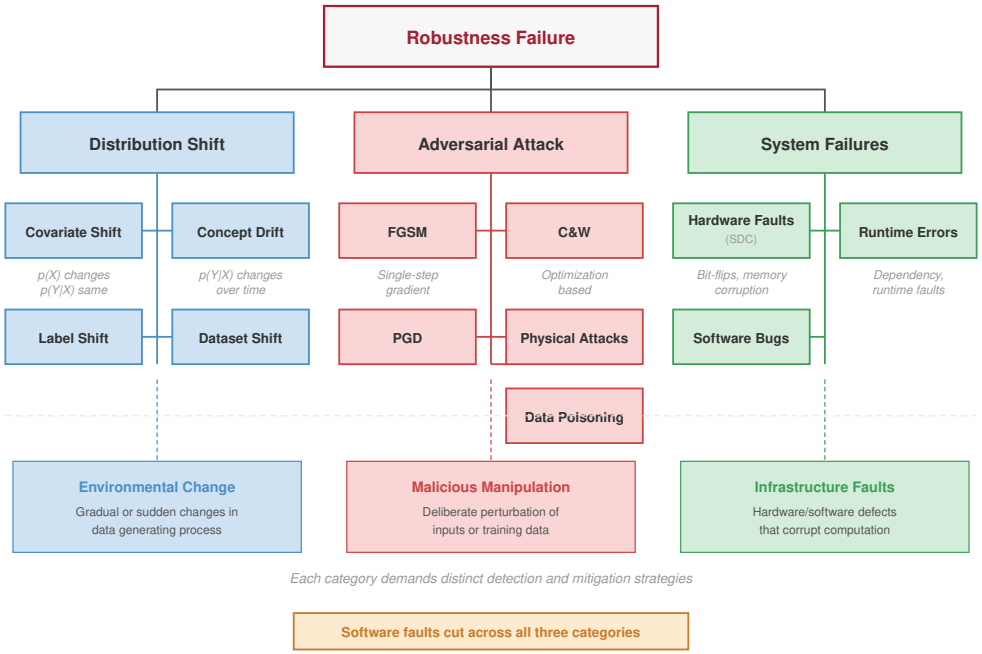


Figure 14.8: The Robustness Taxonomy: Environmental shifts, input-level attacks, and system-level faults decomposed into concrete failure subtypes. Software faults can appear within or amplify all three categories.

The taxonomy in figure 14.8 reveals that no single defense covers all three pillars: environmental shifts, input-level attacks, and system-level faults each require distinct detection, degradation, and adaptation mechanisms, making defense-in-depth the core strategy for production systems.

14.3.5 Integration across the ML pipeline

Robustness cannot be bolted onto a trained model; it is a quality attribute enforced at every stage of the ML lifecycle, a principle often called defense in depth. In the data ingestion phase, sanitization filters must reject malformed or statistically anomalous records before they enter the training set, preventing data poisoning attacks at the source. During training, adversarial training directly exposes the model to worst-case perturbations, while randomized smoothing later turns noisy repeated predictions into a certifiable robustness bound. Both families try to limit how quickly outputs can change as inputs change, the intuition behind the model's **Lipschitz constant**[14]. Validation extends beyond simple accuracy metrics to include stress testing on out-of-distribution (OOD) datasets, ensuring the model's decision boundary is well-behaved in the open world.

Once deployed, the focus shifts to runtime defense. A robust inference server complements its serving architecture with the detection techniques in section 14.6.1.2, including input filtering that intercepts adversarial queries before they reach the accelerator. For a production fraud detection pipeline, this layered approach yields compound benefits: cheap statistical validation catches only the crudest poisoning attempts during data ingestion, while semantic input filtering at serving time blocks a much larger share of sophisticated evasion attacks. The monitoring layer acts as the safety net, detecting distribution drift—such as a sudden shift in transaction amounts or user geolocations—within days to weeks, triggering retraining workflows before performance degrades below the service level objective (SLO).

[14] **Lipschitz Continuity**: A mathematical property that bounds how much a function's output changes relative to its input change ($\|f(x) - f(x')\| \leq K\|x - x'\|$). In robust AI, minimizing the Lipschitz Constant (K) ensures the model's decision surface is "smooth" rather than "jagged," making it physically impossible for small adversarial perturbations to flip the model's prediction, though enforcing a low constant trades away clean-data accuracy and adds training cost.

The holistic view integrates with hardware reality. Hardware faults (transient, permanent, and intermittent) are covered in detail in section 7.2, where they integrate with the broader fault detection and recovery mechanisms for distributed systems. A robust software pipeline treats silent data corruption in the ALU or a bit flip in HBM as another form of noise to be filtered or retried, not as an exceptional crash. With the lifecycle and hardware frame in place, the chapter now turns to the most common source of model degradation: the real world constantly evolves while training datasets remain frozen in time.

Checkpoint 14.1: Diagnosing the failure signal

The unified framework asks you to name which of the three pillars produced a silent failure before choosing a defense, and to recognize that software faults can masquerade as any of them.

Classifying the threat

- ☐ **Pillar classification**: Classify a model losing accuracy as customer behavior evolves, a crafted input that forces misclassification, and an accelerator HBM bit flip under the correct robustness pillars.
- ☐ **Software-fault mimicry**: Explain why a preprocessing bug that shifts every inference's feature statistics is hard to distinguish from genuine environmental shift, then classify it under the appropriate pillar.

Choosing the response

- ☐ **Response selection**: Map an ECC-corrected single-bit memory error and a fired drift monitor to the shared responses of detection, graceful degradation, and adaptive response.
- ☐ **Fleet-scale effects**: Explain why a negligible per-device silent-corruption rate becomes an architectural constraint at a fleet scale of tens of thousands of accelerators.

14.4 Environmental Shifts

Training data freezes a past world, while production traffic keeps changing. Environmental shifts are the robustness failures that follow from this mismatch: data distributions, user behavior, and operational contexts move after the model has learned its boundary. These shifts also interact with other vulnerability types: a model experiencing distribution shift becomes more susceptible to adversarial attacks, while software errors may manifest differently under changed environmental conditions.

14.4.1 Distribution shift and concept drift

A medical diagnosis model trained on X-ray images from a well-resourced hospital plummets in accuracy when deployed in a rural clinic with older equipment. The underlying medical conditions have not changed; the image characteristics differ. The world the model encounters differs from the world it learned from, and the result is distribution shift.

Napkin Math 14.2: Detecting a real distribution shift

Problem: A model monitors a critical input feature with a known standard deviation of 0.3. The baseline mean was 0.5. Over the last 1,000 requests, the mean has shifted to 0.55. The engineering question is whether this is a random fluctuation or a real distribution shift.

Math: Detection requires proving that the observed change is statistically unlikely under the baseline distribution.

1. **Difference in means**: 0.05.
2. **Standard error**: $0.3/\sqrt{1{,}000} \approx 0.009$.

3. **Statistical significance**: The shift is approximately 5.3 standard errors away from the mean.
4. **P-value**: < 0.001.

Systems insight: Statistical significance is the signal-to-noise ratio of the monitoring system. A shift of 0.05 might seem "small," but with 1,000 samples, the probability of it being random noise is less than 0.1 percent. In the machine learning fleet, this is a confirmed drift alert, not yet a confirmed model regression. The system should trigger investigation, increased monitoring, and correlation with precision, recall, latency, and business metrics; model fallback or retraining is warranted only when the shifted feature is high importance, the drift crosses severe thresholds, or service-level metrics degrade.

The taxonomy in figure 14.9 separates the three failure modes that a drift detector can surface: the inputs can move, the label prior can move, or the input-label relationship itself can change.

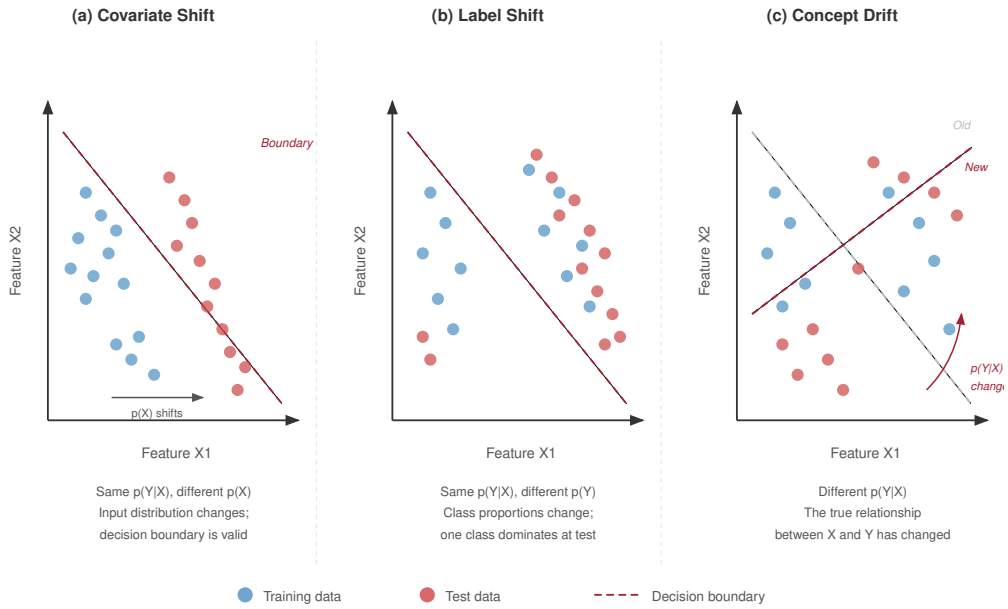


Figure 14.9: Types of Distribution Shift: Comparison of covariate shift ($p(x)$ changes), label shift ($p(y)$ changes), and concept drift ($p(y \mid x)$ changes). Understanding the specific type of shift is crucial for selecting the correct adaptation strategy (for example, importance re-weighting vs. model retraining).

These shifts occur naturally as environments evolve. User preferences change seasonally, language evolves with new slang, and economic patterns shift with market conditions. Unlike adversarial attacks that require malicious intent, these shifts emerge organically from the dynamic nature of real-world systems.

14.4.1.1 Technical categories

Covariate shift occurs when the input distribution changes while the relationship between inputs and outputs remains constant (Quiñonero-Candela et al. 2009). Autonomous vehicle perception models trained on daytime images can experience accuracy degradation on the order of 15–30 percent when deployed in nighttime conditions despite the underlying object recognition task being unchanged, with the magnitude depending on luminance shift and sensor characteristics. Weather conditions introduce additional covariate shift: rain, snow, and fog are widely reported

to drop object detection mAP by roughly 10–25 percent compared to clear-weather baselines in autonomous-driving evaluations. These numbers should be read as representative magnitudes from autonomous-vehicle perception benchmarks rather than a single cited result. These environmental changes effectively shift data points relative to the learned decision boundary (figure 14.10), causing misclassification without any change to the model itself.

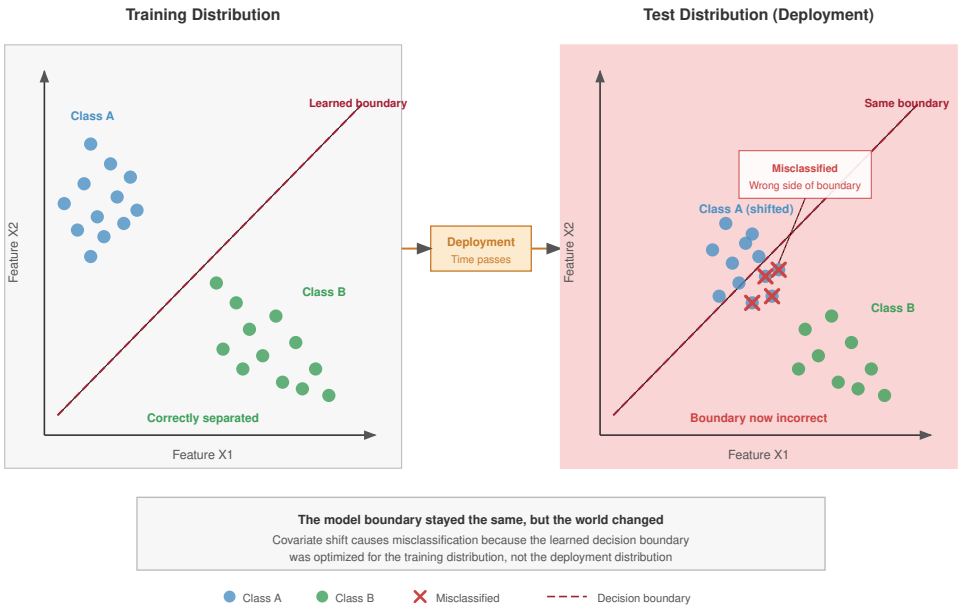


Figure 14.10: Decision Boundary Under Distribution Shift: Environmental changes (for example, daytime to nighttime, clear to foggy) shift data points in the input space. When the shift moves points across the learned decision boundary, the model misclassifies inputs that it would have handled correctly under training conditions. Unlike adversarial perturbations, these shifts arise naturally and affect entire populations of inputs rather than individual examples.

Figure 14.10 illustrates the case where input distributions move while the true mapping $p(y \mid x)$ stays fixed. A more insidious variant occurs when the mapping itself changes: the correct label for a given input today is different from what it was during training.

Definition 14.2: Concept drift

Concept Drift is the deployed-model subtype of distribution shift (see section 14.4.1) in which the statistical relationship $p(y \mid x)$ changes over time, meaning the decision boundary itself becomes incorrect rather than merely the input distribution. Its sibling is data drift (see section 12.6), in which $p(x)$ changes while $p(y \mid x)$ remains stable.

1. **Significance**: It causes silent model degradation because the historical mapping learned by the model is no longer representative of current reality. Within the iron law, it compresses the effective deployment window before retraining is required: fraud-detection models, recommender systems, and other behavior-dependent models may need periodic retraining or recalibration as adversaries, users, and policies change. Each forced retraining cycle incurs the full $O/(R_{\text{peak}} \cdot \eta_{\text{hw}})$ cost of the original training run, making the amortized per-prediction cost a direct function of drift velocity.
2. **Distinction**: Unlike data drift (where fresh $p(x)$ data with unchanged labels fully restores performance), concept drift requires relabeling under the new $p(y \mid x)$, because the same ground-truth labeling procedure that cures data drift is insufficient when the correct answer for the same input has changed. This makes concept drift structurally more

expensive to remediate: it demands human annotation of recent examples, not merely resampling of the existing labeled distribution.

3. **Common pitfall**: A frequent misconception is that concept drift is detectable by monitoring input feature statistics. Because $p(x)$ may be entirely unchanged, input-level monitoring (PSI, KL divergence on features) will show no signal. Concept drift can only be confirmed by comparing predictions to ground-truth outcomes, making it significantly harder to detect in real time and requiring a ground-truth feedback loop before remediation can begin.

Concept drift represents changes in the underlying relationship between inputs and outputs over time (Widmer and Kubat 1996). In production, this often appears in domains such as fraud detection or recommendation, where adversaries, seasonal patterns, and user preferences change the label relationship and force periodic recalibration or retraining.

Label shift affects the distribution of output classes without changing the input-output relationship (Lipton et al. 2018). During COVID-19, for example, hospital case mix and disease prevalence changed rapidly, so diagnostic models could require threshold recalibration even when image features carried the same clinical meaning. Similar class-prevalence shifts can occur as seasons, policies, or user populations change, requiring recalibration or reweighting rather than assuming that the feature-label relationship itself has changed.

Models can also fail because they learned the wrong lessons from the training data, not because the world changed. A classic example is a model that learns to identify "cow" by detecting "grass" background. When presented with a cow on a sandy beach, the model fails. The underlying cause is a **spurious correlation**: a feature that is predictive in the training set but not causally related to the label.

Standard training by empirical risk minimization (ERM) encourages these shortcuts because they are often statistically easier to learn than the robust features (shape, texture). Techniques like **Group Distributionally Robust Optimization (Group DRO)** explicitly mitigate this by minimizing the *worst-case group loss* (for example, cows on sand) rather than the *average loss*. The method requires groups to be known or inferred in advance, but when those groups are available, it forces the model to learn features that work across all contexts.

14.4.2 Monitoring and adaptation strategies

Drift monitoring earns its place only when it turns a distribution signal into an operating decision: investigate, adapt, retrain, or continue watching.

Statistical distance metrics quantify the degree of distribution shift by measuring differences between training and deployment data distributions. In this illustrative H100-class monitoring scenario, Maximum Mean Discrepancy (MMD) with RBF kernels ($\gamma = 1.0$) processes 10,000 samples in 150 ms; its sensitivity depends on the shift model and kernel choice. Kolmogorov-Smirnov tests can detect univariate shifts with 1,000+ samples, but scale poorly to high-dimensional data and miss joint changes that preserve marginals. Population Stability Index (PSI)[15] thresholds of 0.1 to 0.25 indicate significant shift requiring model investigation.

Once a monitor fires, adaptation becomes a budgeted response instead of an automatic retraining command. Online learning enables models to continuously adapt to new data while maintaining performance on previously learned patterns (Shalev-Shwartz 2012). The adaptation budget depends on model size, drift rate, and feedback latency: updating too aggressively can chase noise, while updating too slowly lets performance drift. In production, online-learning systems usually bound update frequency, state size, and serving latency explicitly rather than assuming adaptation is free. Techniques like Elastic Weight Consolidation reduce catastrophic forgetting by penalizing changes to parameters important for previous tasks (Kirkpatrick et al. 2017).

Adaptive ensemble methods maintain multiple models or hypotheses and weight or select among them using recent performance, making them useful under gradual concept drift (Gama et al. 2014). This approach trades extra serving and monitoring complexity for the ability to respond when a single static model no longer matches the deployment distribution.

[15] **Population Stability Index (PSI)**: Originally developed in the 1980s for credit scoring to detect whether the demographic of current loan applicants shifted from the historical baseline. In ML monitoring, PSI's symmetric log-ratio formulation makes it a common industry tool for identifying data drift in categorical features, providing a single scalar trigger for retraining workflows.

Federated learning enables distributed adaptation when the data cannot be centralized for privacy, regulatory, or bandwidth reasons. The adaptation then has to travel to the data instead, which makes the communication budget, not compute, the binding constraint: each round ships model parameters across many participants, so the design question is how many rounds and how much per-round transmission the deployment can afford. The federated mechanics belong to Chapter 11; the robustness-relevant point is that any privacy noise added to protect participants (for example, through differential privacy) carries a utility cost that must be measured for the application rather than assumed away.

14.4.3 Quantitative drift detection

Quantitative drift detection must answer an operational question: whether the model should keep serving, be monitored more closely, or be retrained. PSI supplies the cheap fleet-wide alert that starts that decision, while the mathematical foundations and operational thresholds below transform drift detection from a subjective judgment into an engineering discipline.

14.4.3.1 Population stability index (PSI)

Section 12.6.4.3 introduced the Population Stability Index as a cheap fleet-wide alerting signal, with its credit-scoring origin and the standard threshold bands. This section develops the full statistical machinery behind that signal: how PSI is computed, what binning and smoothing choices govern its sensitivity, and how it combines with KL divergence and significance tests into a retraining decision. PSI measures the divergence between an expected (baseline) distribution p_{base} and an actual (current) distribution p_{curr} by computing a symmetric log-ratio difference across discretized bins.

For a feature discretized into k bins, PSI is defined as:

$$\text{PSI} = \sum_{i=1}^{k} (p_i - q_i) \times \ln\left(\frac{p_i}{q_i}\right)$$

where p_i represents the proportion of observations in bin i for the baseline distribution and q_i represents the corresponding proportion in the current distribution. The logarithmic term penalizes large relative changes, while the $(p_i - q_i)$ term weights by absolute magnitude. Established threshold bands translate these PSI values into actionable decisions.

Table 14.1 collects common PSI ranges and the recommended monitoring action at each tier. These threshold bands are monitoring conventions, especially common in credit-scoring practice, rather than universal statistical guarantees; PSI should be interpreted together with feature importance and downstream model-performance metrics (Yurdakul and Naranjo 2020).

Table 14.1: PSI Interpretation Thresholds: Common Population Stability Index ranges and the recommended monitoring action at each tier.

PSI Value	Interpretation	Recommended Action
$\text{PSI} < 0.1$	Negligible shift	Continue monitoring
$0.1 \leq \text{PSI} < 0.2$	Minor shift	Investigate root cause
$0.2 \leq \text{PSI} < 0.25$	Moderate shift	Consider retraining
PSI ≥ 0.25	Major shift	Retrain required

Several implementation choices determine whether PSI is sensitive enough to be useful. Bin selection significantly affects PSI sensitivity. For categorical features, each category forms a natural bin. For continuous features, equal-width bins (10–20 bins typical) or quantile-based bins provide different trade-offs: equal-width bins preserve the absolute scale of the feature space, while quantile bins ensure adequate sample sizes in each bin but may mask shifts in the tails. Production systems often use ten bins with a minimum of 5 percent of observations per bin to ensure statistical stability.

When a bin has zero observations in either distribution, adding a small smoothing constant (typically $\epsilon_{\text{smooth}} = 10^{-8}$) prevents undefined logarithms while minimally affecting the PSI value. The subscript keeps this numerical safeguard distinct from the adversarial perturbation radius ϵ

introduced later in the chapter. As figure 14.11 shows, monitoring PSI over time reveals when a model drifts from stable (Green Zone) into warning (Orange) and critical (Red Zone) regions, triggering an escalation path that may lead to retraining after performance correlation.

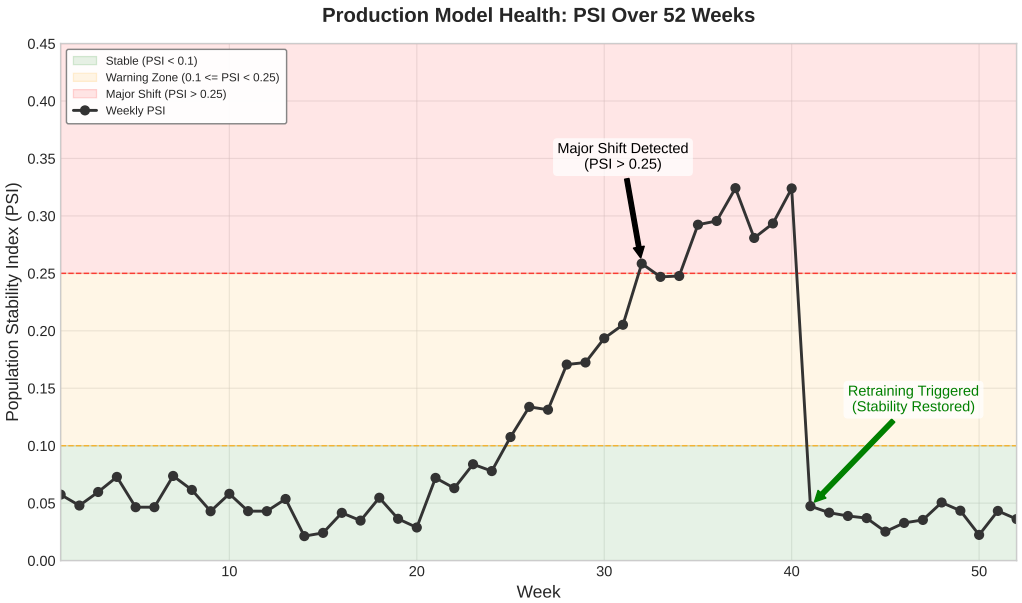


Figure 14.11: The Distribution Shift Detector: Monitoring Population Stability Index (PSI) over a year. The model remains stable through week 24, enters the orange warning zone at week 25 as user behavior shifts, and breaches the red critical threshold at week 32. Automated retraining restores model stability at week 41.

14.4.3.2 Kullback-Leibler divergence

For continuous features where binning may lose information, Kullback-Leibler (KL) divergence provides a more direct measure of distributional difference. The KL divergence from baseline distribution p_{base} to current distribution p_{curr} is defined as:

$$\mathcal{D}_{\text{KL}}(p_{\text{base}} \| p_{\text{curr}}) = \int_{-\infty}^{\infty} p_{\text{base}}(x) \ln\left(\frac{p_{\text{base}}(x)}{p_{\text{curr}}(x)}\right) dx$$

where $p_{\text{base}}(x)$ and $p_{\text{curr}}(x)$ are the probability density functions of the baseline and current distributions, respectively. Unlike PSI, KL divergence is asymmetric: $\mathcal{D}_{\text{KL}}(p_{\text{base}} \| p_{\text{curr}}) \neq \mathcal{D}_{\text{KL}}(p_{\text{curr}} \| p_{\text{base}})$. For drift detection, we typically compute $\mathcal{D}_{\text{KL}}(\text{baseline} \| \text{current})$, measuring how much information is lost when using the current distribution to approximate the baseline.

To address asymmetry, practitioners often use the Jensen-Shannon divergence:

$$\mathcal{D}_{\text{JS}}(p_{\text{base}} \| p_{\text{curr}}) = \frac{1}{2}\mathcal{D}_{\text{KL}}(p_{\text{base}} \| p_{\text{mix}}) + \frac{1}{2}\mathcal{D}_{\text{KL}}(p_{\text{curr}} \| p_{\text{mix}})$$

where $p_{\text{mix}} = \frac{1}{2}(p_{\text{base}} + p_{\text{curr}})$ is the mixture distribution. Jensen-Shannon divergence is bounded between 0 and $\ln(2)$ (approximately 0.693), making threshold selection more intuitive than unbounded KL divergence.

For drift monitoring in production, table 14.2 gives practical thresholds for interpreting KL divergence values.

Table 14.2: KL Divergence Threshold Reference: Practical thresholds for interpreting KL divergence values during drift monitoring. Values below 0.05 indicate minimal divergence; 0.05 to 0.1 indicates moderate divergence that warrants investigation; values at or above 0.1 indicate significant divergence that typically triggers retraining workflows.

$\mathcal{D}_{\text{KL}}$ Value	Interpretation
$\mathcal{D}_{\text{KL}} < 0.05$	Minimal divergence
$0.05 \leq \mathcal{D}_{\text{KL}} < 0.1$	Moderate divergence
$\mathcal{D}_{\text{KL}} \geq 0.1$	Significant divergence

For practical computation, kernel density estimation (KDE) with Gaussian kernels provides smooth density approximations suitable for integration, though computational cost scales as $\mathcal{O}(n^2)$ for n samples, making sampling necessary for large datasets.

14.4.3.3 Statistical significance testing

PSI and KL divergence quantify how large a distributional change appears to be; statistical hypothesis tests ask the complementary question of whether the observed difference is larger than sampling noise. The two-sample Kolmogorov-Smirnov (KS) test (Berger and Zhou 2014) compares the empirical cumulative distribution functions (CDFs) of two samples without assuming any specific parametric form. The test statistic is:

$$D_{n,m} = \sup_x |F_n(x) - G_m(x)|$$

where F_n and G_m are the empirical CDFs of samples of size n and m respectively. The null hypothesis (no distributional difference) is rejected when:

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}}$$

where $c(\alpha)$ depends on the significance level (for example, $c(0.05) \approx 1.36$). The KS test is particularly effective for detecting shifts in location (mean) and spread (variance) but less sensitive to changes in distribution shape.

For categorical features, the **chi-square goodness-of-fit test** compares observed frequencies to expected frequencies under the baseline distribution:

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i^{\text{obs}} - n_i^{\text{exp}})^2}{n_i^{\text{exp}}}$$

where n_i^{obs} is the observed count in category i and n_i^{exp} is the expected count based on the baseline distribution. With $k-1$ degrees of freedom, the null hypothesis is rejected when χ^2 exceeds the critical value for significance level α.

When monitoring many features simultaneously, the significance test must also account for repeated comparisons. Applying Bonferroni correction (dividing α by the number of tests) or false discovery rate (FDR) control prevents excessive false alarms. For m features at significance level $\alpha = 0.05$, Bonferroni requires each test to achieve $p < 0.05/m$ for significance.

14.4.3.4 Worked example: Production fraud detection model

Consider a fraud detection model serving an e-commerce platform with two key input features: user country (categorical) and transaction amount (continuous). After six months in production, the operations team suspects distribution drift and must decide whether to retrain.

Step 1: Categorical feature analysis Table 14.3 compares the baseline (training) distribution and current (production) distribution for four named countries plus the Other bucket.

Table 14.3: Country-Feature PSI Decomposition: Per-country contribution to the total PSI for the user country feature in a production fraud-detection model. The "Other" bucket drives most of the divergence (0.0470 of the 0.065 total), reflecting growth of non-USA/UK/Germany/France traffic over six months in production.

Country	Baseline (p_i)	Current (q_i)	$p_i - q_i$	$\ln(p_i/q_i)$	Contribution
USA	0.45	0.38	0.07	0.169	0.0118
UK	0.20	0.18	0.02	0.105	0.0021
Germany	0.15	0.14	0.01	0.069	0.0007
France	0.10	0.12	-0.02	-0.182	0.0036
Other	0.10	0.18	-0.08	-0.588	0.0470

Summing the per-country contributions gives $\text{PSI}_{\text{country}} = 0.0118 + 0.0021 + 0.0007 + 0.0036 + 0.0470 = 0.065$.

The PSI of 0.065 indicates negligible drift in user country distribution, falling well below the 0.1 threshold. No action required for this feature.

Step 2: Continuous feature analysis For the transaction amount feature (log-transformed for normality), compute KL divergence using kernel density estimation:

- Baseline distribution: $\mu = 4.2$, $\sigma = 1.1$ (log-dollars)
- Current distribution: $\mu = 4.5$, $\sigma = 1.3$ (log-dollars)

For approximately Gaussian distributions, KL divergence has a closed-form solution:

$$\mathcal{D}_{\text{KL}} = \ln \frac{\sigma_{\text{curr}}}{\sigma_{\text{base}}} + \frac{\sigma_{\text{base}}^2 + (\mu_{\text{base}} - \mu_{\text{curr}})^2}{2\sigma_{\text{curr}}^2} - \frac{1}{2}$$

The closed-form solution evaluates to $\mathcal{D}_{\text{KL}} = \ln \frac{1.3}{1.1} + \frac{1.30}{3.38} - 0.5 = 0.167 + 0.385 - 0.5 = 0.052$. The KL divergence of 0.052 indicates moderate drift, warranting further investigation but not immediate retraining.

Step 3: Statistical significance via KS test Using the KS test on 10,000 baseline samples and 10,000 current samples for transaction amount:

$$D_{10000,10000} = 0.089$$

Critical value at $\alpha = 0.05$: $c(0.05)\sqrt{\frac{20000}{10^8}} \approx 0.019$

Since the observed statistic 0.089 exceeds the critical value 0.019, the difference is statistically significant $(p < 0.001)$. However, *statistical significance* alone does not mandate retraining; the *practical significance* (PSI, KL values) suggests monitoring rather than immediate action.

Step 4: Decision framework application Table 14.4 combines the quantitative evidence.

Table 14.4: Retraining Decision Matrix: Combined quantitative evidence (PSI, KL divergence, KS test) compared against thresholds and the resulting action level. Drift signals below the practical threshold but with statistical significance call for monitoring at increased frequency rather than immediate retraining.

Metric	Value	Threshold	Action Level
PSI (country)	0.065	< 0.1	Monitor
$\mathcal{D}_{\text{KL}}$ (amount)	0.052	< 0.1	Monitor
KS test	$p < 0.001$	$\alpha = 0.05$	Significant

Decision: Continue monitoring with increased frequency (weekly instead of monthly). If PSI or KL divergence exceeds 0.1 in the next monitoring cycle, or if model performance metrics (precision, recall) degrade by more than 5 percent, initiate retraining. The example stops at a monitoring decision; the general framework turns that same logic into a repeatable retraining gate.

14.4.3.5 Retraining decision framework

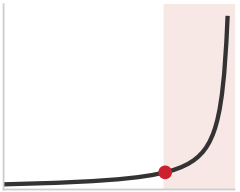
A systematic decision framework integrates drift metrics with performance monitoring to determine optimal retraining timing. The three levels below deliberately separate detection, correlation, and action so metric alerts do not become automatic retraining commands.

Level 1: Automated monitoring Configure automated alerts for three drift thresholds:

- PSI > 0.1 on any high-importance feature
- $\mathcal{D}_{\text{KL}} > 0.05$ on continuous features
- KS test p-value < 0.01 with Bonferroni correction

Level 2: Performance correlation When drift alerts trigger, three performance correlations determine the response:

- If performance degradation exceeds 3 percent and coincides with drift: Initiate retraining
- If drift detected but performance stable: Continue monitoring, investigate drift source
- If performance degrades without detected drift: Investigate concept drift or label shift



Drift stays benign until the index crosses a threshold; past the knee, reliability degrades fast.

Level 3: Retraining vs. investigation Not all drift requires retraining. Table 14.5 separates immediate remediation from investigation and continued monitoring.

Table 14.5: Retraining Decision Rules: Drift response depends on the strength of the statistical signal, observed performance change, and external evidence about whether the environment has truly changed.

Action	Trigger conditions	Response logic
Retrain immediately	• PSI > 0.25 on critical features and performance degraded by more than 5% • Concept drift confirmed ($p(y \mid x)$ changed) • Regulatory or compliance requirements mandate fresh models	The learned mapping, compliance baseline, or production population is no longer valid enough for monitoring alone
Investigate first	• $0.1 \leq$ PSI ≤ 0.25 with stable performance • Drift localized to nonpredictive features • Drift may be temporary, such as seasonal effects or one-time events	The signal is real, but the remediation could be more expensive or riskier than the current degradation
Continue monitoring	• PSI < 0.1 across all features • Performance within acceptable bounds • No external signals suggesting environmental change	The evidence does not yet justify changing the model, but the baseline should remain under observation

The quantitative framework transforms drift detection from reactive troubleshooting into proactive model maintenance, enabling ML systems to maintain reliability as production environments evolve (Gama et al. 2014).

14.4.4 Robustness in generative AI

Large language models (LLMs) shift the failure surface from incorrect classification to semantic reliability: a fluent answer can be factually baseless while the system still appears healthy. The earlier robustness question was whether a label changed under perturbation; the generative version is whether an open-ended output preserves factuality, policy constraints, and task intent under prompt variation. Evaluations in specialized domains such as legal or medical advice show that hallucination rates vary strongly with model, retrieval context, prompt design, and sampling temperature. Addressing this requires rigorous **Uncertainty Quantification (UQ)**: a robust system must be self-aware enough to flag when it is guessing. One approach monitors the entropy of the output distribution; a "flat" probability distribution across the vocabulary indicates high uncertainty, which can trigger a fallback to a human operator or a refusal to answer. Log-probabilities at the token level reveal segments where the model transitions from confident generation to speculative completion.

More advanced UQ techniques involve **Self-Consistency**, where the model is prompted to generate multiple distinct reasoning paths for the same query. If five sampling runs produce five contradictory answers to a factual question, the system treats the output as unstable and suppresses it. This statistical approach transforms the nebulous concept of "truthfulness" into a measurable variance

metric that integrates naturally with the MLOps monitoring pipeline (Chapter 12). Predictive entropy—aggregating the Shannon entropy across the full output sequence—provides a scalar score that can be thresholded to route high-risk generations for human review.

In Retrieval-Augmented Generation (RAG) architectures (Chapter 10), robustness depends heavily on the quality of the retrieved context. **Retrieval Noise**, the injection of irrelevant or conflicting documents into the prompt, can distract the model, causing it to ignore its internal parametric knowledge and propagate errors from the context. Robust RAG deployments can use re-ranking models and context verifiers to filter out noise before it reaches the generation step.

Generative models also face the unique threat of Prompt Injection, where an attacker embeds instructions within the input data to override the model's system prompt. While often discussed as a security issue in Chapter 13, prompt injection is equally a robustness failure: a model that can be easily manipulated into ignoring its behavioral constraints has failed to maintain its output invariants under adversarial input. A common deployment pattern is to add **Output Guardrails**—lightweight classification models that scan generated text for policy violations, toxicity, or logical errors before returning the response to the user. This final validation step helps keep the system as a whole reliable even if the core model enters a failure mode.

While prompt injection exploits the linguistic flexibility of generative models, it represents a bridge between natural environmental shifts and deliberate adversarial manipulation. The interaction runs both directions: distribution monitoring systems themselves can be exploited by adversaries who craft inputs that evade drift detection thresholds, turning a defensive tool into a blind spot. When an adversary stops relying on natural drift and actively begins reverse-engineering the model's decision boundaries to force specific errors, we move from the domain of environmental robustness into the mathematically rigorous battleground of input-level attacks.

Checkpoint 14.2: Drift detection and the retraining decision

Environmental shifts split into distinct types, and the chapter's drift framework deliberately separates a fired metric from a retraining command.

Distinguishing the shift type

- ☐ **Shift taxonomy**: Distinguish covariate shift, label shift, and concept drift by which probability the change touches, and explain why feature-statistics monitoring stays silent under pure concept drift.
- ☐ **Shortcut failure**: Identify the failure in a cow-on-a-beach misclassification where the world has not changed but the model learned the wrong lesson, and explain why standard empirical risk minimization encourages it.

Reasoning about thresholds and cost

- ☐ **Threshold judgment**: Select retraining, investigation, or continued monitoring for a high-importance feature whose population stability index is in the major-shift band while precision and recall stay flat.
- ☐ **Cost asymmetry**: Explain why concept drift is structurally more expensive to remediate than data drift, even when the same volume of fresh data is available.

14.5 Input-Level Attacks and Model Robustness

Adding a microscopic, mathematically calculated layer of noise to an image of a benign skin lesion, noise so subtle a human dermatologist cannot see it, causes a production diagnostic model to diagnose it as malignant with 99.9 percent confidence. The high-dimensional decision boundaries learned by deep neural networks possess counterintuitive blind spots that malicious actors can deliberately exploit. The practical question is what the attacker can see or control: gradients, queries, physical sensors, or training data. Adversarial attacks expose these blind spots.

14.5.1 Adversarial attacks

Definition 14.3: Adversarial attack

Adversarial Attack is a deliberate, mathematically crafted perturbation to model inputs designed to cause misclassification while remaining imperceptible to humans.

1. **Significance**: It reveals that high-dimensional decision boundaries have counterintuitive vulnerabilities. The per-feature perturbation magnitude required for misclassification scales inversely with input dimensionality, meaning models operating on high-dimensional inputs (for example, high-resolution images) are susceptible to attacks in which no single feature changes perceptibly.
2. **Distinction**: Unlike random noise (which the model can learn to ignore), adversarial perturbations are gradient-directed: they are specifically optimized to maximize the model's prediction error.
3. **Common pitfall**: A frequent misconception is that adversarial vulnerability is a "bug" to be patched. In reality, it is a structural vulnerability of many standard neural networks trained by empirical risk minimization; robust defense typically requires fundamental changes to the objective function (for example, adversarial training).

Adversarial attack categories encode access and cost. A white-box attacker can use gradients directly, a black-box attacker relies on transfer, and a physical attacker must survive cameras, lighting, and distance. Figure 14.12 demonstrates the shared mechanism: small, carefully designed perturbations to input data can cause high-confidence misclassification, with perturbations invisible to the human eye but devastating to model accuracy.

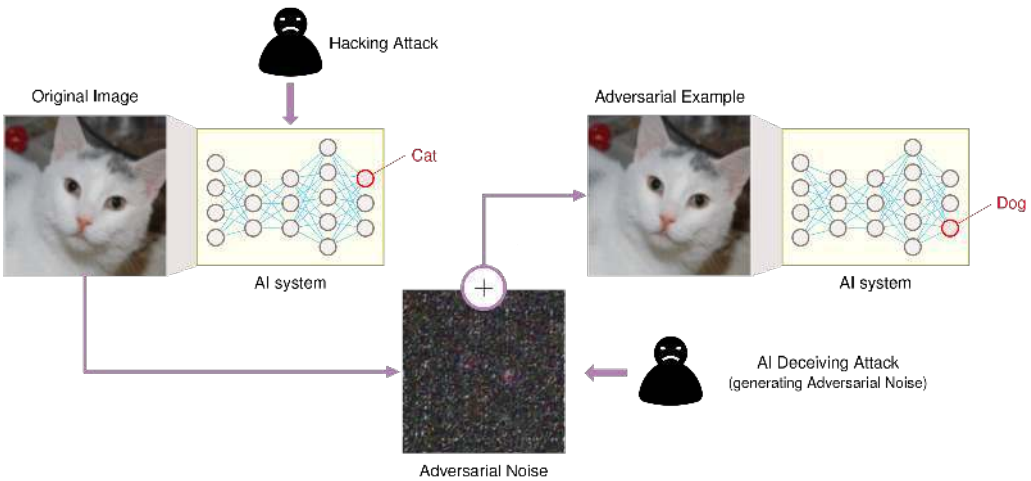


Figure 14.12: Adversarial Perturbation: Subtle, intentionally crafted noise can cause neural networks to misclassify images with high confidence and expose vulnerabilities in model robustness. These perturbations, imperceptible to humans, alter the input in ways that maximize prediction error and highlight the need for defenses against adversarial attacks. Source: Sutanto (2019).

The effectiveness of these attacks traces to a fundamental mismatch between human and machine perception[16]. Neural networks draw nonlinear decision boundaries through a high-dimensional feature space, and adversarial perturbations exploit the geometry of those boundaries: the many input dimensions give an attacker many directions to push simultaneously, so a change too small to see in any one dimension can still cross the boundary.

16 **Human vs. Machine Perception**: First highlighted by Szegedy et al. (2013), neural networks learn statistical correlations in pixel space rather than the semantic invariances human vision enforces. This gap is not a bug to be patched but a structural consequence of gradient-based optimization on finite training data: the model finds decision boundaries that minimize empirical risk, and adversarial perturbations exploit the vast regions of input space those boundaries leave unguarded.

14.5.1.1 Attack categories and mechanisms

The useful axis is not the attack name by itself but the attacker's access. Each mechanism reveals a different cost that the defense must raise: gradient access, optimization time, surrogate-model construction, or physical control over the sensor environment.

The most direct case is white-box gradient access. Neural networks compute gradients to learn how parameter changes reduce loss; an attacker with access to gradients can run that logic against the input instead. For an image classifier that correctly identifies a cat, the gradient with respect to the input image reveals which pixel-level changes would most increase prediction error. The **Fast Gradient Sign Method**[17] turns that idea into a single-step attack by moving each input feature in the direction that increases loss fastest.

The underlying mathematical formulation captures this intuitive process:

$$x_{\text{adv}} = x + \epsilon \cdot \text{sign}(\nabla_x \mathcal{L}(\theta, x, y))$$

where:

- x is the original input
- x_{adv} is the adversarial example
- $\mathcal{L}(\theta, x, y)$ is the prediction loss
- $\nabla_x \mathcal{L}$ identifies the input changes that most increase that loss
- $\text{sign}(\cdot)$ keeps only the direction of steepest ascent
- ϵ controls how much perturbation the attacker is allowed to add

Figure 14.13 visualizes how this approach generates adversarial examples by taking a single step in the direction that increases the loss most rapidly, moving the input across the decision boundary with minimal perturbation.

17 **Fast Gradient Sign Method (FGSM)**: Proposed by Goodfellow et al. (2014) at ICLR, FGSM generates adversarial examples in a single gradient step, making it practical to use during training. This dual role is its lasting systems significance: the same attack that exposed neural network fragility became a simple adversarial-training primitive, where FGSM-generated perturbations augment clean batches with examples chosen to increase loss.

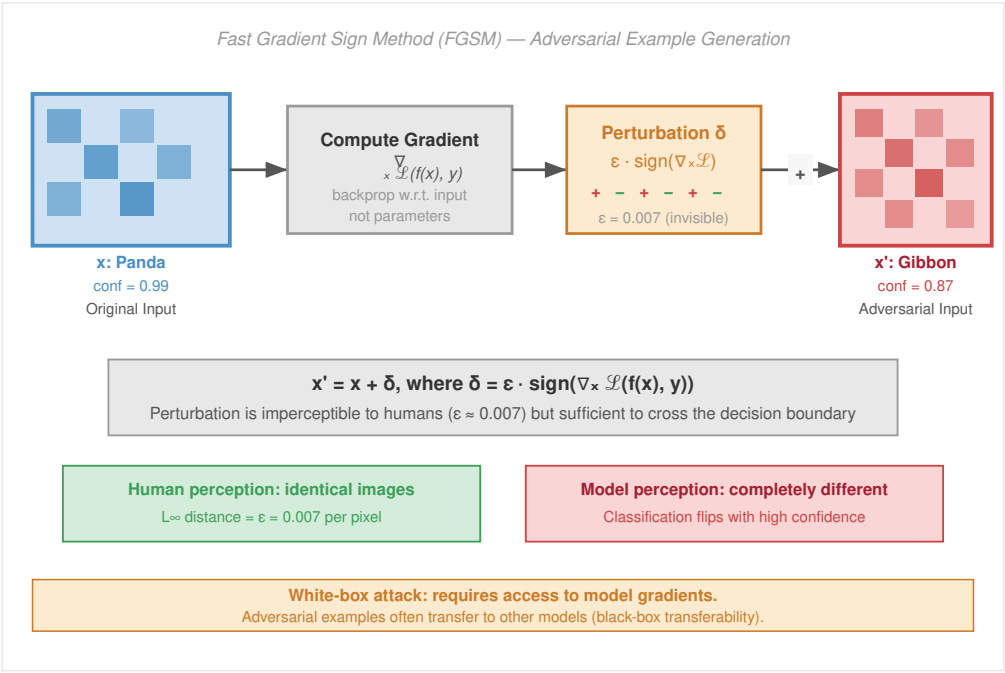


Figure 14.13: Adversarial Perturbations: Gradient-based attacks generate subtle, intentionally crafted input noise with magnitude controlled by ϵ that maximizes the loss function $\mathcal{L}(\theta, x, y)$ and causes misclassification by the model. These perturbations, imperceptible to humans, exploit model vulnerabilities by moving the input x across the decision boundary. Source: adapted from a gradient-based-attacks tutorial at defence.AI.

FGSM is cheap because it takes one step. The **Projected Gradient Descent (PGD) attack** (Madry et al. 2018) spends more compute for a stronger attack: it repeatedly applies gradient updates and projects each step back into the allowed norm ball around the original input. That iterative refinement makes PGD a standard white-box robustness benchmark. The **Jacobian-based Saliency Map Attack (JSMA)** (N. Papernot et al. 2016) uses the Jacobian to identify the most influential input features and perturb a smaller set of dimensions toward a target class. These methods are most effective in white-box settings[18], where the attacker knows the model architecture and gradients.

When the attacker cares less about speed and more about stealth, the attack becomes an optimization problem. The Carlini and Wagner (C&W) attack[19] (Carlini and Wagner 2017) searches for the smallest perturbation that causes misclassification while preserving perceptual similarity to the original input. Instead of merely following the sign of a gradient, C&W optimizes a custom objective that trades perturbation size against confidence in the wrong answer.

C&W attacks are difficult to detect because the perturbations are usually imperceptible to humans and can be optimized under different norm constraints, such as ℓ_2 or ℓ_∞. The **Elastic Net Attack to DNNs (EAD)** adds elastic net regularization, combining ℓ_1 and ℓ_2 penalties to generate sparse, localized perturbations. These optimization-based methods are more computationally intensive than gradient-sign attacks, but they give the attacker finer control over the adversarial example's geometry.

Black-box attackers lose direct gradient access, but they can still exploit transferability[20]. **Transferability** is the phenomenon in which adversarial examples crafted for one model can fool other models, even when the architectures or training datasets differ. An attacker can train or obtain a surrogate model, craft attacks offline, and then submit the resulting examples to the target API without ever seeing its weights or gradients.

Transfer success depends on model similarity, training-data overlap, and regularization. Attackers can improve transfer by using input diversity, such as random resizing or cropping, and momentum during optimization. This threat model is especially relevant for commercial APIs, where the attacker can observe inputs and outputs but not internal computation.

Physical-world attackers face the hardest constraint: the perturbation must survive sensors, distance, lighting, viewing angle, and ordinary deployment variation. Adversarial patches are printed patterns placed on objects so that cameras and detectors misread the scene. Modified road signs, clothing patches, and 3D-printed objects move the attack from the digital input tensor into the physical environment. That makes the robustness question operational rather than merely mathematical: a defense that works on a saved image may fail once the attack passes through a camera lens, compression, motion blur, and changing illumination. These threats matter most for AI systems deployed in physical spaces, such as autonomous vehicles, drones, and surveillance systems, where a model error becomes a safety, security, and accountability problem in the world.

Example 14.1: The stop sign attack

Context: The traffic-sign attack that Chapter 13 analyzes as a security threat is, from the robustness side, a physical adversarial example: black and white stickers fooled an object detector into reading a stop sign as a speed limit sign, with misclassification surviving distance (up to 30 feet), viewing angle, and lighting variation (Eykholt et al. 2018).

Systems lesson: The robustness obligation that case raises is mechanism, not classification. High-confidence predictions can be manipulated by physically realizable changes that are obvious to humans but catastrophic for the model's feature extractors, so robustness must be evaluated against the deployed sensor loop, not just against digital test images.

The point of table 14.6 is defense selection, not vocabulary. A model that fails a white-box PGD test has no credible worst-case robustness claim; an API-facing model must budget for surrogate transfer and probing; a safety-critical vision system must test the full sensor loop rather than only saved images. The rows therefore identify what the attacker uses to cross the decision boundary: gradients, optimization, surrogate transfer, or physical sensor manipulation.

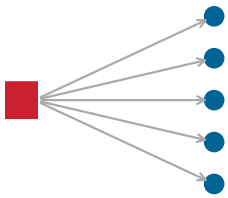
18 **White-Box Attack**: Adversarial attack with complete model knowledge (architecture, weights, gradients). Methods like PGD and Carlini-Wagner (C&W) are strong because they optimize directly against the model rather than probing it from the outside. Though less realistic than black-box scenarios for many deployed systems, white-box analysis establishes a demanding benchmark within a specified norm, perturbation budget, and attack procedure. Passing PGD is evidence for that threat model, not a universal guarantee against every weaker-looking attack.

19 **Carlini and Wagner (C&W) Attack**: Proposed in 2016 (IEEE S&P 2017), C&W formulates adversarial example generation as a constrained optimization problem that finds the minimal perturbation causing misclassification. Its significance is methodological: C&W broke defensive distillation (an early defense that masked gradients to blunt weaker attacks) and several other defenses that resisted FGSM and PGD, establishing the principle that robustness claims must be evaluated against optimization-based attacks, not just gradient-sign heuristics.

20 **Transferability**: The property, analyzed in Nicolas Papernot, McDaniel, and Goodfellow (2016), that adversarial examples crafted for one model can fool different architectures. This transforms the threat model for deployed ML systems: attackers need not see the target model's weights—they can train a substitute locally, craft attacks offline, and transfer them to production APIs. Ensemble adversarial training trains against perturbations from multiple models to improve transfer robustness, but the extra attack generation and model diversity make it a deliberate training-budget decision rather than a free mitigation (Tramèr et al. 2017).

Table 14.6: Adversarial Attack Categories: Attack mechanisms determine defense selection: gradient and optimization attacks require robustness against explicit decision-boundary crossing, transfer attacks test black-box exposure, and physical attacks require evaluation through the deployed sensor loop.

Category	Method	Mechanism
Gradient	FGSM	Perturbs inputs along the loss gradient
	PGD	Iterative multi-step FGSM refinement
	JSMA	Targets the most influential features
Optimization	C&W	Minimizes perturbation size subject to misclassification
	DeepFool	Finds minimal perturbation to cross the decision boundary
	EAD	Elastic net regularization for sparse perturbations
Transfer	Transferability	Adversarial examples transfer across models (black-box)
Physical	Patches	Printed patches fool detectors in the real world
	3D Objects	Sculpted objects deceive sensors in deployment



One surrogate-crafted adversarial example transfers across many models.

The defense cannot be chosen generically. Adversarial training raises the cost of gradient-based attacks, input transformation disrupts some small perturbations before inference, ensembles make transfer less reliable, and physical evaluation exposes failures that digital tests miss. The reason this defense budget matters is that adversarial attacks extend far beyond the basic misclassification that figure 14.14 illustrates, where an imperceptible perturbation makes GoogLeNet relabel a panda as a gibbon on an otherwise unchanged image. That single-image failure is only the entry point; the same principle scales into physical, transferable, and systemic attacks that create risks across deployment domains.

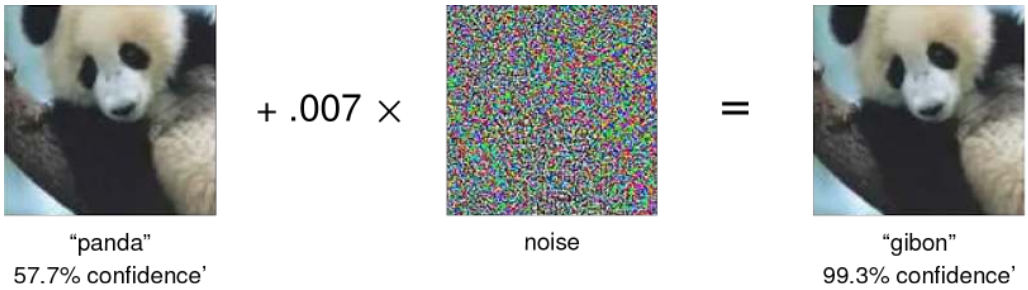


Figure 14.14: Adversarial Perturbations: Subtle, intentionally crafted noise added to an image can cause a trained deep neural network (GoogLeNet) to misclassify it, even though the perturbed image remains visually indistinguishable to humans. This vulnerability underscores the lack of robustness in many machine learning models and motivates research into adversarial training and defense mechanisms. Source: (Goodfellow et al. 2014).

The physical sticker attack on stop signs (the canonical telling is the case study in Chapter 13) misclassified stop signs as speed limit signs over 85 percent of the time, with the perturbation legible to humans yet decisive for the classifier. The implication for autonomous vehicles is direct: stickers deployed on actual roads could cause a self-driving car to misread a stop sign as a speed limit, leading to rolling stops or unintended acceleration into intersections (figure 14.15).

Beyond performance degradation, adversarial vulnerabilities create cascading systemic risks. In healthcare, attacks on medical imaging could enable misdiagnosis (Tsai et al. 2023). Financial systems face analogous manipulation risk when sentiment, fraud, or trading models rely on brittle input patterns. Adversarial vulnerabilities undermine model trustworthiness by exposing reliance on features that are predictive on the training distribution but unstable under crafted perturbation (Goodfellow et al. 2014; Madry et al. 2018). Every defense against them, in turn, charges its own bill: adversarial training inflates training cost (Bai et al. 2021) and runtime detection such as feature squeezing (Xu et al. 2018) adds inference-time evaluations, so the defense itself becomes a budgeted line item rather than a free safeguard. Adversarial vulnerability therefore highlights the urgent need for the defense strategies examined in section 14.6.



Figure 14.15: Adversarial Perturbation (Conspicuous Variant): Physically realizable modifications can cause machine learning models to make incorrect predictions. The image shown depicts a high-visibility variant (overt vandalism rather than imperceptible perturbation) to make the perturbation legible in print; in the literature, the more concerning attack class uses small, inconspicuous stickers that look like ordinary wear yet trigger the same misclassification — see (Eykholt et al. 2017).

14.5.2 Data poisoning

The attacks so far perturb a fixed model at inference time. Poisoning instead corrupts the model itself by reaching back into the data it learns from, and Microsoft's Tay chatbot is the canonical illustration. Tay was an online learning loop in which adversarial users shaped the system's future behavior rather than a norm-bounded image perturbation. Within 24 hours of launch, coordinated users manipulated its learning mechanisms to generate inappropriate and offensive content. The system lacked content filtering, user input validation, and behavioral monitoring, any one of which could have detected and prevented the exploitation. Systems that learn from user interactions require input validation, content filtering, and continuous behavioral monitoring as baseline safeguards.

inject

learn

trigger

Data poisoning runs a lifecycle: inject, learn, trigger.

Definition 14.4: Data poisoning

Data Poisoning is the corruption of training data to compromise model behavior at inference time, either by injecting malicious samples or modifying existing labels.

1. **Significance**: It undermines the foundational assumption of data integrity. Even a small fraction of poisoned samples (for example, <1 percent) can create backdoors or systematic biases that remain latent until triggered by specific inputs during serving.
2. **Distinction**: Unlike adversarial attacks (which occur at inference time), data poisoning occurs during data collection or training, contaminating the model's learned mapping from the source.
3. **Common pitfall**: A frequent misconception is that poisoning can be "fixed" by more data. In reality, poisoning often exploits the aggregation property of training: adding more clean data may not "wash out" a carefully targeted backdoor that uses a unique trigger.

Data poisoning targets the training data itself, contaminating the model's learned mapping before deployment begins. The distinction from adversarial attacks is fundamental: adversarial perturbations fool a trained model at inference time, but poisoning teaches the model wrong patterns from the start. As ML systems increasingly ingest data from automated pipelines, web scraping, and crowdsourced annotation, poisoning becomes a pipeline integrity problem as much as a model robustness problem: the system must detect corruption before the learner internalizes it.

A classic early formulation in ML security is Biggio et al.'s attack[21] on support-vector machines

[21] **Data Poisoning**: In this classic formulation, the attacker injects malicious training samples to corrupt the learning process itself. Unlike adversarial examples that target inference and can sometimes be mitigated at serving time, poisoning embeds vulnerabilities into model weights during training. At web scale, that changes the systems problem from filtering individual inference requests to preserving data provenance, sanitizing suspicious batches, and auditing training samples before they become model behavior.

(Biggio et al. 2012). More recent poisoning work broadens the target to web-scale and generative-model data pipelines (see also Shan et al. 2023). Poisoning attacks alter existing training samples, introduce malicious examples, or interfere with the data collection pipeline (figure 14.16). The consequences are especially severe in high-stakes domains like healthcare, where even small disruptions to training data can lead to dangerous misdiagnoses (Marulli et al. 2022).



Figure 14.16: Data Poisoning Examples: Mismatched image-text pairs represent a common data poisoning attack where manipulated training data causes models to misclassify inputs. These adversarial examples can compromise model integrity and introduce vulnerabilities in real-world applications.

Data poisoning typically unfolds in three stages. During injection, the attacker introduces poisoned samples into the training dataset—altered versions of existing data or entirely new instances designed to blend in with clean examples. The attacker may target specific classes, insert malicious triggers, or craft outliers intended to distort the decision boundary. During training, the model incorporates these samples and learns spurious or misleading patterns; because the poisoned data is often statistically similar to clean data, the corruption goes unnoticed during standard evaluation. Finally, during deployment, the attacker exploits the compromised model—triggering backdoor misclassifications, degrading overall accuracy, or manipulating predictions in targeted ways that are difficult to trace back to training data.

Poisoning categories differ by what the adversary wants the trained model to do (Oprea et al. 2022). In availability attacks, a substantial portion of the training data is poisoned with the aim of degrading overall model performance. A classic example involves flipping labels, for instance, systematically changing instances with true label $y = 1$ to $y = 0$ in a binary classification task. These attacks render the model unreliable across a wide range of inputs, effectively making it unusable.

In contrast, targeted poisoning attacks aim to compromise only specific classes or instances. Here, the attacker modifies just enough data to cause a small set of inputs to be misclassified, while overall accuracy remains relatively stable. The subtlety of targeted attacks makes them especially hard to detect.

Backdoor poisoning[22] introduces hidden triggers into training data, subtle patterns or features that the model learns to associate with a particular output. When the trigger appears at inference time, the model is manipulated into producing a predetermined response. These attacks are often effective even if the trigger pattern is imperceptible to human observers.

Subpopulation poisoning compromises a specific subset of the data population. While similar in intent to targeted attacks, subpopulation poisoning applies availability-style degradation to a localized group, such as a particular demographic or feature cluster, while leaving the rest of the model's performance intact. The localized nature of these attacks makes them both highly effective and especially dangerous in fairness-sensitive applications.

22 **Backdoor Attack**: Introduced by Gu et al. (2017), backdoor attacks embed hidden triggers in training data that activate malicious behavior at inference when a specific pattern appears. BadNets showed that trigger-based attacks can maintain clean-data accuracy while causing targeted misclassification, so ordinary test accuracy alone is a weak detector. The defense challenge is asymmetric: the attacker needs only a small trigger patch, while the defender must audit the training dataset or inspect high-dimensional activation space for spectral anomalies (Tran et al. 2018).



Lighthouse 14.1: Archetype B (DLRM at scale): fake profile injection

Archetype B (DLRM at Scale), the DLRM workload, is especially exposed to **Fake Profile Injection**. Because recommendation systems often use online learning to adapt to user trends in real time, an adversary can create a network of "sybil" accounts, fake identities controlled by the same attacker, that interact with specific items to artificially boost their popularity or associate them with high-value demographics. This poisoning bypasses traditional firewalls because the malicious interactions look like legitimate user behavior, requiring spectral-signature defenses,

which search activation patterns for suspicious low-rank clusters, or other activation-space methods to detect.

A common thread across all four categories is their subtlety: manipulated samples are typically indistinguishable from clean data, making them difficult to identify through standard validation. Attacks may originate from internal actors with privileged pipeline access or from external adversaries who exploit weak points in data collection, particularly in crowdsourced environments or open data pipelines that lack integrity checks and lineage tracking.

14.5.2.1 Data poisoning attack methods

The four categories above describe an attacker's *objective*; the attack *mechanism* depends on the attacker's access to the system and knowledge of the data pipeline. All of them share a common shape: a poisoned record enters the training pipeline alongside clean data and corrupts the model that the pipeline produces (figure 14.17). The most direct mechanism is label modification, where an attacker selects a subset of training samples and alters their labels, flipping $y = 1$ to $y = 0$ or reassigning categories in multi-class settings. Even small-scale label corruption can shift decision boundaries significantly.

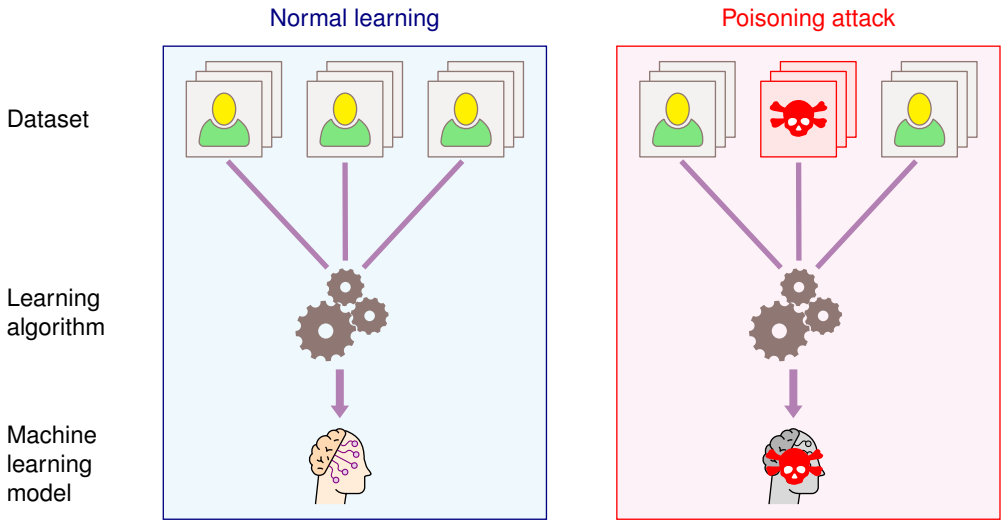


Figure 14.17: Data Poisoning Impact: A single poisoned record injected into the training set corrupts the model the pipeline produces, contrasting a normal learning pipeline (clean data, dataset, learning algorithm, model) against a poisoning attack where one malicious record carries through to the trained model. Even limited adversarial control over training data can disrupt model learning, highlighting the vulnerability of data-driven approaches to malicious manipulation.

After label modification, the attacker can keep the label intact and corrupt the features instead. Imperceptible image perturbations, subtle shifts in structured fields, and fixed trigger patterns all aim for the same outcome: the training example still appears legitimate, but it bends the learned decision boundary toward a future failure. Generative methods and data-synthesis tools raise the same risk at larger scale because they can create natural-looking examples whose only purpose is to distort what the model learns.

Whether those examples become poisoning depends on the data boundary. Web scraping, social media feeds, crowdsourced annotation, and untrusted user submissions all allow poisoned records to enter upstream and pass through weak cleaning checks in a "trusted" form. Physical systems add a second path: a sticker on a road sign is an inference-time adversarial attack when the car sees it, but it becomes training-time poisoning if a fleet-learning pipeline later harvests that image and folds it into the corpus. Online learning systems tighten the feedback loop further, allowing an attacker to introduce small increments of malicious data until model behavior drifts without a single obvious anomaly. Collaborative settings widen the boundary again: when many clients

each contribute model updates to a central aggregator, a single malicious participant can poison the shared global model that aggregation produces (figure 14.18).

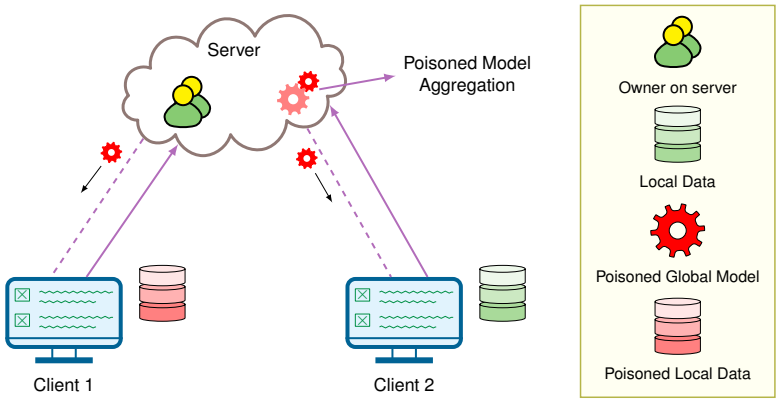


Figure 14.18: Collaborative Poisoning via Aggregation: In a collaborative learning setting, multiple clients each train on local data and contribute model updates to a central server that aggregates them into a shared global model. A single malicious participant can inject poisoned updates that the server folds into the aggregated model, so the corruption spreads to every client without any one obvious anomaly. This attack surface differs from adversarial examples because it targets the model during *training* rather than at *inference*.

Insider collaboration adds a final layer of complexity. Malicious actors with legitimate access to training data, such as annotators, researchers, or data vendors, can craft poisoning strategies that are more targeted and subtle than external attacks because they possess knowledge of the model architecture or training procedures. Whether the result is degraded accuracy, a hidden backdoor, or amplified bias against a demographic subgroup, data poisoning ultimately undermines the trustworthiness of the system itself: a model trained on poisoned data cannot be considered reliable, even if it performs well in benchmark evaluations.

14.5.2.2 Case study: Art protection via poisoning

Data poisoning is not always malicious. Researchers have begun exploring it as a defensive tool, particularly for protecting creative work from unauthorized use by generative AI models.

Nightshade, developed by researchers at the University of Chicago, helps artists prevent their work from being scraped and used to train image generation models without consent (Shan et al. 2023). Nightshade allows artists to apply subtle perturbations to their images before publishing them online. These changes are invisible to human viewers but cause serious degradation in generative models that incorporate them into training.

When Stable Diffusion was trained on just 300 poisoned images, the model began producing bizarre outputs, such as cows when prompted with "car," or cat-like creatures in response to "dog" (figure 14.19). The experiment demonstrates concept poisoning: poisoned samples can distort a model's semantic associations.

What makes Nightshade especially potent is the cascading effect of poisoned concepts. Because generative models rely on semantic relationships between categories, a poisoned "car" can bleed into related concepts like "truck," "bus," or "train," leading to widespread hallucinations. The same technique used to protect artistic content could also be repurposed to sabotage legitimate training pipelines, highlighting the dual-use dilemma[23] at the heart of machine learning security.

Before moving from poisoning mechanisms to defenses, a quick classification check separates targeted poisoning from broader availability attacks.

> Example 14.2: Targeted poisoning classification
>
> **Scenario**: An attacker modifies the labels of a small subset of training data so that one deployed classifier consistently maps a particular protected class or object pattern to the wrong output.

23 **Dual-Use Dilemma**: The structural tension that defensive ML capabilities (adversarial training, data poisoning tools, red-teaming frameworks) are simultaneously offensive capabilities. This creates an arms race where defensive research publications become attack playbooks. For ML systems engineering, the consequence is architectural: robustness mechanisms must assume attacker knowledge of the defense, ruling out security-through-obscurity and requiring formally verifiable guarantees wherever feasible.



Figure 14.19: Concept Poisoning Dose-Response: Stable Diffusion XL outputs across four poison doses (clean, 50, 100, 300 samples) and eight source prompts (dog, car, handbag, hat, fantasy art, cubism, cartoon, concept art). As the dose grows, outputs morph from the intended source concept toward the attacker's destination concept, demonstrating Nightshade-style concept poisoning of a pretrained model.

Diagnosis: This is a targeted attack. The attacker is not trying to degrade overall model performance; they are inducing a specific error while preserving enough general accuracy that aggregate validation metrics may still look healthy.

Systems lesson: Defending against targeted poisoning requires slice-level evaluation, provenance checks, and canary examples for high-risk classes. Aggregate accuracy alone is a weak detector because the attack is designed to hide inside otherwise normal performance.

The mechanics of these input-level attacks, from adversarial perturbations and data poisoning to their interaction with natural distribution shifts (section 14.4), define the threat surface. The next question is *how* to engineer the algorithmic defenses required to protect production models against them.

14.6 Adversarial Defenses

An adversarial defense is a budgeted response to a specific threat model. The stronger the guarantee, the more the system pays in clean accuracy, training compute, inference latency, or operational coverage.

14.6.1 Adversarial defense workflow

The adversarial-defense workflow has four ordered layers:

1. **Define the threat model**: Specify the perturbation budget, attacker knowledge, and whether the attack occurs at inference time or during training.
2. **Choose the robustness budget**: Decide whether to spend the budget in training, certification, input detection, or serving-time guardrails.
3. **Measure the trade-off**: Evaluate both sides of the defense, because improving worst-case behavior can reduce clean accuracy, raise inference latency, or multiply training cost.
4. **Keep evaluation adversarial**: Test the defense against attacks stronger than the ones used to design it.

The order matters because an uncalibrated threat model makes the budget and evaluation loop meaningless.

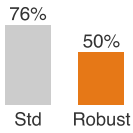
For model-facing perturbation threats, the most direct training-time defense is **adversarial training**, which incorporates adversarial examples into the training process itself.

Systems Perspective 14.1: The robustness tax

Making a ResNet-50 more robust to small norm-bounded image perturbations ($\epsilon = 8/255$) carries a measurable cost in clean ImageNet performance. In the published literature, a standard ResNet-50 reaches 76 percent Top-1 accuracy on ImageNet, while an adversarially trained ResNet-50 ($\epsilon = 8/255$) reaches ~50 percent Top-1 accuracy on clean ImageNet.

Gaining robustness against rare adversarial attacks therefore sacrifices 26 percentage points of clean accuracy on normal inputs. The model must learn to ignore "nonrobust features" (like high-frequency textures) that are predictive but brittle.

Systems insight: Robustness cannot simply be "turned on" for free. It is a fundamental trade-off between *average-case performance* and *worst-case reliability*.



Adversarial robustness costs roughly 26 points of clean accuracy.

The robustness compute penalty (principle 18) quantifies this cost: PGD-style adversarial robustness can demand several times more compute per epoch than standard optimization because each batch runs inner attack steps. For many applications, it is more efficient to rely on external guardrails (input filtering, output verification) than to train intrinsic robustness into the model weights.

Every defense carries an engineering tax in accuracy, in compute, or both. Selecting a defense therefore starts with the threat model and the budget available to absorb that cost.

Example 14.3: Defense selection by threat model

Scenario: Three production systems face different robustness threats and compute budgets.

Setup:

1. **Production image classifier (evasion)**: Use adversarial training or randomized smoothing despite the inference latency cost.
2. **Recommendation system (poisoning)**: Use robust matrix factorization or trim outliers in the training data distribution.
3. **Fraud detection (distribution shift)**: Use continuous monitoring with automated retraining triggers based on KS-test statistics rather than static robustification.

Systems lesson: Defense selection follows the threat model. A robustness technique that is appropriate for evasion may waste compute or miss the failure mode entirely when the real risk is poisoning or drift.

14.6.1.1 Certified defenses

Adversarial training is empirical; it resists specific attacks seen during training but often fails against novel or stronger perturbations. **Certified robustness** offers a mathematical guarantee: for a given input x and radius ϵ, no perturbation $\|\delta\|_p < \epsilon$ exists that changes the model's prediction. A widely used technique for scaling this idea to high-dimensional inputs like ImageNet is **randomized smoothing**. Instead of classifying x directly, we classify the smoothed function $g(x)$, defined as the expected prediction of the base classifier f under Gaussian noise: $g(x) = \arg\max_c \Pr(f(x+\delta) = c)$ where $\delta \sim \mathcal{N}(0, \sigma^2 I)$.

Cohen et al. (2019) proved a tight multiclass bound for the certified radius R using a lower bound on the top-class probability p_A and an upper bound on the runner-up probability p_B: $R = \frac{\sigma}{2}\left(\Phi^{-1}(p_A) - \Phi^{-1}(p_B)\right)$, where Φ^{-1} is the inverse standard normal CDF. In the binary special case, where $p_B = 1 - p_A$, this reduces to $R = \sigma\Phi^{-1}(p_A)$. The bound transforms robustness into a statistical estimation problem. If a binary decision has top-class probability $p_A = 0.999$ under noise $\sigma = 0.5$, the simplified radius is approximately 1.5, but multiclass ImageNet certificates must also account for the runner-up class. However, this guarantee comes at a steep price in both accuracy and compute. On ImageNet, Cohen et al. report nontrivial certified accuracy for randomized smoothing, including

certification at radius 0.5, but certification requires sampling many noise vectors per inference to estimate class probabilities with sufficient confidence. The large increase in inference latency restricts certified defenses to asynchronous auditing or high-stakes safety interlocks, rather than real-time serving paths.

Certified defenses cover one part of the adversarial workflow. Production systems still need detection paths for suspicious inputs, mitigation strategies that change training or serving behavior, and evaluation procedures that verify which threat models remain covered.

14.6.1.2 Detection techniques

Detecting adversarial examples before they reach the model forms the first line of defense only when the signal reaches a serving decision: reject, transform, route, or audit. The cheapest signal asks whether the current input population still resembles the reference population. Statistical tests such as the Kolmogorov-Smirnov test[24] (Berger and Zhou 2014) or the Anderson-Darling test measure distributional discrepancy and can flag inputs that deviate from a known benign baseline.

That signal is weak against attacks that preserve marginal distributions, so production defenses often add a perturbation-destroying check. Feature squeezing[25] (Xu et al. 2018) reduces input-space complexity through dimensionality reduction or discretization, then compares the model's prediction before and after the transformation. A large prediction change is evidence that the original input relied on brittle high-frequency detail.

The most expensive signal asks whether the model is uncertain about its own answer. Adversarial examples often sit near unstable regions of the decision boundary, so uncertainty can route an input for rejection, human review, or a more robust model. Bayesian neural networks estimate uncertainty by treating weights as distributions, while ensemble methods compare predictions from independently trained models and use disagreement as the warning signal (Lakshminarayanan et al. 2017). Both improve detection quality but spend extra inference compute, which makes them easier to justify for batch scoring or safety interlocks than for every low-latency request.

Dropout[26], originally designed as a regularization technique to prevent overfitting during training (Hinton et al. 2012), randomly deactivates a fraction of neurons during each training iteration, forcing the network to avoid over-reliance on specific neurons and improving generalization. The same mechanism can be repurposed for uncertainty estimation through **Monte Carlo dropout**[27] at inference time, where multiple forward passes with different dropout masks approximate the uncertainty distribution. The resulting estimates are less precise than Bayesian methods because dropout was designed for regularization, not uncertainty quantification. Hybrid approaches that combine dropout with lightweight ensemble methods or Bayesian approximations balance computational efficiency with estimation quality, making uncertainty-based detection more practical for production deployment.

14.6.1.3 Defense strategies

Once the detection layer flags adversarial inputs, defense strategies mitigate their impact and improve model robustness. The most common strategy is adversarial training: augmenting the training data with adversarial examples so the model learns to classify perturbed inputs correctly. Listing 14.1 implements this pattern using FGSM to generate perturbations on-the-fly and mix clean data with adversarial examples in each training batch. The method improves robustness but imposes significant computational overhead that production systems must manage carefully.

Training time can increase 3–10× because adversarial example generation during each training step requires additional forward and backward passes through the model (Madry et al. 2018; Bai et al. 2021). Memory overhead depends on whether the implementation stores clean and adversarial examples together, recomputes perturbations, or reuses gradient information. Iterative attacks like PGD, which require multiple optimization steps, demand specialized infrastructure for efficient generation; optimized variants reduce overhead by reusing computations rather than treating every attack step as a separate full training pass (Shafahi et al. 2019).

The clean-accuracy cost depends on the threat model and defense. For strong ImageNet-scale adversarial training at $\epsilon = 8/255$, clean accuracy can drop by roughly 26 percentage points, as in the robustness-tax example above. Lighter defenses, smaller perturbation budgets, or randomized smoothing can impose smaller costs, often in the single digits to mid-teens, but the trade-off remains

24 **Kolmogorov-Smirnov (K-S) Test**: Non-parametric test comparing two probability distributions, computationally efficient at $\mathcal{O}(n \log n)$ but limited to univariate distributions. For adversarial detection, K-S tests compare per-feature input distributions against training baselines, flagging deviations at p-values < 0.05. The critical limitation for ML systems: adversarial perturbations often preserve marginal distributions while corrupting joint structure, so K-S tests may miss attacks that MMD or embedding-space metrics would catch.

25 **Feature Squeezing**: Defense proposed by Xu et al. (2018) that reduces input precision (for example, 256 to 16 color levels) or spatial resolution (median filtering) to destroy the fine-grained perturbations adversarial examples depend on. In Xu et al.'s evaluated setting, feature squeezing eliminated many adversarial examples while maintaining high clean accuracy; adaptive attacks require validation before treating the defense as production-ready. The detection mechanism compares predictions on original vs. squeezed inputs: large divergence flags adversarial manipulation at low latency cost.

26 **Dropout**: Regularization introduced by Srivastava et al. (2014) that randomly deactivates a fraction of hidden units during training, discouraging co-adaptation and improving generalization. Its robustness role is indirect rather than a guarantee against neuron or weight failures: the same stochastic masking mechanism can be kept active at inference to approximate Bayesian uncertainty with Monte Carlo dropout (Gal and Ghahramani 2016).

fundamental to the robust optimization objective. Model size often increases with robustness-enhancing architectural modifications such as wider networks or additional normalization layers that improve gradient stability.

Hyperparameter tuning grows significantly more complex when balancing robustness and performance objectives. Validation procedures must evaluate both clean and adversarial performance using multiple attack methods, and deployment infrastructure must support the additional computational requirements, including GPU memory for gradient computation and storage for adversarial example caches.

Listing 14.1: Adversarial Training Pseudocode: Framework-neutral adversarial training using FGSM-style perturbations during training, mixing clean and perturbed data to improve robustness against gradient-based attacks.

```
function adversarial_training_step(model, clean_batch, labels, epsilon):
    logits = model(clean_batch)
    clean_loss = loss(logits, labels)

    input_gradient = gradient(clean_loss, clean_batch)
    perturbation = epsilon * sign(input_gradient)
    adversarial_batch = clip(clean_batch + perturbation, valid_input_range)

    mixed_batch = concatenate(clean_batch, adversarial_batch)
    mixed_labels = concatenate(labels, labels)

    return loss(model(mixed_batch), mixed_labels)
```

The implementation in listing 14.1 generates adversarial examples on-the-fly during training by differentiating the loss with respect to the input, applying the sign function to extract a perturbation direction, and mixing the resulting adversarial examples with clean training data. Clipping preserves the valid input range, while concatenation doubles the effective batch size by combining clean and adversarial examples. This approach requires careful tuning of the perturbation budget ϵ; optimized variants can reduce adversarial-training overhead by reusing gradient computations (Shafahi et al. 2019).

Once adversarial examples are part of the training loop, deployment must coordinate robustness techniques with MLOps pipelines, monitoring strategies, and distributed training infrastructure that synchronizes updates across multiple nodes. The remaining strategies move the spending point rather than eliminating the cost. Defensive distillation (Nicolas Papernot, McDaniel, Wu, et al. 2016) spends it during training by teaching a student model from the teacher's soft labels, which can smooth decision behavior but must still be evaluated against stronger attacks. Input preprocessing spends it at serving time: image denoising, JPEG compression, random resizing, padding, and random transformations try to erase perturbations before the model sees them. Ensembles spend it through redundancy, combining models with different architectures, training data, or preprocessing paths so that a perturbation that fools one member is less likely to fool all of them.

14.6.1.4 Evaluation and testing

Evaluating adversarial defenses closes the budget loop: the system must measure which attacks the defense actually covers and what performance it sacrifices under controlled attack conditions. Robustness metrics quantify resilience through accuracy on adversarial examples, the average distortion required to fool the model, and performance under different attack strengths, allowing practitioners to compare models or defenses on common terms. Standardized benchmarks such as MNIST-C (Mu and Gilmer 2019), CIFAR-10-C, and ImageNet-C (Hendrycks and Dietterich 2019) provide corrupted or perturbed versions of the original datasets for measuring robustness to common corruptions, but no benchmark closes the problem. Robustness remains an active area requiring multi-layered approaches that combine detection, defense, and regular testing against evolving threats.

While adversarial training and certified defenses provide a strong perimeter against attacks on the model's inputs at inference time, they rely on a dangerous assumption: that the model itself was trained on trustworthy data. If an adversary compromises the data pipeline long before the model

27 **Monte Carlo Dropout**: Proposed by Gal and Ghahramani (2016), MC dropout reinterprets dropout as approximate Bayesian inference by keeping dropout active at inference and running 10–100 stochastic forward passes. The variance across predictions provides an uncertainty estimate with no architectural changes. The systems trade-off is latency: 50 forward passes at 2 ms each adds 100 ms per request, making MC dropout suitable for batch scoring or safety interlocks but too slow for real-time serving without careful batching.

is even compiled, inference-time defenses are meaningless. The data poisoning attacks described earlier demand their own class of defenses.

Checkpoint 14.3: Choosing and budgeting an adversarial defense

An adversarial defense is a budgeted response to a specific threat model, and a stronger guarantee always costs clean accuracy, compute, latency, or coverage.

Matching defense to threat

- ☐ **Worst-case evaluation**: Explain why a model that passes a white-box PGD test makes a worst-case robustness claim that evaluation only on saved digital images cannot support.
- ☐ **Threat-model fit**: Match the defense emphasis to an API-facing classifier and a safety-critical road-sign detector, and explain why adversarial training alone does not transfer between them.

Reasoning about the budget

- ☐ **Certified vs. empirical**: Explain why certified randomized smoothing's guarantee forces asynchronous auditing rather than real-time serving-path use.
- ☐ **Defense budget**: Justify input detection over intrinsic model robustness for a low-risk component given the clean-accuracy tax adversarial training imposes.

14.7 Data Poisoning Defenses

Poisoning defenses protect the training supply chain rather than the inference boundary. The defense sequence applies four layers in order:

1. **Provenance and access control**: Establish which sources are allowed to modify data.
2. **Anomaly detection and sanitization**: Catch suspicious records before training.
3. **Robust objectives**: Reduce the influence of any poisons that remain.
4. **Representation learning**: Make the model less dependent on brittle artifacts.

Each layer covers a different point in the data path; no single detector can replace end-to-end controls.

Consider a hedge fund training a sentiment analysis model on financial tweets. If a rival firm coordinates a network of bots to systematically associate the word "growth" with negative sentiment during the training window, the newly deployed trading algorithm will aggressively short stocks on positive earnings reports. As figure 14.20 illustrates, data poisoning attacks target the supply chain of machine learning, manipulating the raw material of intelligence before the model even begins to learn.

14.7.1 Ingress and anomaly controls

Poisoning defense begins at the data boundary, before any statistical detector runs. The pipeline must know which sources are allowed to contribute training data, how their authenticity is verified, and which roles can modify accepted records. Strong governance enforces the principle of least privilege,[28] logs data access and modification events, and gives each accepted batch a source identity. These controls do not prove that every record is clean, but they bound the attack surface and give anomaly findings a traceable origin.

After the source boundary, anomaly detection asks whether a candidate training example belongs to the distribution that the pipeline intended to learn from. The cheapest tests compare each record against the bulk distribution: Z-score filtering, Tukey's method, and Mahalanobis distance flag examples whose feature values sit far from normal ranges. These tests are useful because they are fast enough to run at ingestion, but they catch only poisons that look like statistical outliers in raw feature space. For high-dimensional or multimodal data, raw-feature distances are a weak signal: a poisoned image patch or a subtly wrong caption looks unremarkable when measured pixel by pixel but is a semantic outlier in the dense embedding space of a pretrained vision or language model. Production pipelines therefore increasingly apply Mahalanobis distance and clustering not to raw

28 **Principle of Least Privilege**: Articulated by Saltzer and Schroeder (1975), this principle restricts access rights to the minimum necessary for each component. Applied to ML pipelines: inference containers should not access training data, training jobs should not reach production databases, and models should not have network access beyond required APIs. Violations create the attack surface for data poisoning—if a training pipeline can read from unverified sources, it will eventually ingest adversarial data. The same principle governs the model registry, the artifact store that maps version identifiers to weight files and controls which alias (for example, `staging`, `production`) resolves to which checkpoint. Promotion to the `production` alias should be restricted to a narrow, audited set of automated evaluation services and named release engineers; if a researcher's credentials are compromised, least-privilege access controls prevent that compromise from propagating into a swapped-in backdoored model reaching live traffic.

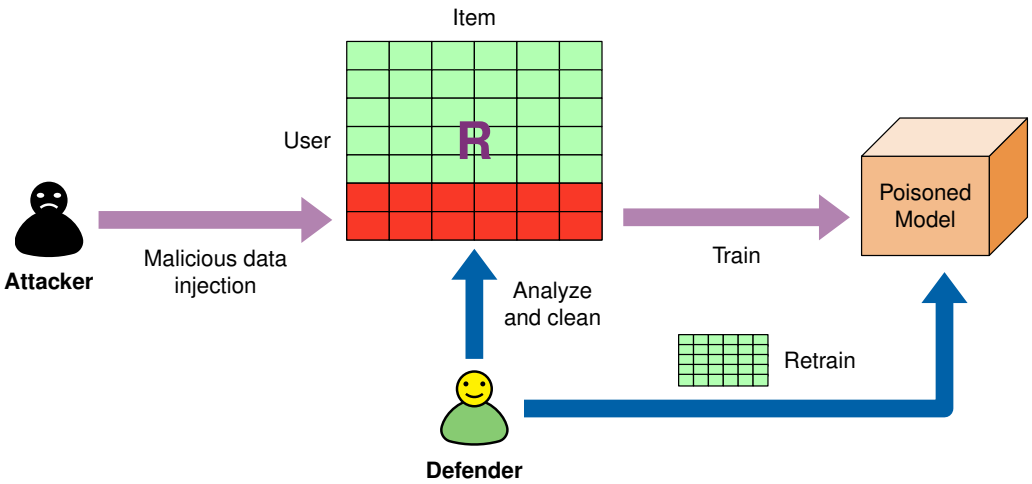


Figure 14.20: Data Poisoning Attack: Adversaries inject malicious data into the training set to manipulate model behavior, potentially causing misclassification or performance degradation during deployment. This attack emphasizes the vulnerability of machine learning systems to compromised data integrity and the need for robust data validation techniques. Source: adapted from a *Mathematics* 12(2):247 illustration.

features but to the latent representations produced by a foundation model encoder, where semantic anomalies that raw statistics miss are often visible as isolated points or low-density clusters far from the class centroids.

More coordinated attacks require structure-aware checks. Clustering methods such as K-means, DBSCAN, and hierarchical clustering look for anomalous groups or isolated points rather than single extreme values. Autoencoders add a learned representation to the same gate: the model reconstructs normal examples well, so high reconstruction error marks a record as abnormal and potentially poisoned (figure 14.21). Each method has a failure mode. Outlier tests miss clean-label poisons, clustering depends on feature representation, and autoencoders can learn the attack pattern if the reference corpus is already contaminated.

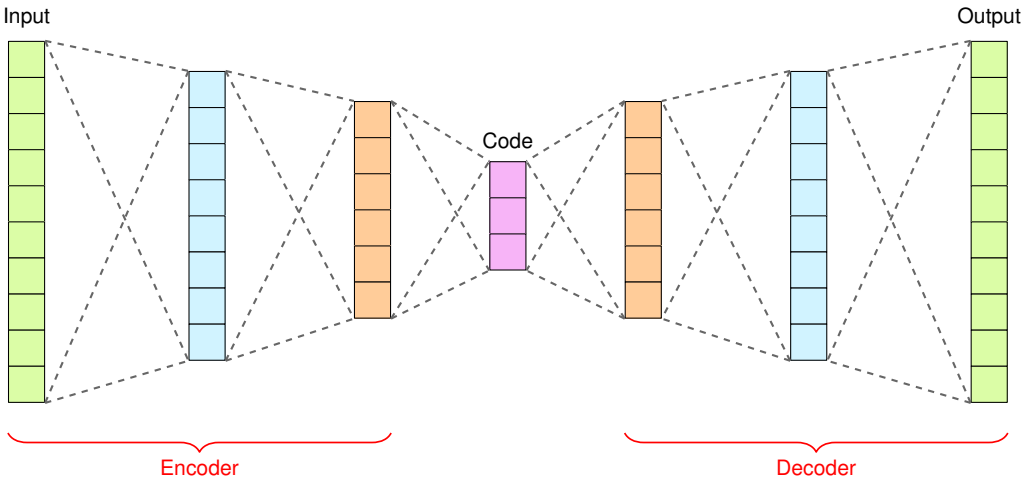


Figure 14.21: Autoencoder Architecture: Autoencoders learn compressed data representations by minimizing reconstruction error, enabling anomaly detection by identifying inputs with high reconstruction loss. During training on normal data, the network learns efficient encoding and decoding, making it sensitive to deviations indicative of potential poisoning attacks. Source: adapted from a *Towards Data Science* autoencoder tutorial.

14.7.1.1 Sanitization and preprocessing

Sanitization turns anomaly signals into training-set changes before the poison reaches optimization. A suspicious record can be rejected, quarantined for review, down-weighted during training, or kept with an explicit provenance flag. Routine cleaning still matters: deduplication, missing-value handling, type checks, range constraints, and cross-field validation remove many low-effort poisoning attempts while improving ordinary data quality.

Data provenance and lineage tracking make those decisions reversible. Each datum needs a record of its source, transformations, validation outcomes, and movement through the pipeline. When a model later exhibits a poisoning symptom, lineage lets the operator trace suspicious behavior back to the source batch, estimate which trained models consumed the compromised examples, and decide whether to remove, relabel, or reweight the affected records.

When suspicious data has already entered the corpus, sanitization moves from source checks to representation checks. **Spectral signatures** (Tran et al. 2018) exploit the observation that backdoor triggers, specific patterns added to inputs to force a target label, introduce a detectable statistical anomaly in the network's internal representations. When activations from a compromised class are analyzed, poisoned samples often align heavily with the top singular vector of the covariance matrix. Projecting samples onto this principal direction and removing outliers can cleanse the dataset without knowing the trigger pattern itself, because the backdoor signal must be strong enough to override natural features and therefore leaves a representation trace.

Influence functions (Koh and Liang 2017) serve a narrower debugging role. They approximate the effect of removing a single training point z on the model's loss for a specific test point z_{test} without retraining. Calculated via the inverse Hessian-vector product, influence $I(z, z_{\text{test}}) \approx -\nabla_\theta \mathcal{L}(z_{\text{test}}, \hat{\theta})^T H_{\hat{\theta}}^{-1} \nabla_\theta \mathcal{L}(z, \hat{\theta})$, this metric identifies which training examples are "responsible" for a specific prediction. If a model misclassifies a stop sign as a speed limit, influence functions can highlight the specific poisoned training images that drove that decision. The limitation is scale: calculating the inverse Hessian H^{-1} is $\mathcal{O}(P^3)$ for P parameters, requiring stochastic approximations like LiSSA (Linear Time Stochastic Second-Order Algorithm) that scale as $\mathcal{O}(nP)$. In deep nonconvex networks, the Hessian is often indefinite, so influence analysis is most useful for gross outliers or labeling errors in the final layer's feature space rather than precise attribution in a large foundation model.

If poisoned samples survive sourcing and sanitization, the training objective becomes the last place to limit their influence. Robust optimization modifies the objective to minimize the impact of outliers or poisoned instances. Robust loss functions such as the Huber loss[29], the Tukey loss (Beaton and Tukey 1974), and the trimmed mean loss down-weight or ignore the contribution of abnormal instances during training. Regularization techniques (ℓ_1 or ℓ_2 regularization) constrain model complexity and reduce sensitivity to poisoned data. At a higher level, robust objective functions such as the minimax[30] or distributionally robust objective optimize the model's performance under worst-case scenarios, providing formal guarantees against adversarial perturbations.

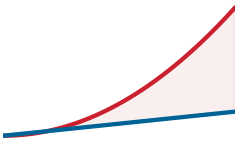
Data augmentation generates additional training examples by applying random transformations or perturbations to existing data (figure 14.22), increasing the diversity and robustness of the training dataset. Controlled variations make the model less sensitive to specific patterns or artifacts that poisoned instances contain. Randomization techniques such as random subsampling or bootstrap aggregating further reduce the impact of poisoned data by training multiple models on different subsets and combining their predictions.

The operating rule is layered: authenticate and govern sources before ingestion, sanitize suspicious data before training,[31] limit surviving outliers during optimization, and preserve lineage so an incident can be traced back to the compromised source. Data poisoning remains an active research area, but production systems should treat the training corpus as a supply chain rather than a passive dataset.

The data boundary is therefore one robustness layer, not the whole defense. Once the training supply chain is governed and suspicious records are filtered, the model still needs representations that survive ordinary deployment variation. Beyond detecting and correcting shifts after they occur (section 14.4), a complementary approach builds shift-resilient representations from the start, drawing on transfer-learning and domain-adaptation foundations (Pan and Yang 2010).

29 **Huber Loss**: This piecewise function transitions from quadratic (MSE) to linear (MAE) at a fixed threshold (typically 1.0–1.5), capping gradient magnitude for extreme samples. In poisoned-data settings, Huber loss prevents a small number of malicious samples from generating outsized gradients that would dominate parameter updates—a property standard MSE lacks, since a single outlier with 100× normal error contributes 10,000× squared-error loss and 100× residual-gradient magnitude.

30 **Minimax**: Game-theoretic strategy from Neumann (1928) that minimizes the maximum possible loss. In adversarial robustness, Madry et al. formulate training as $\min_\theta \max_{\|\delta\| \leq \epsilon} \mathcal{L}(f_\theta(x + \delta), y)$: the model learns to minimize loss under a worst-case perturbation within the chosen norm ball (Madry et al. 2018). The inner maximization is computationally expensive because it is usually approximated with iterative attacks such as PGD, so PGD-adversarial training is a budgeted robustness choice rather than a free objective change (Bai et al. 2021; Shafahi et al. 2019).



Huber loss caps the outlier influence that squared loss amplifies.

31 **Data Sanitization**: Removing sensitive or malicious data from training pipelines. In the poisoning context, sanitization means identifying and removing adversarial training samples before they corrupt model weights. The ML-specific challenge is that poisoned samples are designed to be statistically indistinguishable from clean data, so naive filtering (outlier removal, label verification) misses sophisticated attacks. Effective sanitization requires activation-space analysis (spectral signatures) or influence-function auditing, both computationally expensive at scale.

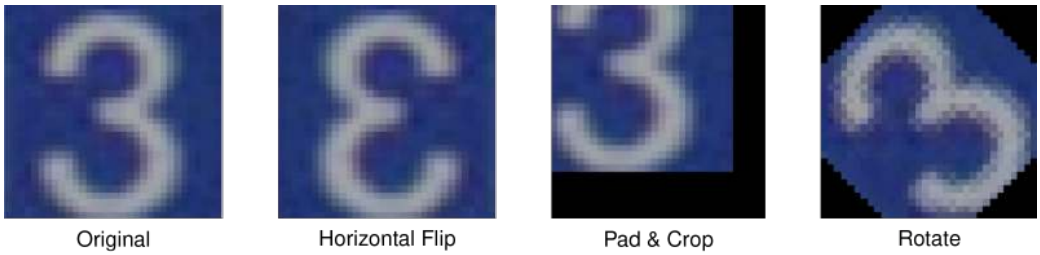


Figure 14.22: Data Augmentation Techniques: Applying transformations like horizontal flips, rotations, and cropping expands training datasets and improves model robustness to variations in input data while reducing overfitting. These techniques generate new training examples without requiring additional labeled data and effectively increase dataset diversity while enhancing generalization performance.

14.7.2 Representation-level defense: Self-supervised learning

Self-supervised learning (SSL) changes the source of supervision. Instead of relying only on task labels, the model learns by solving pretext tasks that require structure in the data itself. This matters for robustness because many brittle models overfit to whichever labeled shortcut is easiest to exploit: a background texture, an annotation artifact, or a narrow phrasing pattern. A pretraining task that rewards stable structure across views, missing tokens, or masked image patches gives the representation a chance to learn signals that survive more deployment variation.

Contrastive learning methods such as SimCLR (T. Chen et al. 2020) make this idea concrete by pushing different views of the same example toward a shared representation. The model is rewarded for treating a crop, color shift, or augmentation as the same underlying object rather than as a new class-specific cue. Masked language modeling in BERT (Devlin et al. 2019) and masked autoencoding in vision (He et al. 2021) use a different route: they hide part of the input and force the model to reconstruct or predict it from context. Both families reduce dependence on a single supervised label signal, which is why SSL representations often transfer better when the deployment distribution differs from the labeled training set.

That transfer benefit is the main systems reason to use SSL in a robustness pipeline. Larger unlabeled corpora expose the model to domains, transformations, and rare cases that would be too expensive to label exhaustively. In production, SSL usually acts as a foundation rather than a complete defense: pretrain on broad unlabeled data, fine-tune on the supervised task, then apply the adversarial, drift, and poisoning defenses from section 14.6 and section 14.7 to the task-specific model. Multi-task training can preserve some of this benefit by keeping a self-supervised objective active while the supervised task pulls the representation toward the deployment metric.

The limitation is that SSL is not a robustness guarantee. A contrastive or masked-pretraining objective can still learn brittle shortcuts, and an attacker who understands the pretext task can target those shortcuts directly. The theory explaining when SSL improves robustness remains incomplete, and the compute cost can be substantial because pretraining moves work earlier in the lifecycle. The systems decision is therefore whether broader representation learning lowers the expected cost of drift, relabeling, and robust fine-tuning enough to justify the added pretraining budget.

14.8 Fallacies and Pitfalls

Robustness spans environmental shifts, input-level attacks, and system-level faults. Each threat domain introduces misconceptions that lead to inadequate defenses or misallocated engineering resources.

Fallacy: *Adversarial examples are an academic curiosity with no real-world impact.*

Published physical-world attacks, such as adversarial patches on clothing or stickers on stop signs, have fooled evaluated vision models under controlled patch and road-sign settings without digital access. Defending against these attacks requires threat-model-specific evaluation, physical-world data augmentation, patch-aware training, and serving-time detection; related PGD-style digital adversarial training commonly increases training cost several-fold, depending on attack steps. Neglecting these defenses leaves open-environment systems vulnerable to failures that ordinary digital test sets do not expose.

Pitfall: *Treating test-set success as robustness proof.*

Standard test sets are drawn from the same i.i.d. distribution as training data and cannot measure resilience to real-world shifts. In production, unmonitored distribution shifts can cause silent performance degradation; reported out-of-distribution evaluations often show large accuracy drops when the deployment population diverges from the training distribution.

Fallacy: *Distribution shift monitoring is optional after deployment.*

Models often degrade silently, maintaining high confidence scores even as predictive performance falls due to drift. Monitoring metrics like the Population Stability Index (PSI) can surface population shifts before accuracy falls below SLA thresholds, enabling proactive intervention when the monitored features are predictive of downstream performance.

Pitfall: *Ignoring poisoning defenses unless the attacker controls the training pipeline.*

Clean-label poisoning attacks compromise models by injecting malicious samples into public datasets or scraped data sources, requiring no access to internal code. The defining property is leverage: a backdoor trigger can be embedded by contaminating a tiny fraction of the training data, well under 1 percent, and remain latent until the trigger appears at inference, so the size of the poisoned slice is no defense.

Fallacy: *Average accuracy is sufficient for robustness measurement.*

High average accuracy often masks fragility: a model with 95 percent accuracy can still be 100 percent vulnerable to targeted perturbations on critical edge cases. Reliable evaluation requires calculating the certified robustness radius or worst-case accuracy under a specific perturbation budget.

Pitfall: *Treating adversarial training as a complete robustness solution.*

Robustness is not universal; it is strictly bound to the specific threat model used during training. A model adversarially trained against ℓ_∞ attacks may offer zero protection against ℓ_2 or geometric attacks, requiring a diverse defense strategy.

Fallacy: *Software faults do not matter when the model is correct.*

Focusing solely on algorithmic robustness ignores the reality that software bugs in data pipelines and serving infrastructure are a major cause of ML failures. Incident analyses often find pipeline and data issues, not model architecture or adversarial attacks alone, at the center of failures. The taxonomy and mitigation of these systems-layer faults is the subject of Chapter 7, and a model hardened against adversarial perturbations remains fragile if a preprocessing bug silently corrupts its inputs.

Pitfall: *Hardening the model while leaving pipeline checks untested.*

Robustness work can concentrate on adversarial training, certified radii, or poisoning defenses while leaving schema validation, preprocessing parity, feature freshness, and rollback drills undertested. That imbalance creates a system that resists one class of attack but fails on ordinary operational faults. A robust deployment validates the model and the pipeline together, because the model only sees the inputs the surrounding system delivers.

These misconceptions share a common root: treating robustness as a single-dimension problem rather than a multi-layered engineering discipline.

14.9 Summary

Robust AI is the "immune system" of the Machine Learning Fleet: it hardens the model and pipeline against failures that can remain invisible under normal service metrics. Reliability is not merely about code correctness; it requires defending against environmental shifts, input-level attacks, and system-level faults, while recognizing that software faults (covered in Chapter 7) can amplify or masquerade as any of the three.

The same arms race appears across empirical defenses like adversarial training and mathematical guarantees like certified robustness. Data poisoning embeds backdoors in the training set, and generative AI introduces "Semantic Reliability" challenges where hallucinations replace traditional classification errors. Integrating spectral filtering, uncertainty quantification, and continuous drift monitoring transforms brittle prototypes into resilient systems.

A model that achieves top benchmark accuracy on a held-out test set can still fail catastrophically in production. The test set, by construction, shares the same distribution as the training data; it cannot reveal how the model behaves when that distribution shifts, when an adversary crafts inputs

designed to exploit geometric vulnerabilities, or when a subtle numerical fault corrupts gradient computations over thousands of training steps. All three fail silently, so the chapter's contribution is the machinery to make them visible and budgeted: distributional distance metrics that surface drift before accuracy falls, worst-case evaluation that exposes adversarial fragility a clean test set hides, the masquerade diagnosis that tells a pipeline fault apart from genuine drift or attack, and the certified radii and uncertainty signals that bound what the model is allowed to assert. By the time a silent failure surfaces through user complaints or downstream metric declines, the damage has already propagated through the system; each of these instruments moves detection earlier.

Building multi-layered defenses against these failure modes transforms a fragile prototype into a production-grade system. Robustness engineering is not a single technique but a discipline that spans the entire model lifecycle: rigorous numerical testing during development, adversarial hardening during training, distribution shift monitoring during deployment, and uncertainty quantification at inference time. Each layer catches failures that the others miss. The cost of this discipline, measured in accuracy trade-offs and additional compute, is real but bounded. The cost of its absence, measured in silent degradation, eroded user trust, and cascading system failures, is far greater and far harder to recover from.

Key Takeaways: Silent failure is the real threat

- **Silence is the failure mode**: Robustness exists because drift, adversarial perturbations, poisoning, and numerical faults often preserve uptime and latency while corrupting predictions. Production systems need monitors that surface competence loss before user complaints or downstream business metrics reveal it.
- **Robustness is bought explicitly**: Strong adversarial training can cost roughly 26 percentage points of clean ImageNet accuracy, while certified defenses and uncertainty sampling add compute. The engineering decision is how much resilience the failure consequence justifies.
- **Drift needs calibrated distance measures**: Statistical distance metrics turn environmental shift into thresholds for review, retraining, rollback, or routing. The metric is useful only when connected to a response path and calibrated against false alarms.
- **Threats masquerade as each other**: A software fault can look like concept drift, a poisoned sample can look like a rare outlier, and an adversarial input can hide inside natural variation. Robust systems combine ingress validation, training defenses, uncertainty signals, and output verification.
- **Generative reliability is semantic**: LLM hallucinations are confidently fluent failures rather than simple label mistakes. Robustness therefore includes grounding checks, self-consistency, entropy or uncertainty signals, and human escalation policies that bound what the model is allowed to assert.

A model robust to every shock would be wonderful, and unaffordable. Robustness is bought, not given: adversarial training can cost tens of points of clean accuracy, certified guarantees and uncertainty sampling multiply inference compute several times over, and drift monitoring runs forever in the background. This is the same trade the whole book turns on, spending one resource to buy a property, now applied to reliability under stress. The engineering question is therefore not whether a system should be robust but how much robustness the cost of failure justifies, and that cost is hard to see, because these threats do their damage silently, before any metric turns red. Pay too little and the system fails without warning; pay too much and it cannot compete on the accuracy and latency that made it worth deploying.

What's Next: From resilience to sustainability

Robustness protects the fleet against distribution drift, input-level manipulation, and system-level faults. Yet a system that survives every failure while consuming a city's worth of power is

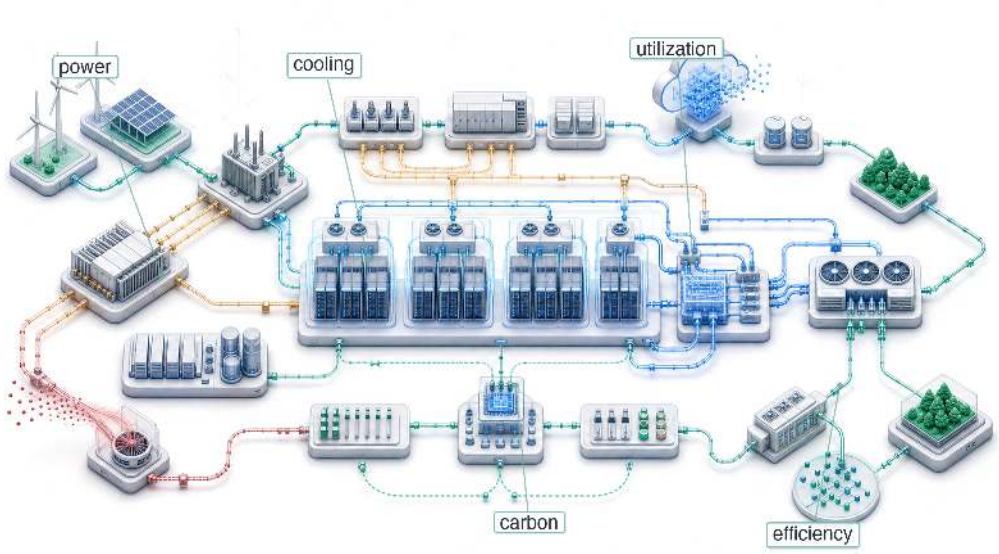
not a viable long-term solution. Sustainability (Chapter 15) turns that reliability question into a resource-accounting problem: lifecycle carbon, carbon-aware scheduling, and algorithmic efficiency determine whether ML systems remain economically and environmentally viable as they scale.

Research Questions: For further inquiry

- How should a production system detect competence loss before silent model failure becomes visible in downstream harm?
- How can operators distinguish true distribution drift from a preprocessing bug, adversarial input stream, or poisoning event?
- When is the robustness tax of adversarial training, certification, or uncertainty estimation justified by failure consequence?
- How should drift thresholds be calibrated so retraining decisions balance false alarms against missed degradation?

15

Sustainable AI

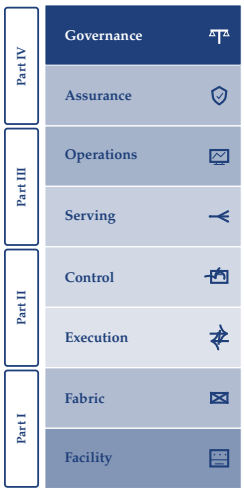


15.1 The Energy Ceiling
15.2 Energy Measurement and Modeling
15.3 Carbon Footprint Calculation
15.4 Data Center Energy and Resource Consumption
15.5 Training vs. Inference Energy Analysis
15.6 Hardware Lifecycle and E-Waste
15.7 Mitigation Strategies
15.8 Policy, Regulation, and the Path Forward
15.9 Fallacies and Pitfalls
15.10 Summary

Purpose

Why does energy consumption determine what machine learning systems can exist, not just what they cost to operate?

Power is not merely an operational expense but a hard physical constraint that limits what can be built. A data center has a fixed power budget determined by its electrical infrastructure and cooling capacity; exceeding that budget is not expensive but impossible. Training runs that require more power than available cannot happen regardless of budget. Deployment locations are constrained by grid capacity and cooling feasibility, not just real estate prices. At the largest scales, the question shifts from affordability to physical feasibility, and the answer depends on energy efficiency as much as algorithmic capability. Organizations that treat energy as a first-class engineering constraint alongside accuracy and latency can build systems that fit within the physical infrastructure available to them. Sustainability is where the thermodynamic limit closes over C^3 co-design: no amount of compute, communication, or coordination can exceed the megawatt capacity of the facility.



Learning Objectives

- Explain why facility power and cooling limits constrain which ML systems can physically exist
- Analyze compute growth, hardware efficiency, and demand rebound to forecast fleet resource demand
- Calculate operational, embodied, and lifecycle carbon from energy use, power usage effectiveness, manufacturing, and grid intensity
- Analyze energy bottlenecks from operations, memory movement, cooling, water, and power delivery constraints
- Compare training, inference, edge, and federated workloads by energy, carbon, and battery budgets
- Design mitigation strategies across algorithms, infrastructure, scheduling, hardware lifetime, and carbon-aware operations
- Evaluate policy, offsets, and environmental justice claims against measurable reductions in fleet emissions

15.1 The Energy Ceiling

A model can be secure against adversaries and robust under distribution shift, yet still fail if the available grid cannot power its accelerators or the facility cannot cool them. **Sustainability** begins at that boundary: the point where accuracy, latency, and reliability are no longer enough because the system must also fit within power, cooling, water, carbon, and hardware lifetime budgets. Energy is the ultimate currency of machine learning, and power density is the ceiling on data-center computational capacity.[1]

When an engineer optimizes a database query to save 100 ms, that is ordinary performance tuning. At fleet scale, the same saving repeated billions of times per day becomes megawatts of electrical power, cooling load, and avoided carbon emissions (Lacoste et al. 2019). A system that exceeds its operating envelope has failed operationally in the same sense as one that crashes, because it cannot be deployed at the intended scale.

That ceiling makes sustainability a design constraint, not a reporting category. The same engineering discipline that budgets memory, bandwidth, and fault tolerance must also budget joules, cooling load, embodied carbon, and hardware replacement cycles.

Contemporary machine learning applications operate at industrial scales, with environmental impact comparable to established heavy industries. Training a single large AI model can consume as much electricity as roughly 120 US homes do in an entire year. When compute demand grows faster than hardware efficiency, the result is the **sustainability paradox** in artificial intelligence (Sevilla et al. 2022). Sustainable AI treats that gap as a first-class systems problem rather than an externality to be managed after deployment.

[1] **Joule**: The SI unit of energy (1 J = 1 Watt-second). To ground the scale of the fleet: a single A100 GPU at peak load consumes ~400 Joules every second. Large-model training runs are measured in billions to trillions of joules, so small per-operation inefficiencies scale into facility-level energy demand.

Definition 15.1: Sustainable AI

Sustainable AI is the systems engineering practice of measuring and optimizing the full environmental cost of ML systems (energy, water, and embodied carbon across training, inference, and hardware manufacturing) and incorporating those costs as explicit constraints in architecture decisions alongside performance and accuracy objectives (Wynsberghe 2021; Lannelongue et al. 2021; Henderson et al. 2020).

1. **Significance**: Training GPT-3 consumed approximately 1,287 MWh of energy (Li 2020), equivalent to roughly 120 US household-years of electricity. The same lifecycle logic applies after training: pretrained models can often be adapted with far less work than training from scratch, and high-volume inference can eventually dominate total energy. Sustainable AI therefore tracks both one-time training runs and recurring serving workloads rather than treating model training as the whole footprint.

2. **Distinction**: Unlike corporate sustainability reporting (which aggregates energy usage into annual CO_2 disclosures), sustainable AI engineering operates at the individual workload level—selecting hardware based on FLOP/s per watt efficiency, scheduling training during periods of high renewable availability, and choosing model architectures that minimize inference FLOPs rather than simply maximizing accuracy.
3. **Common pitfall**: A frequent misconception is that switching to renewable energy solves the sustainability problem. For hardware-intensive ML, embodied carbon (the carbon emitted manufacturing the chips, servers, and cooling equipment before they ever run a training job) often equals or exceeds operational carbon; over 50 percent of an edge device's lifecycle carbon can come from manufacturing, making hardware longevity and utilization rate as important as energy source.

The environmental impact of AI systems spans the complete lifecycle: from semiconductor manufacturing and data center construction to model training, inference deployment, and electronic waste (Wynsberghe 2021; Gupta et al. 2022; Luccioni et al. 2023). Treating this full lifecycle as an engineering problem rather than a corporate responsibility exercise transforms sustainability from a vague objective into a measurable engineering requirement. Before we can optimize this massive footprint, however, we must ground our intuition by calculating the raw physical energy required by a large training run.

Checkpoint 15.1: The energy of intelligence

A 175B parameter model requires approximately 3.14×10^{23} FLOPs to train. Assuming a data center PUE of 1.1 and an end-to-end realized training efficiency of 50 GFLOP/J:

- ☐ **Training energy**: Calculate the total energy consumption in megawatt-hours (MWh).
- ☐ **Household-years**: Convert the training run's energy into household-years using the average US household consumption of 10.7 MWh per year.
- ☐ **Metric boundary**: Assess whether household-years captures the true environmental cost by separating energy consumption (MWh) from carbon intensity, the grams of CO_2 the grid emits per kilowatt-hour delivered (g CO_2/kWh).

That household-year figure is the stake; it is also only the electricity entering the building. The next question is where that energy physically goes: how much reaches the accelerators, how much the cooling and power-delivery overhead consumes, and how the grid behind the meter converts those kilowatts into carbon.

15.1.1 The scale of environmental impact

A training run measured in megawatt-hours is hard to reason about: few engineers carry an intuition for what a megawatt-hour of grid electricity costs the atmosphere. Converting that energy into emissions and then into a known carbon anchor, the CO_2 of a trans-Atlantic flight, turns an abstract number into one with physical stakes.

Napkin Math 15.1: The carbon cost of training

Problem: A team trains a large model (GPT-3 size) consuming 1,287 MWh. How much CO_2 is emitted, and how does that compare to a trans-Atlantic flight?

Math:

1. **Energy**: Training energy consumption is 1,287 MWh = 1,287,000 kWh.
2. **Carbon intensity (US average)**: ≈ 0.429 kg/kWh.
3. **Total Emissions**: Training emissions are 1,287,000 kWh $\times$ 0.429 kg/kWh $\approx$ 552,123 kg.

4. **Comparison**:
   - One passenger, NY to London (round trip): $\approx$ 1000 kg.
   - **Ratio**: 552,123 kg / 1000 kg = 552.1.

Systems insight: A single training run emits as much carbon as hundreds of trans-Atlantic passenger round trips. Optimization matters. Moving this job to a hydro-powered region (0.020 kg/kWh) would reduce emissions by 21.4× to about 25.7 passenger round trips.

That arithmetic shows the carbon cost of one run; at large-cluster scale, the next constraint is whether enough power can be delivered to the cluster at all.

Lighthouse 15.1: Archetype A (GPT-4/Llama-3): The energy wall

Archetype A (GPT-4-class training) exposes the power problem most sharply. A single 25,000-GPU cluster drawing 700 W per chip requires 17.5 MW of accelerator power before server, network, and cooling overhead. The constraint is *physical*, not *financial*: it is a grid capacity problem. Organizations operating Archetype A workloads may need dedicated power infrastructure or regions with excess renewable energy, making carbon-aware scheduling (shifting nonurgent work toward cleaner grid hours) and geographic optimization (placing workloads where power, cooling, and carbon constraints fit the job) critical fleet decisions on the same tier as learning rate tuning.

That accelerator draw alone, before server, network, and cooling overhead, already competes with heavy industry for grid capacity, which makes *where* and *when* a job runs the dominant lever, the subject of the calculations that follow.

A two-region comparison is tractable by hand, but a real fleet weighs many regions against time-varying carbon intensity and regional electricity rates at once, a design space too large to eyeball. At that point the placement decision becomes an optimization problem, and a solver does the search.

Example 15.1: Automated carbon-aware placement

Scenario: A team is planning a large training run requiring 10,000 MWh of energy, choosing between three regions with different electricity prices and carbon intensities.

Problem: With an internal carbon tax of $100/tonne, which region minimizes the true Total Cost of Ownership (TCO)?

Approach: The `PlacementOptimizer` synthesizes grid carbon intensity, regional electricity rates, and the carbon tax into a single optimization objective, then evaluates the design space to select the global minimum.

- **Optimal region**: Carbon-aware scheduling selects Quebec.
- **Total projected cost**: $0.66M (including carbon penalty).

Systems insight: In a pure energy-cost-only model, engineers might choose the region with the lowest raw electricity rate. Once a carbon tax is introduced, the optimization objective internalizes a cost that would otherwise sit outside the engineering budget. The optimizer shows that the hydro-powered grid in Quebec is the most cost-effective choice, because the carbon savings more than offset any marginal difference in electricity pricing.

The scheduling optimizer treats carbon intensity as a time-varying input, shifting workloads to low-carbon hours within a single region. Geographic placement extends the same logic across space: because national grids differ by an order of magnitude in carbon intensity, choosing *where* to run a job can dwarf any gain from choosing *when* to run it. The following calculation isolates this geographic factor by holding energy demand constant and varying only the grid.

Napkin Math 15.2: The geography of carbon

Problem: A team is choosing a data center for a 10,000 MWh training run.

- **Site A (Quebec)**: Hydropower, 20 g/kWh CO_2.
- **Site B (Poland)**: Coal-heavy, 820 g/kWh CO_2. How does the location affect a model's carbon footprint?

Math: Carbon = Energy × Grid Intensity.

1. **Site A emissions**: 10,000,000 kWh × 20 g/kWh = 200,000,000 g = 200 t CO_2.
2. **Site B emissions**: 10,000,000 kWh × 820 g/kWh = 8,200,000,000 g = 8200 t CO_2.
3. **Ratio**: 8200 t / 200 t = 41× difference.

Systems insight: Site selection dominates the modeled levers. The Quebec-versus-Poland pair gives a 41× difference, which exceeds many common algorithmic efficiency gains. Efficiency extends beyond FLOPs to the carbon-intensity of those FLOPs, making carbon-aware scheduling a first-class operational competency in the machine learning fleet.

This Quebec-versus-Poland pair is the canonical anchor for the geographic lever used throughout the chapter. Grid carbon intensity spans roughly ten to eighty times across the world's grids, and representative region pairs land in the eight to forty times range, with the precise figure set by which two grids are compared and whether temporal variation is included. Later sections quote different numbers within this span; each names the assumption that moves it.

Training a single large language model consumes thousands of megawatt-hours of electricity, equivalent to powering hundreds of households for months.[2] IEA projects global data-center electricity consumption to reach about 945 TWh by 2030, just under 3 percent of global electricity demand, with AI-accelerated servers driving much of the growth.[3] Computational demands increased 350,000× from 2012 to 2019 (Schwartz et al. 2020), while hardware efficiency improved at a far slower rate, creating an unsustainable growth trajectory.

Beyond direct energy consumption, AI systems drive environmental impact through hardware manufacturing and resource consumption. Training and inference workloads depend on specialized processors that require rare earth metals whose extraction and processing generate pollution.[4] The growing demand for AI applications accelerates electronic waste production, with global e-waste reaching 54 million metric tons annually (Forti et al. 2020). AI hardware rapidly becomes obsolete due to accelerating performance requirements.[5]

These environmental challenges are part of the energy ceiling, not an add-on to it: the communities that host land, water, grid capacity, and waste streams also carry part of the system cost. The scale calculations above turn sustainability into an allocation problem. AI progress creates benefits in one part of the system while assigning electricity, water, land, and disposal costs to another, so environmental responsibility must be treated as part of systems design rather than as postdeployment reporting.

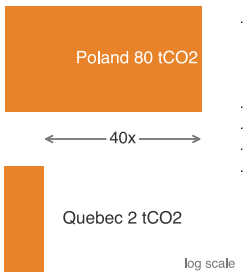
15.1.2 Environmental justice and responsible development

Site selection therefore becomes an environmental-justice decision, not only a facilities decision. Environmental sustainability extends ML systems responsibility from model behavior to ecological stewardship (Vinuesa et al. 2020). The computational resources required for AI development concentrate environmental costs on specific communities while distributing benefits unequally across global populations. Data centers consume on the order of a few percent of global electricity and substantial water for cooling (Andrae and Edler 2015; Jones 2018), often in regions where energy grids rely on fossil fuels and water resources face stress from climate change.

That geographic concentration creates environmental-justice risks that align with broader responsible AI frameworks.[6] The fairness claim here is narrow and systems-specific: an ML fleet allocates benefits, electricity demand, water use, land pressure, and e-waste across different communities. Communities hosting AI infrastructure can bear disproportionate environmental burdens while

2 | **Household Energy Baseline**: The average U.S. household consumes 10.7 MWh annually. GPT-3's verified 1,287 MWh training run equals roughly 120 households' annual electricity, and larger later runs can require substantially more compute. This comparison anchors an otherwise abstract energy figure to physical infrastructure: a single training run can draw more grid capacity than a residential neighborhood.

3 | **Data Center Industrial Scale**: IEA's 2025 *Energy and AI* analysis projects data centers to consume about 945 TWh of electricity by 2030, just under 3 percent of global elec



The same job's carbon emissions depend on where it runs.

footprint; reports 1,312 kg CO_2e for an eight-H100 baseboard, or roughly 164 kg CO_2e per H100 if allocated evenly across the eight GPUs (NVIDIA Corporation 2025). Advanced-node manufacturing also requires substantial ultrapure water, specialty gases, chemicals, and high-temperature process steps. This embodied cost means that in clean-grid regions (hydro, nuclear), manufacturing emissions can rival or exceed operational carbon, making hardware longevity and circular economy reuse critical sustainability levers.

5 | **AI Hardware E-Waste**: Global e-waste reached 53.6 million metric tons in 2019, with computing equipment contributing 15 percent. AI accelerators compound this: 3-5 year obsolescence cycles driven by rapidly advancing architectures mean that a fleet of 10,000 GPUs generates 10–20 metric tons of toxic e-waste per refresh cycle, containing lead, mercury, and cadmium requiring specialized disposal.

having limited access to AI's economic benefits, so site selection becomes part of the engineering design space rather than a facilities afterthought.

15.1.3 Exponential growth vs. physical constraints

Rapid growth in computational demands challenges the long-term sustainability of AI training and deployment. In the 2012–2019 window studied here, reported AI training compute increased 350,000×[7] (Schwartz et al. 2020). Many large-model training regimes since then have continued to favor larger models, larger training datasets, or higher computational budgets. Sustaining that trajectory poses sustainability challenges when hardware efficiency gains fail to keep pace with workload demand.

Historically, computational efficiency improved with advances in semiconductor technology. Moore's Law predicted that the number of transistors on a chip would double approximately every two years, leading to continuous improvements in processing power and energy efficiency.[8] However, advanced process nodes face core physical limits, making further transistor scaling difficult and costly. Dennard scaling, which once ensured that smaller transistors would operate at lower power levels, has also ended, leading to stagnation in energy efficiency improvements per transistor.[9]

When AI models scale faster than the hardware running them improves, the energy budget opens a gap that algorithmic efficiency has to close. As figure 15.1 illustrates for the 2012–2019 compute-growth window, the divergence between computational demand and hardware efficiency creates an unsustainable trajectory in the sensitivity scenario. This technical reality underscores why sustainable AI development requires coordinated action across the entire systems stack, from individual algorithmic choices to infrastructure design and policy frameworks.

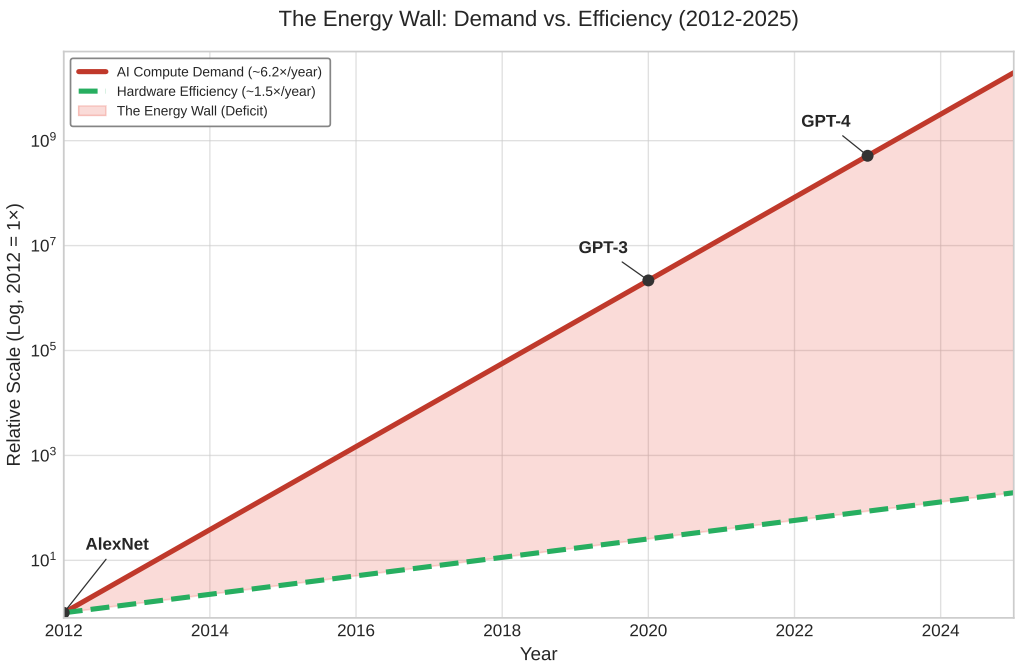


Figure 15.1: The Energy Wall Quantified: The widening gap between AI compute-demand growth (approx. 6.2×/year, matching the 2012–2019 historical claim) and the slower pace of hardware efficiency gains (approx. 1.5×/year) creates a massive energy deficit in the sensitivity scenario.

To make the uncertainty visible, figure 15.2 shows high-growth sensitivity scenarios for data center electricity usage rather than the IEA baseline forecast above. The spread between best, expected, and worst cases illustrates how strongly the outcome depends on efficiency improvements and demand growth assumptions.

6 **Environmental Justice in Data Center Siting**: Data centers gravitate toward low-cost land and electricity, which often means economically disadvantaged areas. The result is an asymmetric externality: communities hosting AI infrastructure bear water depletion, heat island effects, and grid strain, while economic benefits concentrate in distant tech hubs. For ML systems engineers, this creates a design constraint: site selection must factor in social license alongside grid carbon intensity, because community opposition can block or delay facility expansion.

7 **AI Compute Growth Rate**: The 350,000× increase from 2012 to 2019 implies a doubling time of approximately 4.6 months, roughly 5.3× faster than Moore's Law's 2-year doubling. This divergence is one major driver of the energy wall: no physically realizable improvement in silicon efficiency can match a doubling cadence measured in months by process scaling alone, making algorithmic efficiency and carbon-aware scheduling central sustainability levers at scale.

8 **Moore's Law**: Gordon Moore's 1965 observation that transistor density doubles every two years drove decades of "free" efficiency gains for the semiconductor industry. At single-digit-nanometer process nodes, physical limits make further gains harder: individual atoms become part of the constraint. For AI sustainability, the slowdown of process scaling means that additional efficiency gains must come from architectural specialization and algorithmic optimization rather than process shrinks alone.

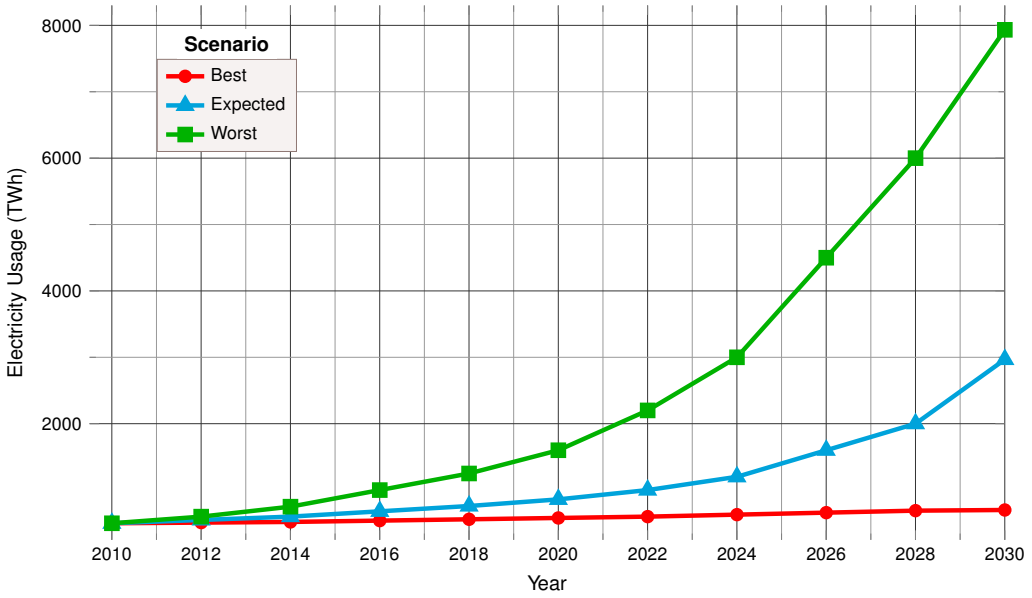


Figure 15.2: High-Growth Data Center Energy Sensitivity Scenarios: Global data center electricity consumption scenarios from 2010 to 2030 under aggressive demand-growth assumptions. The three trajectories—best case, expected, and worst case—diverge significantly after 2018, highlighting the uncertainty and importance of efficiency improvements; they should be read as a sensitivity envelope, not as the IEA baseline projection.

15.1.4 The energy wall: Divergent scaling

Figure 15.1 frames the energy wall as a divergence between compute demand and silicon efficiency, but silicon efficiency is only one ceiling. Even if every accelerator hit its theoretical limit, a second ceiling remains: the physical energy infrastructure of battery density and grid efficiency, which scales far more slowly than compute demand. AI sustainability presents a unique engineering challenge because it is a race between two fundamentally different physics: the *exponential scaling of logic* and the *linear scaling of energy infrastructure*.

As figure 15.3 shows, AI compute grew ~350,000× over the 2012–2019 period cited above while battery density and grid efficiency improve at only ~2–5 percent annually.

While AI logic follows the "iron law" of software optimization, energy follows the laws of chemistry and thermodynamics. Over the same seven-year interval, battery energy density would improve by only ~40.7 percent at a 5 percent annual rate, and grid efficiency by ~14.9 percent at a 2 percent annual rate. The 248,738.5× gap between these curves is the energy wall—the point where we can no longer "buy our way out" of the efficiency problem with more power.

15.1.5 Data center grid dynamics

Sustainable AI requires looking beyond the server rack to the electrical grid interface. Traditional data centers are "Steady-State" loads; they pull constant power 24/7. ML training clusters, however, are transient loads.

War Story 15.1: When the grid became the bottleneck

Context: In July 2022, the Government of Ireland published its *Government Statement on the Role of Data Centres in Ireland's Enterprise Strategy*, formalizing the country's posture toward a sector that by then consumed roughly 18 percent of Ireland's total metered electricity—concentrated almost entirely in the greater Dublin area (Government of Ireland 2022; International Energy Agency 2024a).

[9] **Dennard Scaling**: Robert Dennard observed in 1974 that smaller transistors could operate at constant power density by reducing voltage proportionally. This scaling pattern ended around 2005 when leakage current made further voltage reduction impractical. The consequence for AI sustainability is direct: without Dennard scaling, each process node no longer delivers proportional power savings, which pushes efficiency work toward specialized accelerators—GPUs and Tensor Processing Units (TPUs)—that exploit architectural parallelism rather than transistor physics alone.

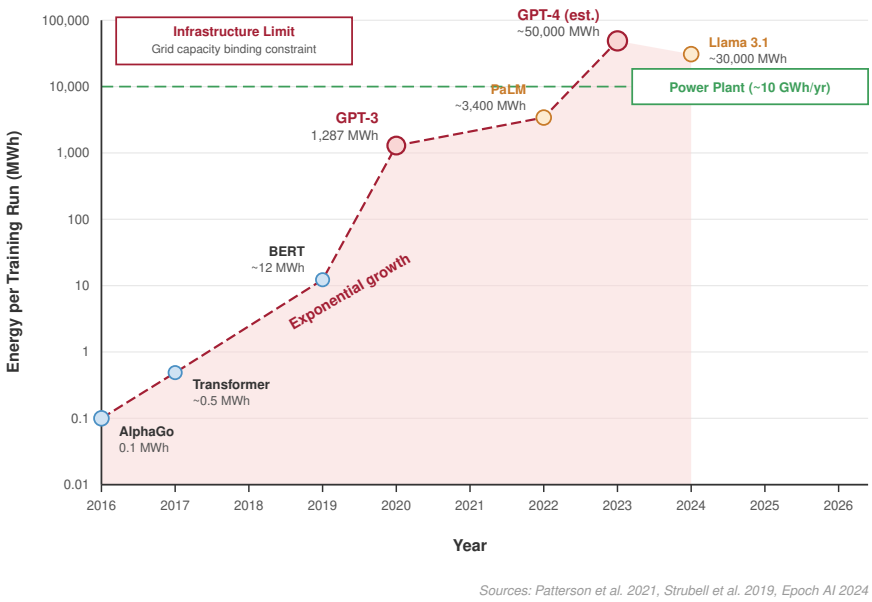


Figure 15.3: The Energy Wall: Training-run energy (MWh) plotted across 2016–2026 against a horizontal power-plant reference line, with milestone models annotated along the trajectory. The curve rises by roughly 350,000× across this window in published estimates while battery density and grid efficiency improve at only 2–5 percent annually, framing the 'energy wall' where algorithmic ambition outruns the physical substrate.

Failure mode: Grid expansion did not keep pace with connection demand. EirGrid and the national regulator reported multiple gigawatts of prospective data-center load queued near Dublin, with the local transmission system pushed to its limit and reinforcement timelines stretching into the next decade.

Consequence: EirGrid effectively imposed a moratorium on new data-center grid connections in the Dublin region through 2028. New connections became a grid-planning and policy decision rather than a normal procurement step, with location, on-site generation, and reinforcement requirements shaping what could be built—and where AI capacity could land at all.

Systems lesson: Sustainable AI is constrained by interconnection, geography, and power-system adequacy. Efficient accelerators help, but the fleet cannot scale faster than the grid that feeds it, and the binding constraint moves from the chip to the substation.

The same grid-interface constraint appears at millisecond scale, with power-delivery hardware providing the mechanism behind it. A 10,000-GPU cluster can swing its load by 5–10 megawatts during an AllReduce synchronization step. For an electrical utility, this is a noise event: when thousands of GPUs suddenly stop computing to wait for the network, they cause a voltage spike on the grid; when they resume, they cause a voltage sag. Managing these transients requires **Energy Buffering**: using on-site battery arrays or massive capacitors to smooth the training iterations, ensuring the ML Fleet does not destabilize the local municipal power grid.

Heat is the paired facility constraint because a data center physically converts high-quality energy (electricity) into low-quality energy (waste heat). A sustainable fleet treats that heat as a recoverable byproduct rather than a pollutant. Modern facilities in Nordic regions, for example, use **district heating** to pipe waste heat into municipal heating systems, while **industrial coupling** can route low-grade waste heat at roughly 45°C into greenhouse climate control or water desalination. These

data-center waste-heat recovery patterns offset nearby thermal demand instead of exhausting all heat into the atmosphere[10] (Ebrahimi et al. 2014).

15.1.6 Training-scale energy concentration

The grid, siting, and heat-reuse constraints become severe because large training campaigns concentrate energy demand into long, synchronized runs. OpenAI's GPT-3[11] exemplifies this scale: its 1,287 MWh training run, the chapter's roughly 120-household-year anchor, reflects the computation required to train large language models on large datasets, with additional energy overhead from distributed-training communication[12] (Maslej et al. 2023).

That concentration makes efficiency improvements an engineering imperative. Large generative models intensify the problem when successive generations increase parameter counts, token budgets, or both.

Within the ranges studied by Kaplan et al. (2020), model scaling laws showed that increasing model size, dataset size, and compute used for training improved performance smoothly. Figure 15.4 demonstrates that test loss decreases predictably across those studied ranges as each of these three factors increases. Beyond training, high-volume deployed systems such as large-scale recommender systems and generative services require continuous inference at scale, consuming energy even after training completes. The cumulative energy burden therefore depends on both the one-time training run and the sustained query volume that follows deployment.

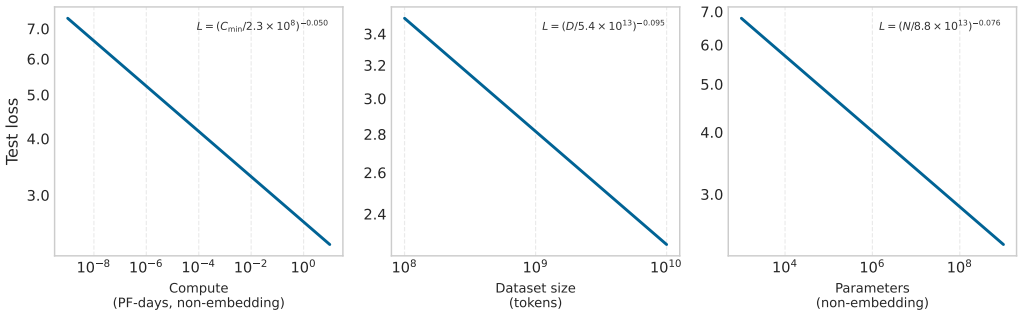


Figure 15.4: Model Scaling Laws: In the ranges studied by Kaplan et al. (2020), increasing model size, dataset size, and compute consistently reduced test loss. The figure reproduces source-paper notation, where N denotes parameter count and C_{min} denotes compute; elsewhere this book uses P for parameter count and O for operation count. These scaling laws explain why larger models, more data, and increased compute became attractive design choices within that regime.

The hardware choice is the first place where energy physics becomes an architecture decision. Different processor types affect environmental impact through their energy characteristics. Using pJ/FLOP as a common comparison point, central processing units consume approximately 100 pJ/FLOP, graphics processing units achieve roughly 10 pJ/FLOP for dense tensor operations, specialized tensor processors reach about 1–2 pJ/FLOP, and fixed-function low-precision accelerators approach 0.1 pJ/operation.[13] These hardware platforms require rare earth metals and complex manufacturing processes with embodied carbon.

The production of AI chips is energy-intensive, involving multiple fabrication steps that the Greenhouse Gas Protocol classifies as Scope 3 value-chain emissions rather than direct electricity use by the operator. As model sizes continue to grow, the demand for AI hardware increases, exacerbating the environmental impact of semiconductor production and disposal.

15.1.7 Theoretical efficiency limits as a sustainability model

To understand the scale of AI's energy challenge, it helps to compare large digital systems with the theoretical limits of computational efficiency. Large language models (LLMs) can operate with an energy efficiency gap of roughly $10^6\times$ compared with highly efficient physical and biological pattern-recognition systems. The comparison is approximate rather than a FLOP-for-synapse equivalence: it says that dense digital models spend enormous energy on always-on arithmetic and global data movement, while sparse physical systems often compute only when signals change. This gap is the headroom the energy wall leaves on the table.

10 **PUE (Power Usage Effectiveness)**: In the early 2000s, PUE values of 2.0-2.5 were common, meaning more power went to cooling than to computing (The Green Grid 2007). Google's 2009 disclosure of PUE 1.21 proved that free-air cooling could halve data center overhead. The shift from PUE to CUE (Carbon Usage Effectiveness) and WUE (Water Usage Effectiveness) reflects a systems-level insight: optimizing watts alone is insufficient when water and carbon constraints bind independently.

11 **GPT-3 Energy Scale**: GPT-3's 1,287 MWh training cost translates to roughly $130,000 in US electricity and 552 metric tons of CO_2 at average grid intensity. The energy-per-parameter ratio of approximately 7.35 MWh per billion parameters reveals the co-design opportunity: optimized architectures using mixed precision and sparsity achieve sub-1 MWh per billion parameters, a several-fold efficiency gain that compounds across large training runs.

12 **Training Communication Overhead**: Distributed training adds 15–30 percent energy overhead beyond raw computation due to gradient synchronization and checkpointing across nodes. For large models requiring thousands of GPUs, this communication tax alone can consume more energy than the entire training run of a mid-scale model, making parallelism strategy selection a first-order sustainability decision.

Training a single model like GPT-3 creates a stark reminder of this gap: silicon-based systems consume megawatts to process trillions of tokens, while sparse, event-driven computation points toward far lower energy per useful operation for some pattern-recognition workloads. This motivates the search for alternative computing paradigms that prioritize energy-aware architecture over raw throughput.

15.1.7.1 Principles of high-efficiency computing

The sustainability lesson from high-efficiency computing is where dense ML wastes energy: continuous activation, data-hungry learning, and global movement. Three principles make those loss channels explicit:

- **Selective, Event-Driven Activation**: Rather than processing all information continuously, high-efficiency systems are asynchronous. They activate only small portions of the network at any time and consume energy only when actively processing changing signals.[14]
- **Local Learning and Sample Efficiency**: Dense language-model scaling often requires training on trillions of tokens to achieve broad competence. High-efficiency models use strong inductive biases and self-supervised local learning to acquire capabilities from 10,000× less data in the motivating biological comparison, reducing the cumulative energy cost of the training phase.
- **Sparsity and Sparse Interconnects**: In accelerator workloads with high data movement and global synchronization, energy is often spent moving operands rather than performing arithmetic. High-efficiency systems use sparse representations where only 1-2 percent of parameters are active for any given task, reducing bandwidth and switching energy by 50–100× when the sparsity maps to hardware-visible work removal.

The biological model points toward promising research directions for sustainable AI. Architectures that implement **Spiking Neural Networks (SNNs)**, event-driven models that communicate through discrete spikes, or sparse activation patterns can achieve significant energy reductions by mimicking sparse communication models[15] (Prakash et al. 2023). Local learning algorithms and self-supervised approaches offer additional pathways toward more sample-efficient and energy-conscious systems.

Achieving sustainable AI requires a systematic shift in system design, moving from continuously active, dense architectures toward event-driven, sparse computation models. As compute demands outpace incremental efficiency improvements in silicon manufacturing, addressing AI's environmental impact demands rethinking the fundamental "Physics" of the algorithm based on these efficiency principles.

Figure 15.5 shows how a six-step energy-gap intervention cascade can reduce the energy gap by approximately 10,000×, transforming an intractable divergence into an engineering challenge. The six per-step factors multiply to a larger figure on paper, but the headline 10,000× is a deliberately conservative composite: overlapping levers do not stack cleanly, since each intervention erodes the savings available to the next. No single lever is sufficient; closing the gap requires simultaneous progress across algorithmic, hardware, and systemic fronts.

The convergence of exponential computational demands with hard physical efficiency limits creates an unsustainable trajectory that threatens the long-term viability of AI scaling. To alter this trajectory, we must move beyond back-of-the-envelope calculations and establish rigorous, systemic frameworks for measuring and assessing energy consumption across the entire ML infrastructure.

15.2 Energy Measurement and Modeling

Engineers cannot optimize what they cannot measure. A cluster consuming five megawatts during a large language model training run directs only a fraction of that power into matrix multiplications; the remainder is consumed by cooling fans removing the resulting heat. Effective energy modeling requires decomposing the monolithic data center power bill into granular, component-level metrics that engineers can target for optimization.

The data center infrastructure foundations from Chapter 2 established power and cooling as dominant engineering constraints. Systematic measurement transforms these constraints into actionable sustainability metrics across three critical areas: energy consumption tracking during training and inference, carbon footprint analysis across system lifecycles, and resource usage assessment for

13 **pJ/FLOP and pJ/MAC**: Energy-efficiency specifications often mix floating-point operations and multiply-accumulate operations. One MAC performs a multiply and an add, so direct comparisons require converting the unit convention and precision. The simplified hierarchy used here aligns with the energy-efficiency comparison later in the chapter: CPUs at roughly 100 pJ/FLOP, GPUs around 10 pJ/FLOP for dense tensor operations, TPUs around 1–2 pJ/FLOP, and custom low-precision ASICs approaching 0.1 pJ/operation. This hierarchy defines the sustainability opportunity: choosing the right hardware tier for a given workload can reduce energy consumption by 100–1,000× without any algorithmic changes.

14 **Event-Driven Computing**: A paradigm where computation triggers only on input changes rather than continuous clock cycles. Neuromorphic chips like Intel's Loihi exploit this to achieve 100–1,000× energy reductions for temporal tasks (audio, video, sensor data) by drawing near-zero power when inputs are static. The trade-off: event-driven architectures sacrifice throughput on batch workloads where all data changes simultaneously.

15 **Spiking Neural Networks (SNNs)**: Third-generation neural networks that communicate through discrete spikes rather than continuous activations. SNNs process information only when spikes occur, achieving 10–100× energy savings on temporal data (audio, video, sensor streams). The sustainability trade-off: SNN training algorithms remain less mature than backpropagation for many benchmarked workloads, but hardware implementations like Intel Loihi 2 demonstrate the efficiency ceiling these architectures can approach.

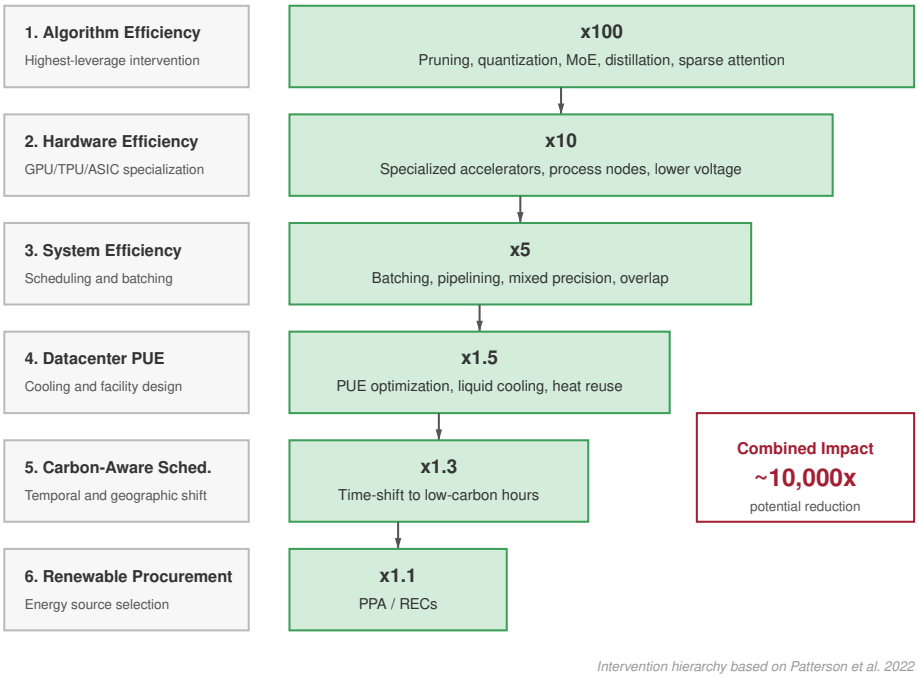


Figure 15.5: Energy Gap Intervention Cascade: Six successive intervention steps progressively reduce the 350,000× energy gap. The combined cascade delivers a conservative composite of roughly $10{,}000\times$, below the naive product of the per-step factors because overlapping levers do not stack cleanly, leaving a tractable residual that engineering can close.

hardware and infrastructure. Just as performance engineering requires profiling before optimization, sustainable AI engineering requires measurement before mitigation.

The decision procedure is the same throughout the chapter: measure the dominant lifecycle term, identify the physical bottleneck behind it, choose the intervention that changes that term, and check whether rebound effects erase the gain. Operational electricity, embodied carbon, cooling overhead, water use, and e-waste each call for different levers. A scheduler cannot fix manufacturing emissions, and hardware longevity cannot fix carbon-intensive runtime placement, so sustainable design begins by locating the term that actually dominates the workload.

15.2.1 Carbon footprint analysis

Carbon footprint analysis turns sustainability from a general obligation into a design constraint. It links energy consumption, grid carbon intensity, and lifecycle resource demands to the same decisions that already govern performance and efficiency. Teams that build and deploy AI systems therefore need a workload-level accounting model before they can choose among larger models, lower-power hardware, cleaner regions, or deferred training.

The accounting model matters because it makes ethical trade-offs auditable. The pursuit of larger models can prioritize accuracy and capability over energy efficiency, increasing carbon emissions when compute growth outpaces efficiency gains. Optimizing for sustainability may introduce engineering trade-offs such as extra tuning effort, hardware constraints, or task-dependent accuracy changes, so the engineering task is to make those costs explicit against environmental benefits. Integrating environmental considerations into AI system design is therefore an engineering obligation, expressed through energy-aware training techniques, low-power hardware designs, and carbon-conscious deployment strategies (Schwartz et al. 2020; Patterson et al. 2021).

Traceability is the technical bridge between sustainability measurement and accountability. Transparency, fairness, and accountability act as sustainability constraints (figure 15.6): transparency

gaps obscure energy and carbon costs, fairness failures distribute harms unevenly, and weak accountability makes resource consumption difficult to trace. In this chapter, accountability means auditability of resource claims. A team should be able to connect a model version, training run, serving workload, region, energy source, and lifecycle estimate well enough that carbon and water claims can be checked.

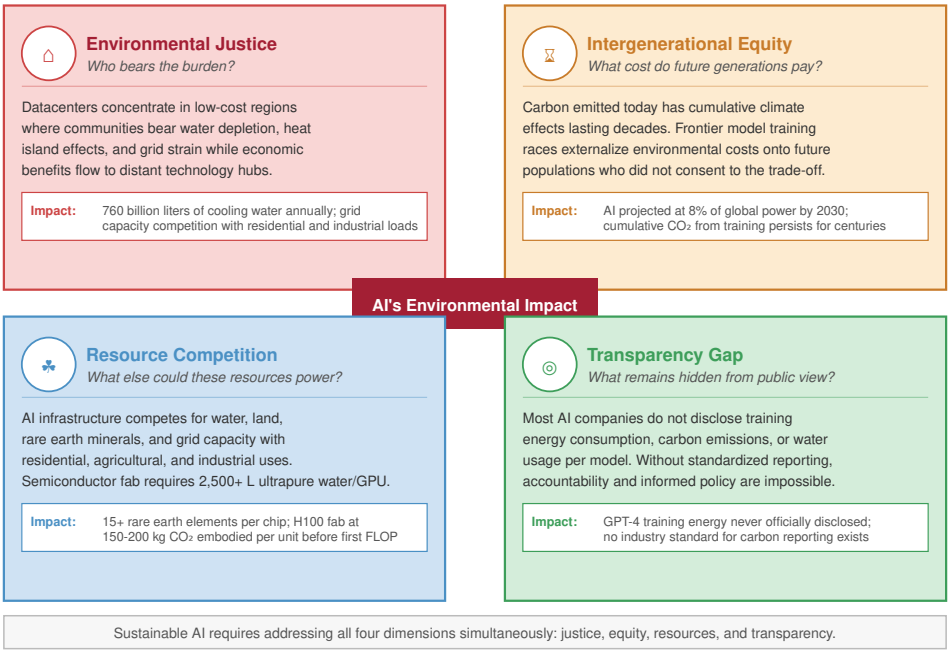


Figure 15.6: Ethical AI Concerns: Four-quadrant layout of environmental-justice and accountability concerns in AI systems, spanning transparency, fairness, and sustainability. The quadrants make explicit how design choices in one quadrant (for example, opacity in training data) propagate into outcomes in an adjacent quadrant (for example, discriminatory predictions or untraceable resource consumption). Addressing these challenges requires proactive design choices that prioritize accountability and minimize negative societal and environmental impacts.

For measurement to constrain design, reported metrics must expose workload-level cost rather than aggregate cloud-scale claims, which stay opaque when reporting holds at the company-total level. The practical standard is therefore accountability for resource usage across the full AI lifecycle, and it requires the same evidence chain used for latency, accuracy, and reliability: a claim should connect to logs, meters, model versions, and deployment decisions rather than to aggregate sustainability pledges alone. The carbon-footprint calculation makes that evidence chain explicit by combining workload energy, facility PUE, grid carbon intensity, and embodied carbon.

Napkin Math 15.3: Lifecycle carbon estimation

Problem: Calculate the total carbon footprint for training a 70B parameter model.

Variables: 2,048 H100 GPUs, 30 days, 700 W TDP, rack-profile support power, PUE 1.12, grid intensity 429 g CO_2/kWh.

Math: Accelerator power = $2{,}048 \times 0.7$ kW ≈ 1433.6 kW. Applying the DGX H100 rack profile for host, memory, networking, and conversion support raises the IT load to 1971.2 kW; facility power = 1971.2 kW $\times$ 1.12 $\approx$ 2,207.7 kW. Energy = 2,207.7 kW $\times$ 24 h/day $\times$ 30 days $\approx$ 1,589,575.7 kWh. Emissions $\approx$ 681.9 t CO_2.

Embodied: Assume manufacturing footprint is ≈ 164 kg CO_2 per H100 GPU, allocating NVIDIA's HGX H100 baseboard product carbon footprint evenly across its eight GPUs (NVIDIA Corporation 2025). Amortized for a 1-month window of a 3-year cycle: (2,048 × 164 kg) / 36 ≈ 9.3 t CO_2.

Result: 681.9 t + 9.3 t ≈ 691.3 t CO_2.

Systems insight: The operational term dominates this training window, but embodied carbon is not zero. Lifecycle accounting prevents teams from hiding manufacturing emissions outside the training budget.

Translating power consumption into carbon emissions is only the first measurement challenge. A systematic lifecycle assessment across the full hardware lifecycle reveals where carbon emissions concentrate and where engineering interventions yield the greatest returns.

15.2.1.1 Three-phase lifecycle assessment framework

The practical question is which lifecycle phase dominates a given workload, because the answer determines the optimization lever. Effective carbon footprint measurement therefore separates three phases that collectively determine environmental impact.

For training-centric research workloads, the training phase often dominates operational emissions because mathematical optimization requires sustained parallel computation[16]. As demonstrated by the GPT-3 case study, large language model training runs exemplify this energy intensity. Geographic placement affects emissions: moving an identical workload between hydro-heavy and coal-heavy grids can create tens-fold differences in carbon intensity.[17]

For high-volume production services, the inference phase can dominate lifetime emissions because model serving repeats continuously after the training run is complete. While individual inferences require less computation than training, the cumulative impact scales with deployment breadth and usage frequency. Models serving millions of users generate ongoing emissions that can exceed training costs over extended deployment periods.

The manufacturing phase contributes **embodied carbon** from hardware production, including semiconductor fabrication, rare earth mining, and supply chain logistics.[18] Its share is smaller for long-running workloads on carbon-intensive grids, but it can reach 30–50 percent of lifetime emissions on clean grids or low-utilization hardware. Often overlooked, this phase represents irreducible baseline emissions independent of operational efficiency.

15.2.1.2 Geographic and temporal optimization

Carbon intensity varies across geographic locations and time periods, creating optimization opportunities. Temporal scheduling can reduce emissions when deadline-tolerant workloads are shifted toward lower-carbon hours or regions, with the realized gain depending on workload flexibility, grid mix, and whether the scheduler uses average or marginal emissions (Patterson, Gonzalez, Le, et al. 2022; Radovanovic et al. 2021). Carbon-aware scheduling systems can automatically shift nonurgent training jobs to regions and times with lower carbon intensity.

Development-time carbon tracking matters when it feeds placement and complexity decisions rather than becoming a report after the fact. Tools such as CarbonTracker (Anthony et al. 2020) and CodeCarbon (Schmidt et al. 2021) wrap the workload boundary, estimate energy from hardware counters or utilization models, attach the local grid intensity, and record the resulting emissions alongside the experiment metadata. The important systems pattern is not the API call but the timing of the measurement: the estimate must appear while model size, training duration, region, and schedule are still adjustable, before the run has already consumed its energy.

15.2.2 Power modeling fundamentals

Understanding where energy goes in AI systems requires grounding in the physics of digital computation. The CMOS power equation provides the foundation for reasoning about energy consumption in digital processors, but the useful model must climb three levels: chip power explains why voltage, precision, and activity matter; optimization techniques show how algorithms change those variables; and facility-level metrics reveal whether chip-level savings survive cooling and power-delivery overhead.

16 **Optimizer Memory as Energy Cost**: Adaptive Moment Estimation (Adam) requires 3× the memory of plain SGD because it stores per-parameter first and second moment estimates alongside the weights themselves. For a 70B model in FP32, this means 840 GB of optimizer state. The sustainability implication is direct: larger optimizer state means more HBM accesses per training step, and at 160 pJ/byte for DRAM, memory overhead can dominate the energy budget of parameter updates.

17 **Carbon Intensity Variance**: Grid carbon intensity spans two orders of magnitude: coal at 820 g CO_2/kWh vs. hydro at 10–30 g CO_2/kWh. Critically, intensity also varies temporally: Texas fluctuates 10× within a single day based on wind generation. This dual geographic and temporal variance is what makes carbon-aware scheduling viable: combining the geographic lever (the 8 to 40 times range from section 15.1.1) with temporal shifting pushes the achievable spread toward the high end of the 10 to 80 times span, so identical training runs can differ several-fold in emissions based solely on when and where they execute.

18 **Embodied Carbon**: The CO_2 emitted during manufacturing, transport, and disposal before a device computes its first FLOP. Allocating NVIDIA's HGX H100 baseboard product carbon footprint evenly across its eight GPUs gives roughly 164 kg CO_2e per H100 (NVIDIA Corporation 2025); at 700 W on the average U.S. grid, continuous operation matches embodied carbon in roughly three to four weeks. As data centers shift to renewables, embodied carbon's share of total lifetime emissions grows, potentially exceeding 30 percent, making hardware refresh cycles a first-order sustainability decision.

15.2.2.1 The CMOS power equation

Every digital circuit consumes power through two fundamental mechanisms. **Dynamic power** arises from switching transistors between states, while **static power** results from leakage current that flows even when transistors are nominally off. Equation 15.1 formalizes the total power consumption:

$$P_{\text{total}} = P_{\text{dynamic}} + P_{\text{static}} = \alpha_{\text{sw}} C V^2 f + V I_{\text{leak}} \tag{15.1}$$

The dynamic power component $P_{\text{dynamic}} = \alpha_{\text{sw}} C V^2 f$ depends on four parameters. The **switching activity factor** α_{sw} represents the fraction of transistors changing state per clock cycle, ranging from 0 to 1. General-purpose CPUs typically exhibit $\alpha_{\text{sw}} \approx 0.1$ to 0.3 due to diverse instruction mixes, while specialized AI accelerators can achieve $\alpha_{\text{sw}} \approx 0.6$ to 0.8 through optimized dataflow that keeps more circuits active during computation. The load capacitance C scales with transistor count and interconnect length. Supply voltage V enters quadratically, making voltage reduction the highest-impact lever for energy efficiency. Clock frequency f determines operations per second.

The static power component $P_{\text{static}} = V \cdot I_{\text{leak}}$ represents leakage current that increases exponentially with temperature, approximately doubling for every 10 degrees Celsius rise. This thermal dependence creates a feedback loop: higher power generates heat, which increases leakage, which generates more heat. Managing this thermal runaway constrains achievable power density and explains why cooling infrastructure represents such a significant fraction of data center energy consumption (Dayarathna et al. 2016).

The practical implications for AI systems follow directly from these physics. The quadratic voltage dependence means that reducing voltage from 1V to 0.8V decreases dynamic power by 36 percent, even before considering that lower voltages often enable frequency reduction with additional linear savings. This relationship explains why specialized AI accelerators operating at lower voltages but higher utilization can achieve order-of-magnitude efficiency improvements over general-purpose processors.

15.2.2.2 Why optimization techniques save energy

The power equation illuminates why specific optimization techniques achieve their efficiency gains. Quantization reduces numerical precision from 32-bit floating point to 8-bit integers, which directly reduces datapath capacitance C by approximately 4 times since narrower datapaths require fewer transistors and shorter interconnects. Additionally, lower precision arithmetic enables reduced supply voltage V because the circuits have larger noise margins. The combined effect yields 6 to 10 times energy reduction per operation, closely matching published measurements of INT8 vs. FP32 inference efficiency.

Pruning removes weights from neural networks, reducing the effective capacitance C by eliminating computation paths that would otherwise consume switching energy. Structured pruning, which removes entire channels or attention heads, achieves larger efficiency gains than unstructured pruning because it eliminates complete circuit paths rather than individual operations that the hardware must still orchestrate.

Specialized accelerators improve the activity factor α_{sw} by designing circuits specifically for matrix multiplication and convolution operations. Where a CPU might activate 10 percent of its transistors during typical ML workloads, a systolic array architecture can keep 70 percent or more of its compute units active, effectively performing more useful work per watt of power consumed.

15.2.2.3 Facility-level power metrics

Beyond chip-level power, data center infrastructure imposes additional energy overhead. Equation 15.2 captures this relationship through the Power Usage Effectiveness (PUE) metric:

$$\text{PUE} = \frac{P_{\text{total facility}}}{P_{\text{IT equipment}}} \tag{15.2}$$

Definition 15.2: Power usage effectiveness (PUE)

Power Usage Effectiveness (PUE) is the data-center efficiency ratio ML system operators use to compare total facility power consumption against the power consumed specifically by IT equipment ($P_{\text{facility}}/P_{\text{IT}}$).

1. **Significance**: It measures the Infrastructure Overhead of the data center. A PUE of 1.0 is the theoretical ideal; a PUE of 1.10 means that for every 100 watts of computation, an additional 10 watts are required for cooling and power distribution.
2. **Distinction**: Unlike Computing Efficiency (which focuses on FLOPs per Watt), PUE focuses on Facility Efficiency: it captures how much energy is "wasted" before it even reaches the processor.
3. **Common pitfall**: A frequent misconception is that a low PUE means a "green" data center. In reality, PUE only measures Efficiency, not the Carbon Intensity of the energy source; a coal-powered data center can have a better PUE than a solar-powered one while having a much higher environmental impact.

Napkin Math 15.4: PUE: The cost of cooling

Problem: A team operates a 2 MW cluster. If the facility can be optimized from the industry average PUE (1.58) to state-of-the-art (1.10), how much energy and money does that save annually?

Math: Energy saved is the difference in infrastructure overhead $(\text{PUE} - 1)$ across the IT load.

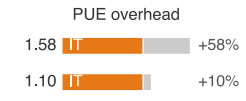
1. **Overhead Reduction**: 1.58 - 1.10 = 0.48.
2. **Annual energy savings**: 2 MW $\times$ 0.48 $\times$ 8760 h/year $\approx$ 8,409.6 MWh.
3. **Financial Savings**: 8,409.6 MWh $\times$ \$70/MWh $\approx$ \$588,672.

Systems insight: Infrastructure optimization is as valuable as algorithmic optimization. Dropping PUE by 0.48 is equivalent to discovering an algorithmic "free lunch" that makes the entire model 30 percent more efficient without changing a single line of training code. For large operators, cooling efficiency is the primary economic lever for sustainability.

A PUE of 1.0 would indicate perfect efficiency where all energy powers computation, though this is physically impossible since cooling, power distribution, and lighting require nonzero energy. Industry-average data centers operate at PUE of 1.5 to 2.0, meaning that 50 percent to 100 percent additional energy beyond computation goes to infrastructure (Uptime Institute 2022). Leading hyperscale facilities achieve PUE between 1.1 and 1.2 through advanced cooling techniques including free-air cooling in cold climates, liquid cooling for high-density GPU clusters, and optimized power distribution.

Equation 15.3 formalizes **Water Usage Effectiveness (WUE)**, capturing the water consumption that evaporative cooling and other processes require:

$$\text{WUE} = \frac{W_{\text{annual\_water\_usage}}}{E_{\text{IT\_equipment}}} \tag{15.3}$$



PUE is the infrastructure energy tax on top of compute.

The units are liters per kilowatt-hour, with typical values ranging from 0.5 to 2.0 L/kWh depending on climate and cooling technology. A data center with WUE of 1.8 L/kWh training a model requiring 10,000 MWh would consume 18 million liters of water, equivalent to roughly 40–50 US household-years of water use under a 380,000–450,000 L/year household baseline.

Facility-level metrics identify where engineering intervention yields the greatest returns. The following case study demonstrates how ML-driven optimization of PUE translates directly into measurable energy savings.

15.2.2.4 Case study: DeepMind energy efficiency

Google's data centers form the backbone of services such as Search, Gmail, and YouTube, handling billions of queries daily (2023). These facilities require substantial electricity consumption, particularly for cooling infrastructure that ensures optimal server performance. Improving data center energy efficiency has long been a priority, but conventional engineering approaches faced diminishing returns due to cooling system complexity and highly dynamic environmental conditions (Buyya et al. 2010). To address these challenges, Google collaborated with DeepMind to develop a machine learning optimization system that automates and enhances energy management at scale.

After more than a decade of efforts to optimize data center design, energy-efficient hardware, and renewable energy integration, DeepMind's AI approach targeted cooling systems, among the most energy-intensive aspects of data centers. Traditional cooling relies on manually set heuristics that account for server heat output, external weather conditions, and architectural constraints. These systems exhibit nonlinear interactions, so simple rule-based optimizations often fail to capture the full complexity of their operations. The result was suboptimal cooling efficiency, leading to unnecessary energy waste.

DeepMind's team trained a neural network model using Google's historical sensor data, which included real-time temperature readings, power consumption levels, cooling pump activity, and other operational parameters. Building on Jim Gao's earlier work demonstrating that machine learning could predict data center PUE with 99.6 percent accuracy (Gao 2014), the model learned the intricate relationships between these factors and could dynamically predict the most efficient cooling configurations. Unlike traditional approaches that relied on human engineers periodically adjusting system settings, the AI model continuously adapted in real time to changing environmental and workload conditions.

The results demonstrated significant efficiency gains. When deployed in live data center environments, DeepMind's AI-driven cooling system reduced cooling energy consumption by 40 percent, leading to an overall 15 percent improvement in PUE[19] (Barroso et al. 2019; Evans and Gao 2016). For a facility operating at the industry-average PUE of 1.5 from equation 15.2, a 15 percent improvement reclaims a substantial fraction of the energy lost to cooling overhead. These improvements were achieved without additional hardware modifications, demonstrating the potential of software-driven optimizations to reduce AI's carbon footprint.

The DeepMind case study illustrates a rare positive feedback loop: machine learning optimizing the infrastructure that powers machine learning. The framework generalizes across facility designs and climate conditions, offering a scalable approach for global data center networks.

19 **PUE Optimization via ML**: Google's best facilities achieve PUE 1.08, meaning only 8 percent energy overhead for cooling and power distribution. DeepMind's reinforcement-learning controller reduced cooling energy by 40 percent by exploiting nonlinear interactions between chillers, pumps, and ambient conditions that rule-based systems miss. This is a rare positive feedback loop where AI improves the efficiency of the infrastructure that powers AI.

15.2.2.5 Carbon intensity and regional variation

The carbon impact of electricity consumption depends critically on the energy generation mix, quantified by carbon intensity measured in grams of CO_2 equivalent per kilowatt-hour (g CO_2eq/kWh). Table 15.1 quantifies carbon intensity by energy source, showing how dramatically these intensities vary:

Table 15.1: Carbon Intensity by Energy Source: Electricity generation carbon intensity varies by more than two orders of magnitude across energy sources. Geographic location of computation can dramatically affect emissions even for identical workloads, enabling order-of-magnitude reductions through strategic placement.

Energy Source	Carbon Intensity (g CO_2eq/kWh)	Regional Examples
Coal	820 to 1,200	Poland, West Virginia
Natural Gas	350 to 500	Texas combined cycle plants
Solar PV	20 to 50	California, Arizona
Wind	7 to 15	Denmark, Scotland
Hydroelectric	10 to 30	Quebec, Norway
Nuclear	5 to 20	France, Ontario

Geographic optimization can reduce carbon emissions by 10–50× through strategic training location selection, as figure 15.7 illustrates across representative regions.

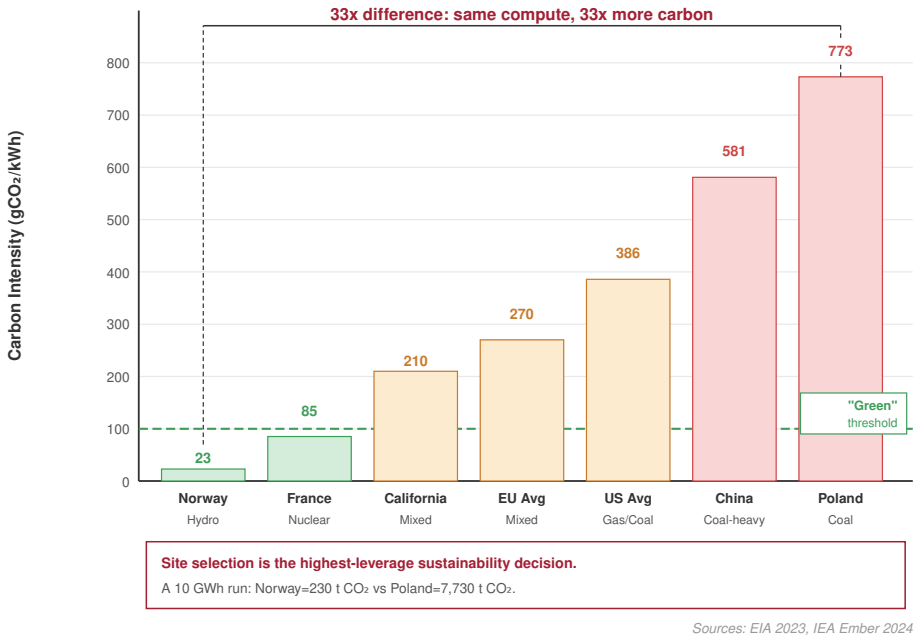


Figure 15.7: Geographic Carbon Intensity: The carbon footprint of a training job depends critically on where it runs. Regions with hydro or nuclear power (for example, Quebec, France) have carbon intensities 10×–50× lower than regions reliant on coal (for example, Poland, West Virginia). Carbon-aware scheduling exploits this variance by moving nonurgent jobs to cleaner grids.

15.2.3 Systematic energy metrics

Quantifying energy efficiency requires systematic metrics that enable comparison across hardware architectures and algorithmic approaches. The metric ladder moves from energy per operation to energy per byte and then to the roofline relationship between arithmetic intensity and data movement. That progression matters because an optimization that reduces FLOPs may do little for a workload whose energy is spent moving bytes.

15.2.3.1 Energy per operation

The fundamental metric for computational energy efficiency is energy consumed per operation, typically measured in picojoules. For AI workloads, the most relevant metrics are energy per floating-point operation and energy per multiply-accumulate, where one MAC operation performs both a multiplication and addition, equivalent to two FLOPs.

Hardware architecture determines energy efficiency across orders of magnitude, spanning nearly four orders of magnitude from general-purpose CPUs to specialized analog accelerators. Table 15.2 quantifies energy efficiency by architecture:

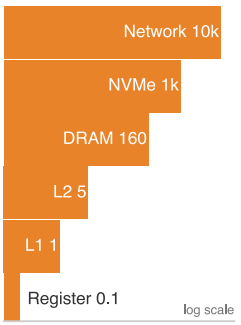
Table 15.2: Energy Efficiency by Architecture: Hardware specialization provides up to four orders of magnitude improvement in energy per operation. Data center accelerators (TPU, GPU) optimize for throughput, while edge accelerators (Ethos-U, MAX78000) optimize for energy per inference. Emerging analog compute approaches promise further efficiency gains.

Architecture	Energy Efficiency (pJ/FLOP or pJ/MAC)	Characteristics
CPU (general)	100 pJ/FLOP	Low utilization, high flexibility
GPU (tensor cores)	10 pJ/FLOP	High throughput, parallel execution
TPU (systolic array)	1-2 pJ/FLOP	Specialized matrix operations, optimized dataflow
Google Edge TPU	2-4 pJ/FLOP	On-device inference, INT8 optimized

Architecture	Energy Efficiency (pJ/FLOP or pJ/MAC)	Characteristics
ARM Ethos-U55	0.5-2 pJ/MAC	Microcontroller NPU, sub-watt TinyML
Maxim MAX78000	0.3-1 pJ/MAC	CNN accelerator with local weight storage
ASIC (INT8)	0.1 pJ/operation	Fixed-function, low precision
Analog/In-Memory Compute	0.01-0.1 pJ/MAC	Emerging technology, compute in memory array

The four-order-of-magnitude spread reflects both circuit-level efficiency and architectural choices affecting utilization. CPUs execute diverse instruction mixes with low average utilization of arithmetic units. GPUs achieve higher utilization through massive parallelism. TPUs and ASICs maximize utilization through specialized datapaths optimized for specific operation types.

Precision directly affects energy per operation. INT8 integer arithmetic consumes approximately one-sixteenth the energy of FP32 floating-point at the same frequency and voltage. This combines reduced datapath capacitance of 4× from bit width with lower voltage requirements of 2× from larger noise margins and simpler control logic of 2× from reduced complexity.



Per-byte move energy spans five orders, register to network.

15.2.3.2 Energy per byte

Data movement often dominates energy consumption in AI workloads with low arithmetic intensity. The energy cost of memory access spans five orders of magnitude across the storage hierarchy:

Table 15.3 reveals a critical insight about memory hierarchy energy costs: moving data from DRAM consumes 10 to 100 times more energy than performing arithmetic operations. The rows trace the path from registers through L1 cache and L2 cache to DRAM, NVMe, and network transfers. For a GPU operating at 10 pJ/FLOP, accessing one FP32 operand from DRAM (4 bytes times 160 pJ/byte = 640 pJ) costs 64 times more than the computation itself. This table makes the energy hierarchy explicit.

Table 15.3: Memory Hierarchy Energy Costs: Energy per byte increases by orders of magnitude moving down the memory hierarchy. Data movement can easily dominate computation energy.

Memory Level	Energy Cost (pJ/byte)	Access Latency
Register	0.1 pJ/byte	1 cycle
L1 Cache	1 pJ/byte	3-5 cycles
L2 Cache	5 pJ/byte	10-20 cycles
DRAM	160 pJ/byte	200-300 cycles
NVMe SSD	1000 pJ/byte	50,000-100,000 cycles
Network	> 10000 pJ/byte	Millions of cycles

The resulting design levers target movement rather than arithmetic:

- On-chip memory for data reuse (NVIDIA tensor cores with shared memory)
- Optimized data layouts minimizing DRAM access (Google TPU systolic arrays)
- Compression reducing data movement (sparse tensor representations)

Together, these levers make sustainability a locality-and-reuse problem, not only a faster-arithmetic problem.

15.2.3.3 Arithmetic intensity and energy roofline

The balance between computation and data movement determines whether energy consumption is compute-bound or memory-bound. Equation 15.4 defines arithmetic intensity (AI), the ratio that determines which resource dominates energy consumption. In these equations, O is the operation count in FLOPs, D_{vol} is data volume in bytes, E_{compute} is energy per FLOP, and E_{move} is energy per byte moved:

$$\text{AI} = \frac{O}{D_{\text{vol}}} \tag{15.4}$$

Arithmetic intensity measured in FLOP/byte determines the dominant energy consumer. Equation 15.5 expresses total energy as the sum of compute and memory contributions, while equation 15.6 isolates the roofline-style dominant term:

$$E_{\text{total}} = O \times E_{\text{compute}} + D_{\text{vol}} \times E_{\text{move}} \tag{15.5}$$

$$E_{\text{dominant}} = \max\left(O \times E_{\text{compute}}, D_{\text{vol}} \times E_{\text{move}}\right) \tag{15.6}$$

The maximum term identifies the dominant bottleneck for roofline reasoning; it is not the full energy in balanced cases. Equation 15.7 defines the crossover arithmetic intensity where compute and memory energy balance:

$$\text{AI}_{\text{crossover}} = \frac{E_{\text{move}}}{E_{\text{compute}}} \tag{15.7}$$

For a GPU with E_{compute} of 10 pJ/FLOP and E_{move} of 160 pJ/byte (DRAM access):

$$\text{AI}_{\text{crossover}} = \frac{160 \text{ pJ/byte}}{10 \text{ pJ/FLOP}} = 16 \text{ FLOP/byte}$$

The **energy roofline model** (figure 15.8) visualizes this relationship between arithmetic intensity and energy efficiency, revealing how different workload types are constrained by different bottlenecks. This energy roofline transposes the performance roofline onto the energy axis: section C.3.1 derives the original ceiling and works the ridge-point analysis that separates memory-bound from compute-bound regimes, the same crossover that here divides memory-dominated from compute-dominated energy consumption.

To make this framework concrete, we can apply it to the most common operation in deep learning: matrix multiplication.

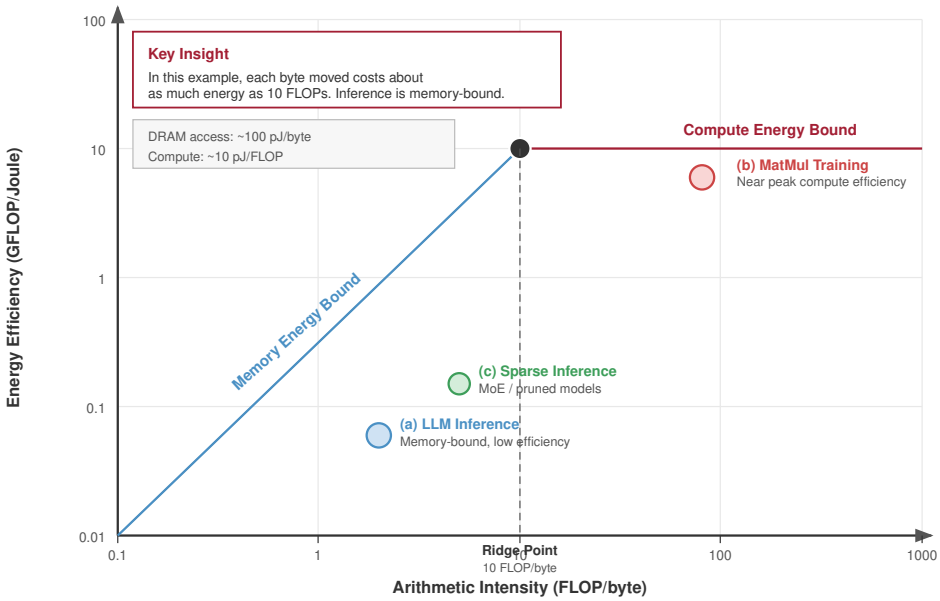
Example 15.2: MatMul energy analysis

Consider matrix multiplication $C = A \times B$ for $N_{\text{mat}} \times N_{\text{mat}}$ matrices in FP32 precision on a GPU with the energy characteristics above.

Step 1: Calculate FLOPs and bytes.

- FLOPs: $2N_{\text{mat}}^3$ (one multiply-add for each of N_{mat}^2 output elements, accumulating over N_{mat} elements)
- Bytes: $3N_{\text{mat}}^2 \times 4$ bytes (read matrices A and B, write matrix C, each FP32 = 4 bytes)
- Arithmetic intensity: $\text{AI} = \frac{2N_{\text{mat}}^3}{12N_{\text{mat}}^2} = \frac{N_{\text{mat}}}{6}$ FLOP/byte

Step 2: Determine the energy-limiting factor. Table 15.4 compares the three regimes directly.



After Horowitz 2014, Williams et al. 2009

Figure 15.8: Energy Roofline Model: Just as performance rooflines limit FLOP/s based on bandwidth, energy rooflines limit FLOP/J. Workloads with low arithmetic intensity (left) are dominated by memory energy (E_{move}), while compute-heavy workloads (right) are limited by arithmetic energy (E_{compute}). Optimizing the wrong metric yields diminishing returns.

Table 15.4: Energy bottleneck comparison: The same energy model classifies small dense work, large dense work, and element-wise work by their dominant energy term.

Workload	Arithmetic intensity	Compute energy	Memory energy	Optimization priority
Small matrix ($N_{\text{mat}} = 96$)	$\text{AI} = 96/6 = 16$ FLOP/byte	$2 \times 96^3 \times 10$ pJ $=$ $17.69\,\mu\text{J} = 0.0177$ mJ	$3 \times 96^2 \times 4 \times 160$ pJ $=$ $17.69\,\mu\text{J} = 0.0177$ mJ	Balanced at the crossover
Large matrix ($N_{\text{mat}} = 1000$)	$\text{AI} = 1000/6 = 167$ FLOP/byte	$2 \times 10^9 \times 10$ pJ $=$ 20 mJ	$3 \times 10^6 \times 4 \times 160$ pJ $= 1.92$ mJ	Improve compute efficiency
Vector addition ($N_{\text{vec}} = 1000$)	$\text{AI} = 1000/12000 =$ 0.083 FLOP/byte	1000×10 pJ $=$ 0.00001 mJ	12000×160 pJ $=$ 0.00192 mJ	Reduce data movement through fusion

Systems insight: Energy optimization follows the same bottleneck logic as latency optimization. Dense matrix multiply rewards efficient arithmetic; element-wise work rewards reducing memory movement.

The energy roofline model reveals why different optimization strategies suit different workloads. Large dense matrix operations benefit from faster arithmetic units. Memory-bound operations like element-wise kernels benefit from data layout optimization, kernel fusion to reduce memory round-trips, and on-chip memory utilization. This framework guides architectural and algorithmic choices for sustainable AI system design.

15.2.4 Energy measurement techniques

Quantifying AI system energy consumption requires measurement at multiple levels of the hardware stack, from chip-level instrumentation to facility-wide monitoring. The method choice is itself an engineering trade-off: hardware counters provide fine attribution, mobile profilers expose platform-specific subsystems, edge instruments capture duty-cycle behavior, and system-level tools connect component measurements to the facility overhead that component counters miss.

15.2.4.1 Hardware power counters

Modern processors include dedicated circuitry for power measurement that software can query through manufacturer-provided interfaces. These hardware counters measure actual power draw rather than estimating from activity, providing ground-truth energy consumption data at microsecond resolution.

Intel's **Running Average Power Limit (RAPL)** interface exposes energy measurements for CPU packages, DRAM, and integrated graphics through model-specific registers (MSRs). RAPL reports cumulative energy, so the measurement pattern is boundary based: sample the counter before a controlled workload region, run the workload, sample it again, and divide the energy delta by elapsed time to recover average power. This makes RAPL useful for CPU preprocessing, data loading, and host-side training work, but it also defines its boundary. RAPL does not cover discrete GPU energy, can require elevated permissions, and must be interpreted with awareness of package scope and counter rollover.

NVIDIA GPUs expose power measurements through the NVIDIA Management Library (NVML), accessible via the `nvidia-smi` command-line tool or programmatic bindings. GPU power monitoring usually starts with instantaneous power draw, which can vary significantly during computation because dynamic voltage and frequency scaling changes the device state from one kernel to the next. A reliable measurement therefore treats the inference or training interval as a trace: synchronize the workload boundary, sample power at a fixed cadence, integrate those samples over time, and report both average and peak power. When data-center GPUs expose accumulated energy counters, those counters are preferable because they avoid aliasing short kernels between samples.

Edge devices and microcontrollers present a different measurement problem. They often lack built-in power counters, operate at milliwatt rather than kilowatt scales, and require external instrumentation for accurate energy profiling. The relevant decision is how much temporal resolution, rail attribution, and cost the workload justifies. INA219 and INA226 I2C-based current sensors provide affordable measurement for development and validation, sampling at rates sufficient to capture inference-level energy consumption. For research requiring nanosecond-resolution measurements of individual operations, instruments like the Joulescope JS220 measure current from sub-microamp sleep states through ampere-level active peaks, enabling characterization of the full dynamic range of edge AI workloads. For large TinyML fleets, edge energy measurement becomes essential for comprehensive sustainability assessment because small per-device errors compound across deployment scale.

15.2.4.2 Mobile platform energy profiling

On mobile platforms, measurement depends on how much attribution the platform exposes. The available profilers trade direct wattage for per-component diagnostic value:

- **Android PowerStats HAL**: Provides per-component power attribution for CPU, GPU, NPU, and radio subsystems, enabling developers to identify which model operations dominate energy consumption.
- **Qualcomm Trepn Profiler**: Offers millisecond-resolution power measurement on Snapdragon platforms, correlating power traces with code execution for NPU workload optimization.
- **ARM Streamline**: Provides energy-annotated profiling for Cortex-A and Mali GPU platforms, enabling identification of inefficient kernel implementations.
- **Apple Instruments Energy Log**: Reports thermal state and energy impact scores for iOS applications, though without direct wattage measurements.

Mobile profiling tools integrate with development workflows, enabling iterative optimization of on-device inference energy consumption during model deployment. Table 15.5 summarizes edge

power measurement instruments across platforms, including resolution, accuracy, and integration requirements.

Table 15.5: Edge Power Measurement Instruments: External power monitors enable energy measurement on devices without built-in counters. The Joulescope JS220 provides the gold-standard accuracy for TinyML research, while INA-series sensors offer cost-effective solutions for deployment validation.

Instrument	Resolution	Accuracy	Use Case
INA219/INA226	100 microsecond sampling	plus or minus 1%	Low-cost embedded profiling
PAC1934	1 millisecond, 4 channels	plus or minus 2%	Multi-rail MCU measurement
Joulescope JS220	Sub-microsecond, nanoamp range	plus or minus 0.1%	Professional TinyML benchmarking
Otii Arc Pro	10 microsecond, automation	plus or minus 0.5%	Automated battery life testing

15.2.4.3 Edge measurement methodology

Edge energy measurements are useful only when they reflect the deployed duty cycle, not a best-case active inference run. Reproducible results require four controls:

- **Baseline characterization**: Measure idle power consumption across all sleep states, as baseline power can vary from 1 microamp in deep sleep to 1 milliamp in idle active states on typical microcontrollers.
- **Warm-up period**: Execute 100 or more inference iterations before measurement to reach thermal equilibrium, as initial iterations may exhibit different power characteristics due to cache warming and voltage regulator settling.
- **Duty cycle accounting**: Report both peak inference power and average power at realistic duty cycles, because edge devices typically operate with significant idle periods between inferences.
- **Peripheral isolation**: Disable or account for peripheral power consumption, such as sensors, radios, and displays, when measuring model inference energy, because these can dominate total system power.

For duty cycle accounting, equation 15.8 expresses the relationship between active and idle power:

$$P_{\text{average}} = P_{\text{active}} \times \delta_{\text{duty}} + P_{\text{idle}} \times (1 - \delta_{\text{duty}}) \tag{15.8}$$

where δ_{duty} is the duty cycle (fraction of time performing inference).

15.2.4.4 System-level energy profiling

Comprehensive energy accounting requires combining chip-level measurements with infrastructure overhead. Equation 15.9 formalizes total energy as the sum of component contributions scaled by facility overhead:

$$E_{\text{total}} = (E_{\text{CPU}} + E_{\text{GPU}} + E_{\text{memory}} + E_{\text{network}}) \times \text{PUE} \tag{15.9}$$

No single counter spans the full energy path, so system-level profilers like Intel VTune, NVIDIA Nsight Systems, and open-source tools such as PowerJoular aggregate measurements across components. For production deployments, smart power distribution units (PDUs) at the rack level provide facility-verified measurements that include cooling overhead.

Equation 15.10 expresses the relationship between measured component power and total facility energy:

$$P_{\text{facility}} = P_{\text{IT}} \times \text{PUE} = (P_{\text{servers}} + P_{\text{network}} + P_{\text{storage}}) \times \text{PUE} \tag{15.10}$$

For a cluster consuming 1 MW of IT power in a facility with PUE of 1.4, total facility power consumption reaches 1.4 MW, with the additional 400 kW powering cooling, power conversion, and infrastructure systems. That automatic 40 percent overhead on all computational power highlights the critical role of facility efficiency. However, operational power consumption is only one piece of the equation; to capture the true environmental cost of our systems, we must formalize how we convert raw kilowatts into tons of carbon emissions.

15.3 Carbon Footprint Calculation

Consider a data center running on 100 percent renewable hydroelectric power. Its operational carbon emissions are effectively zero, but AI trained there is *not* carbon-free. Mining the silicon, manufacturing the GPUs, and pouring the concrete for the data center released thousands of tons of CO_2 before the servers were ever turned on. A true carbon footprint calculation must account for both the energy consumed during operation and the "embodied carbon" emitted during construction.

The lifecycle notebook in section 15.2.1 already computed operational, embodied, and total carbon for one 70B run. This section turns that worked example into the formal accounting model: equations for each term that generalize across workloads, grids, and amortization assumptions rather than a single numeric answer.

15.3.1 Operational carbon calculation

Operational carbon emissions result from electricity consumption during training and inference, scaled by grid carbon intensity. Equation 15.11 quantifies this as the product of energy, grid carbon intensity, and facility overhead:

$$C_{\text{operational}} = E_{\text{total}} \times \text{CI}_{\text{grid}} \tag{15.11}$$

where E_{total} is the facility-level energy from equation 15.9 (component energy already scaled by PUE) and CI_{grid} is the carbon intensity of the electricity grid. The facility overhead enters once, through E_{total}, so it does not appear again in the carbon equation. A concrete training emissions calculation illustrates this framework.

Example 15.3: Training emissions calculation

Consider training a 7 billion parameter model on 64 A100 GPUs for 14 days:

Step 1: Compute energy.

- GPU power: 400 W per A100 at typical training utilization
- Training time: 14 days × 24 h/day = 336 hours
- GPU energy: 64 × 400 W × 336 hours = 8,601,600 Wh = 8,601.6 kWh

Step 2: Add IT support power and apply PUE.

- IT energy after rack-profile support load: 11,827.2 kWh
- Facility PUE: 1.12 (efficient hyperscale data center)
- Total facility energy: 11,827.2 kWh × 1.12 = 13,246.5 kWh

Step 3: Calculate emissions.

- Grid carbon intensity: 429 g/kWh CO_2 (US average)
- Operational emissions: 13,246.5 kWh × 429 g/kWh = 5682.7 kg = 5.7 t

Analysis: Same training in low-carbon region.

- Quebec grid intensity: 20 g/kWh (low-carbon grid)
- Emissions: 13,246.5 kWh × 20 g/kWh = 264.9 kg

Systems insight: The geographic choice alone produces a 21.5× difference in training emissions. Carbon-aware placement changes the environmental cost without changing the model architecture.

15.3.1.1 Embodied carbon assessment

As figure 15.9 illustrates, operational energy dominates total cost of ownership for typical deployments, but embodied carbon from semiconductor fabrication becomes the binding constraint as the grid shifts to renewables.

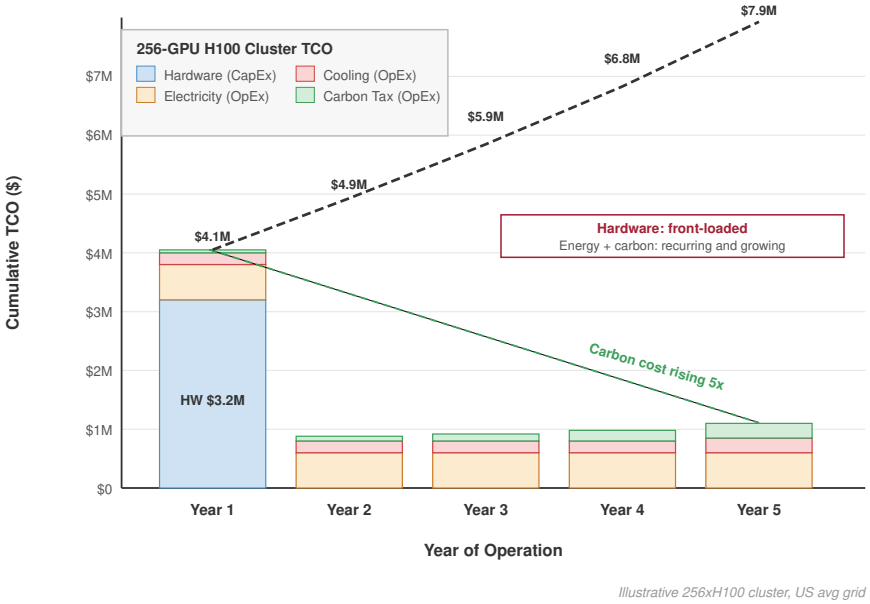


Figure 15.9: Lifecycle Carbon Stack: Five-year stacked bar chart breaking down AI deployment emissions into operational energy and embodied carbon components across years 1–5. For a typical deployment, operational energy dominates the early bars; embodied carbon (semiconductor fabrication and data center construction) becomes the binding component as the grid decarbonizes, making hardware longevity a critical engineering lever.

The key insight from figure 15.9 is the shifting bottleneck: as grids decarbonize, embodied carbon from chip fabrication and data center construction can become the dominant term, making hardware utilization and longevity first-order sustainability levers.

Embodied carbon encompasses emissions from raw material extraction, semiconductor fabrication, assembly, transportation, and end-of-life disposal. For AI hardware, manufacturing emissions are dominated by the energy-intensive nature of advanced semiconductor processes.

Advanced-node AI accelerators carry substantial manufacturing footprints: an NVIDIA A100 GPU embodies approximately 150 kg CO_2eq per unit (Luccioni et al. 2023), and NVIDIA's HGX H100 product carbon footprint implies roughly 164 kg CO_2e per H100 when the baseboard footprint is allocated evenly across its eight GPUs (NVIDIA Corporation 2025), including wafer fabrication at advanced process nodes, high-bandwidth memory production, and packaging. Equation 15.12 amortizes this embodied carbon over the hardware lifetime to compute per-use emissions:

$$C_{\text{embodied,daily}} = \frac{C_{\text{manufacturing}}}{T_{\text{lifetime}} \times 365} \tag{15.12}$$

Understanding how embodied carbon accumulates over time reveals why hardware utilization and lifetime dominate total lifecycle emissions.

> **Systems Perspective 15.1: Embodied carbon amortization**
>
> A hidden cost emerges when renting a GPU for an hour: the fee does not cover electricity alone but also amortizes the carbon debt of manufacturing.

The embodied-carbon amortization formula makes that allocation explicit:

$$C_{\text{total}} = C_{\text{operational}} + \left(\frac{C_{\text{manufacturing}}}{T_{\text{lifetime,years}} \times 365 \times 24} \times T_{\text{job,hours}} \right)$$

Scenario: Training a model for 10 hours on 8 NVIDIA H100s.

- **Operational**: 8 × 0.7 kW × 10 hours = 56 kWh. At 0.429 kg/kWh (US-average grid) = 24 kg.
- **Embodied**: 8 × 164 kg = 1312 kg.
- **Amortization**: Lifetime = 3 years (26280 hours).
  - Hourly "Rent" = 1312 kg / 26280 hours ≈ 0.050 kg/h.
  - Job Cost = 0.050 kg/h × 10 hours = 0.5 kg.

Systems insight: For long-lived hardware in dirty grids, electricity dominates (24 kg vs. 0.5 kg). However, in clean grids (hydro, 0.020 kg/kWh), operational drops to 1.1 kg, making embodied carbon a significant fraction (~30.8 percent) of the total footprint.

This worked example assumes a 4-year service life, slightly longer than the 3-year amortization window used in the lifecycle estimate above; the assumption matters because a longer life spreads the same manufacturing carbon over more service, lowering the per-job share. For an accelerator with 150 kg embodied carbon (per NVIDIA's product carbon footprint) and that 4-year data center lifetime, the first step is daily amortization: 150 kg / (4 years × 365 d/year) ≈ 0.103 kg/day.

The second step assigns that daily share to the job. A training run lasting 14 days on 64 accelerators carries 64 × 14 days × 0.103 kg ≈ 92.1 kg CO_2 of amortized embodied carbon.

The embodied contribution of 92.1 kg represents approximately 1.6 percent of the operational emissions (5682.7 kg) calculated above for the US average grid. If training occurred in Quebec's low-carbon grid, where the same run produced 264.9 kg of operational emissions, the embodied contribution would be about 25.8 percent of total emissions.

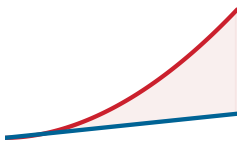
15.3.1.2 Lifecycle carbon accounting

Complete lifecycle assessment combines operational and embodied emissions across all phases. Equation 15.13 aggregates these contributions:

$$C_{\text{lifecycle}} = C_{\text{training}} + C_{\text{inference}} + C_{\text{embodied}} \tag{15.13}$$

As figure 15.10 shows, training dominates this single-deployment lifecycle snapshot, while manufacturing and inference remain significant factors.

That single-deployment snapshot in figure 15.10 tells only part of the story. The cumulative picture is the opposite: a model serving millions of queries per day can exceed its entire training carbon footprint within months, or within days for higher-traffic services, making inference optimization the highest-impact sustainability intervention for production systems over a model's service life.



Cumulative inference emissions overtake the one-time training cost.

For models deployed at scale, inference emissions often dominate the lifecycle. Consider a model serving 10 million queries per day at 0.001 kWh per query. The annual inference energy and emissions break down as follows:

- Daily energy: 10 million queries × 0.001 kWh = 10,000 kWh
- Annual energy: 10,000 kWh × 365 d/year = 3,650,000 kWh
- Annual emissions (US grid): 3,650,000 kWh × 0.429 kg/kWh = 1,565,850 kg = 1565.9 t

Compared with the 7B/64-A100 training example above (5.7 t), cumulative inference emissions exceed training emissions after approximately 1.3 days of deployment at this scale. For larger training runs, the crossover can shift to weeks or months depending on query volume, per-query energy, and grid intensity. The lifecycle perspective therefore sets the priority: optimize inference efficiency for widely-deployed models, and focus training efficiency efforts on models that undergo frequent retraining or experimental iteration.

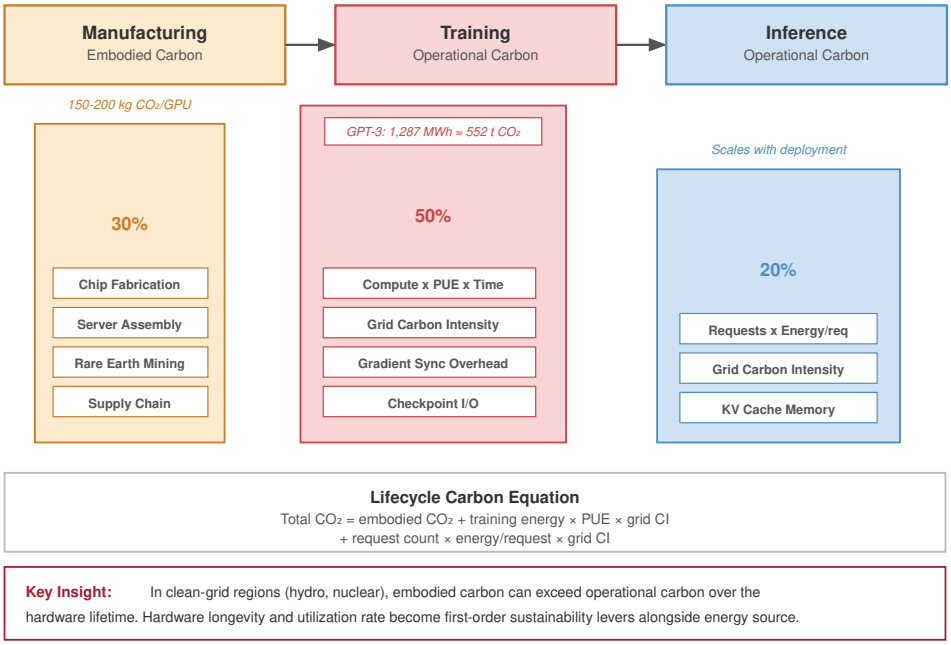


Figure 15.10: The Carbon Lifecycle of an ML System: Single-deployment lifecycle snapshot with training as the largest share, followed by manufacturing and inference.

15.3.1.3 Regional grid intensity data sources

Accurate carbon accounting requires grid intensity data matched to the decision being made. Real-time carbon intensity varies with generation mix, which changes hourly based on demand, renewable availability, and plant dispatch decisions. The data source choice depends on whether the team is estimating a future job, scheduling a live workload, or auditing a completed run.

The US Energy Information Administration (EIA) publishes historical grid emissions factors by region, updated annually. For prospective analysis, these annual averages provide reasonable estimates. ElectricityMap and WattTime provide real-time carbon intensity APIs covering major grids worldwide, enabling carbon-aware scheduling systems. For retrospective analysis of completed training runs, hourly marginal emissions data from these sources enables accurate attribution. Listing 15.1 implements a lifecycle carbon calculator that integrates energy measurements with grid intensity data:

Listing 15.1: Lifecycle Carbon Calculation: Computing total carbon footprint including operational and embodied emissions.

```
def calculate_carbon_footprint(
    gpu_power_watts: float,
    num_gpus: int,
    training_hours: float,
    pue: float,
    grid_intensity_gco2_kwh: float,
    gpu_embodied_kg: float,
    gpu_lifetime_years: float,
) -> dict:
    """Calculate lifecycle carbon footprint for a training run."""

    # Operational emissions
    energy_wh = gpu_power_watts * num_gpus * training_hours
    energy_kwh = (energy_wh * watt * hour).to(kWh).magnitude
    facility_energy_kwh = energy_kwh * pue
    # g CO2 → kg (mlsysim does not export mass pint units yet).
    operational_kg = (
        facility_energy_kwh * grid_intensity_gco2_kwh / THOUSAND
    )

    # Embodied emissions (amortized)
    daily_embodied = gpu_embodied_kg / (gpu_lifetime_years * 365)
    training_days = training_hours / 24
    embodied_kg = num_gpus * training_days * daily_embodied

    return {
        "energy_kwh": facility_energy_kwh,
        "operational_carbon_kg": operational_kg,
        "embodied_carbon_kg": embodied_kg,
        "total_carbon_kg": operational_kg + embodied_kg,
        "embodied_fraction": embodied_kg
        / (operational_kg + embodied_kg),
    }


# Example: 7B model training
_us_grid = Infrastructure.Grids.US_Avg
result = calculate_carbon_footprint(
    gpu_power_watts=400,
    num_gpus=64,
    training_hours=336,  # 14 days
    pue=_us_grid.pue,
    grid_intensity_gco2_kwh=_us_grid.carbon_intensity_g_kwh,
    gpu_embodied_kg=164,
    gpu_lifetime_years=4,
)
print(
    f"Total carbon footprint: {result['total_carbon_kg']:.0f} kg CO2"
)
print(f"Embodied fraction: {result['embodied_fraction']:.1%}")
```

Teams can integrate total lifecycle carbon accounting directly into their orchestration dashboards using this programmatic approach. Calculating operational and embodied emissions for individual training runs, however, captures only one dimension of the problem. The macro-level patterns of how dense AI data centers consume resources at scale reveal additional constraints and optimization opportunities.

15.4 Data Center Energy and Resource Consumption

When a traditional web server handles an HTTP request, the CPU briefly spikes to 20 percent utilization and immediately returns to idle. When a GPU cluster trains a foundation model, thousands of processors run at 100 percent utilization, drawing maximum power continuously for three straight

months. This unprecedented, unyielding thermal density fundamentally breaks traditional data center design, forcing engineers to adopt liquid cooling and redesign entire power distribution networks.

Facility sustainability therefore has to be read as a chain of constraints rather than as a single PUE number. Persistent megawatt demand sets the grid and emissions exposure, power delivery determines how much electricity becomes useful IT load, cooling determines whether dense racks can operate without throttling, and water use determines whether a technically efficient site is locally sustainable.

15.4.1 Data center energy and AI workloads

At facility scale, the optimization target is no longer a single model but the overhead and grid context surrounding every watt of IT power. Data center energy efficiency varies significantly across facilities, so the same IT workload can impose different facility and carbon costs. Power Usage Effectiveness ranges from 1.1 in Google's most efficient facilities to 2.5 in typical enterprise data centers, effectively doubling energy consumption through infrastructure overhead. Geographic location also impacts carbon intensity: the same model trained on a hydro-heavy grid can have tens-fold lower operational emissions than one trained on a coal-heavy grid under the representative intensities used earlier. Without access to renewable energy, these facilities rely heavily on nonrenewable sources such as coal and natural gas, contributing to global carbon emissions. In its 2025 *Energy and AI* analysis, IEA estimated data-center electricity-use emissions at about 180 Mt and projected roughly 300 Mt by 2035 in its Base Case while remaining below 1.5 percent of total energy-sector emissions.[20] The energy burden of AI can grow with data center capacity, training workloads, and inference demand (Patterson et al. 2021). Without intervention, these trends risk making AI's environmental footprint unsustainably large (Thompson et al. 2023; Dodge et al. 2022).

15.4.1.1 Energy demands in data centers

The relevant facility quantity is persistent megawatt-class load, not the model name alone. Companies such as Meta operate hyperscale data centers spanning multiple football fields in size, housing large fleets of AI-optimized servers.[21] Unofficial estimates have suggested GPT-4 may have used on the order of tens of thousands of A100-class GPUs for months (SemiAnalysis 2023), but OpenAI has not disclosed GPT-4's hardware, training compute, model size, or training duration. These facilities rely on high-performance AI accelerators such as NVIDIA H100 GPUs, whose architecture targets high-throughput tensor workloads and improved performance per watt over prior generations (Choquette 2023). Lower-precision methods can improve compute and memory efficiency by replacing full-precision arithmetic when model accuracy permits (Gholami et al. 2021).

AI's rapid adoption across industries drives this dramatic energy consumption. Figure 15.11 illustrates a high-growth scenario in which AI workloads add materially to total data center energy demand after 2024. Masanet et al. (2020) show why this scenario should be read against the historical context: efficiency gains have previously moderated data center energy growth, but sustained demand growth can erode that offset.

Beyond computational demands, cooling accounts for 30–40 percent of data center energy consumption (Ebrahimi et al. 2014), as discussed in section 15.4.6.

While figure 15.11 projects global trends, the United States alone illustrates how cloud and AI infrastructure can reshape national energy planning. Figure 15.12 presents US data center electricity consumption data from the Lawrence Berkeley National Laboratory (LBNL), showing that consumption tripled from 58 TWh in 2014 to 176 TWh in 2023. LBNL's projection treats AI workloads as an important driver of further growth and projects a doubling or tripling by 2028, with the high-end scenario implying that data centers would consume approximately 12 percent of US electricity. This trajectory represents a physical constraint on AI scaling that software optimization alone cannot remove.

15.4.2 Distributed systems energy optimization

Large-scale AI training inherently requires distributed systems coordination, creating additional energy overhead that compounds computational demands. The parallelism strategies examined in Chapter 5 introduce network communication costs that can account for 20–40 percent of total energy

20 **Data Center Emissions Scale**: In IEA's 2025 *Energy and AI* analysis, data centers consumed roughly 1–2 percent of global electricity, with AI contributing to demand growth. IEA estimated emissions from data-center electricity use at about 180 Mt and projected around 300 Mt by 2035 in its Base Case while remaining below 1.5 percent of total energy-sector emissions. The largest hyperscale facilities can draw over 100 MW continuously, equivalent to powering tens of thousands of homes.

21 **Hyperscale Data Center Footprint**: Meta's Prineville facility spans 230,000 m² and houses over 150,000 servers; major cloud fleets consume country-scale electricity annually. These physical scales matter for sustainability because each facility's power demand (100–300 MW) locks in decades of grid-dependency decisions that no algorithmic optimization can undo.

22 **Parallelism Energy Overhead**: Data, model, and pipeline parallelism each impose distinct communication patterns with different energy costs. Data parallelism broadcasts gradients (bandwidth-bound); model parallelism exchanges activations every layer (latency-bound); pipeline parallelism introduces bubble overhead (utilization-bound). GPT-3 combined all three, and the choice of parallelism strategy can swing total training energy by 20–40 percent for the same model.

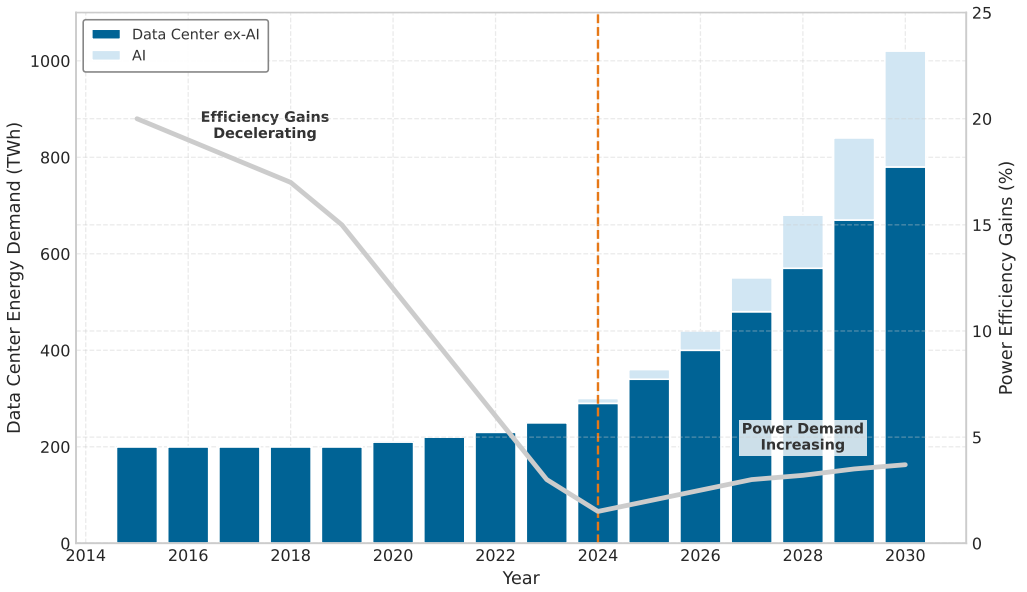


Figure 15.11: Projected Demand: An illustrative high-growth scenario shows AI workloads adding materially to data center power demand by 2030. Masanet et al. (2020) document the historical efficiency gains that previously moderated data center energy growth; the stacked values are scenario assumptions, not a direct forecast from that source.

consumption in large clusters.[22] This coordination across thousands of GPUs requires constant synchronization of computational updates and model parameters[23], generating data movement between nodes. This communication overhead scales poorly: increasing cluster size can increase networking energy superlinearly for all-to-all communication patterns in gradient aggregation.

Addressing these communication overheads, cluster-wide energy optimization requires coordinated resource management that extends beyond individual server efficiency. Four operational levers move the energy budget at cluster scale:

- **Dynamic workload placement**: Consolidate training jobs onto fewer nodes during low-demand periods, allowing unused hardware to enter low-power states and achieving 15–25 percent energy savings.
- **Intelligent scheduling**: Coordinate training across multiple data centers so time-zone differences and regional renewable availability reduce carbon intensity by 30–50 percent through temporal load balancing.
- **Multi-tenant sharing**: Share clusters across model training jobs to improve GPU utilization from typical 40–60 percent to 80–90 percent, effectively halving energy consumption per model trained.
- **Batch processing**: Combine multiple smaller training jobs to use available compute capacity more effectively, reducing the energy overhead of maintaining idle infrastructure.

The common pattern is utilization discipline: the system saves energy by avoiding powered-on capacity that performs no useful model work.

15.4.3 Carbon benchmarks across AI workloads

The environmental impact of AI workloads has emerged as a concern, with carbon emissions approaching levels comparable to established carbon-intensive sectors. Strubell et al. made this concern concrete by estimating that development-scale training and architecture search for a large NLP model could emit as much carbon as several passenger vehicles over their lifetimes (Strubell et al. 2019). Patterson et al. later showed that the most widely quoted architecture-search estimate depended strongly on proxy-task, hardware, data-center efficiency, and grid carbon-intensity assumptions (Patterson et al. 2021). To contextualize AI's environmental footprint, larger and more accurate

23 **Gradient Synchronization Energy Cost**: Ring-allreduce scales communication linearly with message size but requires every node to participate, meaning one slow node wastes energy across the entire ring. At scale, gradient compression (1-2 bit quantization) can reduce network energy by 10–50× per synchronization step, but introduces statistical noise that may require additional training iterations, partially offsetting the savings.

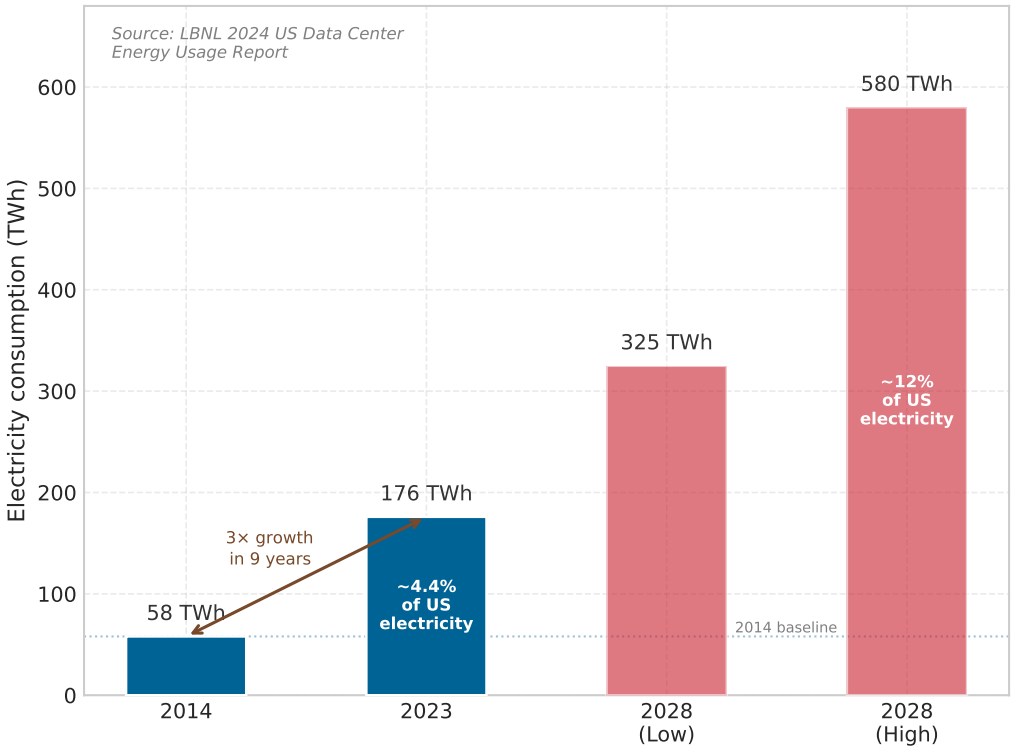


Figure 15.12: The Energy Gap: US data center electricity consumption tripled from 58 TWh (2014) to 176 TWh (2023). LBNL projects a further doubling to 325 to 580 TWh by 2028, with AI workloads treated as an important driver of growth. At the high end, data centers would consume approximately 12 percent of US electricity, representing a physical constraint on AI scaling that software optimization alone cannot remove.

BERT-family models carry meaningfully higher per-query carbon (figure 15.13). The scatter shows that the highest-accuracy variants sit near the top-right of the plot and that the carbon cost rises faster than the accuracy gain. The same trade-off underscores the need for more sustainable AI practices.[24]

The training phase of large natural language processing models can produce carbon dioxide emissions comparable to hundreds of transcontinental flights. At the broader industry scale, AI and data-center emissions are growing rapidly, but the IEA-scale estimates in this chapter do not yet support a direct parity claim with commercial aviation. As AI applications scale to serve billions of users globally, the cumulative emissions from continuous inference operations may ultimately exceed those generated during training.

The operational lesson is that carbon estimates must separate training-only results from deployed inference. Figure 15.14 provides a detailed analysis of carbon emissions across various large-scale machine learning tasks at Meta, illustrating the environmental impact of different AI applications and architectures. This quantitative assessment of AI's carbon footprint grounds mitigation strategies in measured environmental costs rather than estimates.

15.4.4 Comprehensive carbon accounting methodologies

AI's impact extends beyond operational energy consumption. Comprehensive carbon footprint assessment integrates the Three-Phase Lifecycle Analysis (training, inference, manufacturing) with the three standard emission scopes defined by the GHG Protocol. With AI projected to grow rapidly through 2030, understanding total lifecycle costs across all phases and scopes is essential for identifying the most impactful sustainability interventions.

Within an owned data-center facility boundary, Scope 1 emissions originate from on-site power generation including backup diesel generators, facility cooling systems, and owned power plants.

24 **Neural Architecture Search (NAS) Carbon Cost**: The roughly 284,000 kg CO_2e figure from Strubell et al. (2019) is the original Evolved Transformer NAS estimate, not the cost of training one final model. Patterson et al. (2021) later showed that the estimate drops sharply when the search's proxy-task setup, Google's TPU hardware, data-center efficiency, and grid mix are accounted for. The systems lesson is boundary discipline: search, tuning, and failed trials belong inside the accounting boundary, but reusable architecture search should be amortized across the models and domains that reuse the discovered design. This concern helped motivate efficient NAS work: weight sharing, continuous relaxations, and hardware-aware search reduce the search budget, so the meta-optimization of *how* we search for architectures is itself a sustainability lever (Elsken et al. 2019).

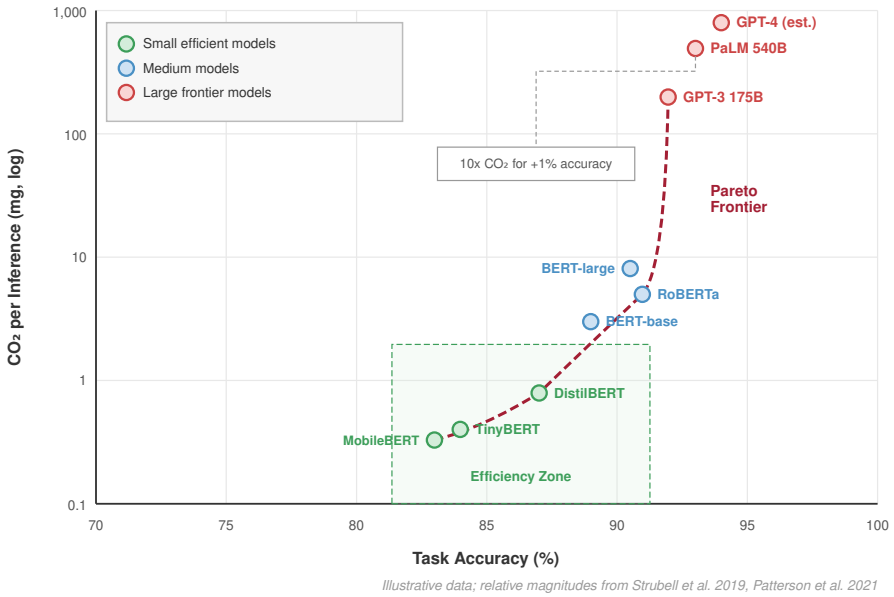


Figure 15.13: Carbon Footprint Benchmarks: Scatter plot of CO_2 emissions per inference (vertical axis) against task accuracy (horizontal axis) for several BERT-family models. Higher-accuracy variants sit near the top-right of the plot; carbon cost rises faster than accuracy gain, emphasizing the environmental impact of pushing toward marginal accuracy improvements.

While many AI data centers primarily use grid electricity, those with fossil-fuel backup systems or owned generation contribute directly to emissions.

Scope 2 emissions represent indirect emissions from electricity purchased to power AI infrastructure. This is often the dominant category for owned facility operations, and it varies dramatically by geographic location and grid energy mix. As established in section 15.1.1, this geographic lever sits in the 8 to 40 times range for representative region pairs; comparing the dirtiest coal grid against the cleanest hydro grid stretches it toward the high end of the 10 to 80 times span.

Scope 3 emissions constitute the most complex category, encompassing hardware manufacturing, cloud supply chains, transportation, disposal, and downstream use. Semiconductor manufacturing is carbon-intensive.[25] For low-utilization hardware or clean-grid deployments, embodied accelerator emissions can rival months or years of operation; for sustained high-power training on carbon-intensive grids, the break-even can be much shorter. Under company-wide or value-chain AI accounting, Scope 3 can dominate even when Scope 2 dominates a single owned facility's operational footprint.

Beyond manufacturing, Scope 3 emissions include the downstream impact of AI once deployed. AI services such as search engines, social media platforms, and cloud-based recommendation systems operate at enormous scale, requiring continuous inference across millions or even billions of user interactions. The cumulative electricity demand of inference workloads can ultimately surpass the energy used for training, further amplifying AI's carbon impact. End-user devices, including smartphones, IoT devices, and edge computing[26] platforms, also contribute to Scope 3 emissions, as their AI-enabled functionality depends on sustained computation. In large technology-company sustainability reports, Scope 3 often dominates companywide emissions; attributing that share specifically to AI-powered services requires workload-level accounting rather than companywide totals alone.

Operational emissions capture only the production phase of AI. Software development itself adds another layer of environmental impact that is rarely accounted for.

25 **EUV Lithography Energy Cost**: Each ASML EUV machine draws 1 MW continuously and consumes 30,000 liters of ultrapure water daily, a 10× energy increase over older deep-UV systems. Since EUV is required for sub-7 nm nodes used in many advanced AI accelerators, the embodied energy of each chip generation compounds: more transistors per die means more EUV exposure steps, making advanced-node fabrication an important component of AI's Scope 3 emissions.

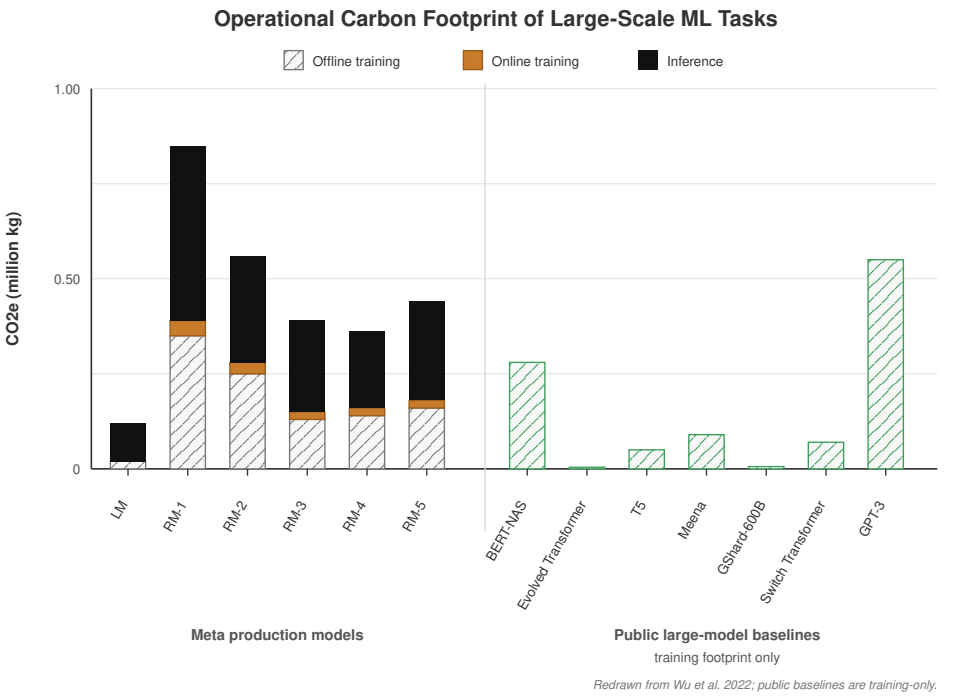


Figure 15.14: Operational Carbon Footprint of Large-Scale ML Tasks: Stacked bars show lifetime CO_2e for Meta's recommendation models (RM-1 through RM-5), a language model (LM), and several open-source large models (BERT-NAS, Evolved Transformer, T5, Meena, GShard-600B, Switch Transformer, GPT-3). Recommendation models are dominated by inference (solid black), while the public open-source baselines report training-only footprints (hatched), underscoring that inference serving at scale can rival or exceed training emissions for deployed systems. Source: (Wu et al. 2022).

26 **Edge AI Energy Paradox**: Edge inference reduces per-query latency from 100–200 ms (cloud) to 1–10 ms, but distributes power draw across many always-on devices. Tesla's FSD computer draws 72 W continuously while driving; scaling comparable onboard compute across a global vehicle fleet would imply roughly 100 GW of collective power, comparable to dozens of large power plants. The sustainability trade-off is that edge eliminates network energy but creates an unmetered, distributed energy footprint invisible to carbon accounting frameworks.

Systems Perspective 15.2: Hidden carbon cost of software

Beyond direct training and inference energy use, the entire software development ecosystem for AI has a significant, though difficult to measure, carbon footprint. The millions of continuous integration and continuous deployment (CI/CD) pipeline runs, constant code recompilation during development, operation of massive version control systems like GitHub, and the computational resources consumed by code review systems, automated testing frameworks, and collaborative development platforms all contribute to environmental impact. Large AI research organizations may run thousands of experimental training runs, most of which never reach production, consuming substantial energy in the exploration process. The entire ecosystem of AI development is energy-intensive, extending well beyond the final model training and inference phases.

The **GHG Protocol**[27] framework (Institute and Sustainable Development 2023) provides the standard categorization for these emissions, summarized in figure 15.15. Three scopes provide an engineering classification checklist:

- **Scope 1 (Direct Emissions)**: Arise from direct company operations—backup generators, company-owned power generation.
- **Scope 2 (Indirect Energy Emissions)**: Electricity purchased from the grid, the primary emission source for cloud computing workloads.
- **Scope 3 (Value Chain Emissions)**: Extend beyond direct control—semiconductor manufacturing, hardware transportation, end-of-life disposal of AI accelerators.

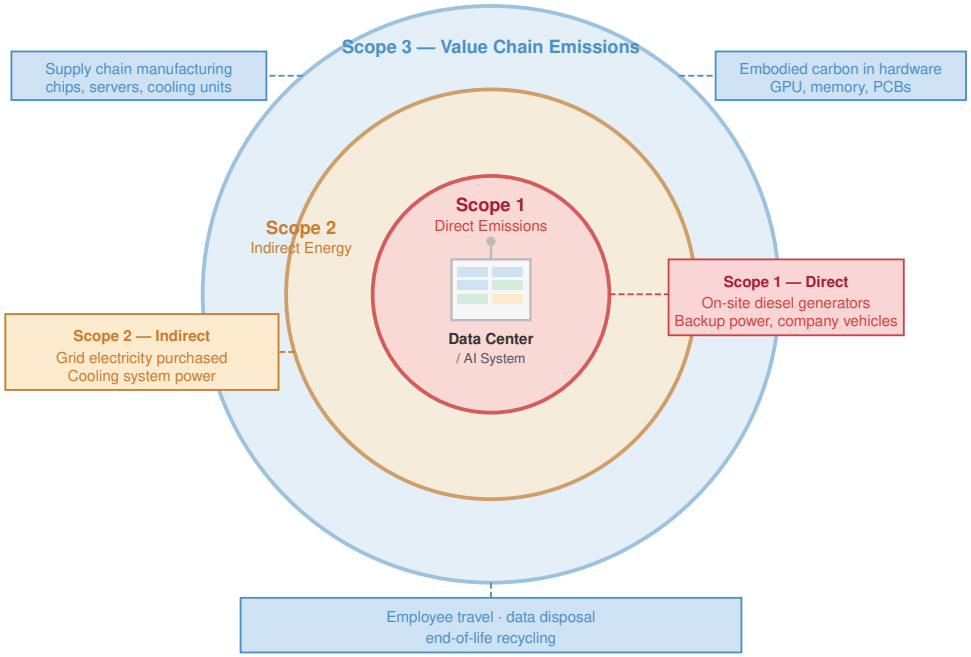


Figure 15.15: GHG Emission Scopes: Hub-and-spoke layout placing the reporting organization at the center, with Scope 1 (direct emissions), Scope 2 (purchased energy), and Scope 3 (value chain) arranged as distinct branches around it. Each branch lists representative emission sources, supporting comprehensive environmental-impact assessment. Source: Ucircularise.

Categorizing these emissions into Scope 1, 2, and 3 frameworks provides a standardized vocabulary for corporate environmental reporting. Correctly applying this framework in practice requires classifying the various hidden emission sources across a typical ML platform's operational lifecycle.

27 **GHG Protocol**: Developed jointly by the World Resources Institute and WBCSD, this framework is used by over 90 percent of Fortune 500 companies reporting to CDP. Its three-scope taxonomy matters for ML systems because most AI carbon hides in Scope 3 (hardware manufacturing, cloud compute supply chains), which companies historically underreport by 50–70 percent compared to Scopes 1 and 2.

Checkpoint 15.2: Accounting for invisible carbon

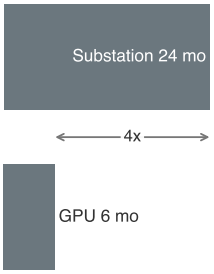
You are auditing the carbon footprint of a machine learning platform. Classify the following emission sources into Scope 1 (Direct), Scope 2 (Indirect Energy), or Scope 3 (Value Chain):

- ☐ **Backup generation**: Classify diesel burned by backup generators during a grid outage at your owned facility.
- ☐ **Purchased electricity**: Classify electricity purchased from the grid to power your leased NVIDIA H100 cluster.
- ☐ **Embodied manufacturing**: Classify the embodied carbon emitted during the manufacturing of the GPUs by TSMC.
- ☐ **End-user operation**: Classify emissions from the end-user's smartphone battery while running your mobile inference app.

Accurately classifying these hidden emissions forces engineering teams to take responsibility for the entire value chain of their deployments. The abstract energy metrics of a facility, and the resulting carbon footprint, are ultimately governed by the physical thermodynamics of the rack. Delivering 120 kW to a single cabinet and extracting the resulting heat requires moving beyond traditional air conditioning.

15.4.5 Power delivery

Capacity planning fails when engineers budget only the accelerator TDP. Electricity loses energy and reliability margin at each transformation before it reaches a GPU's voltage regulators, so the delivery path identifies both the facility bottleneck and the heat that cooling must remove. The path starts outside the building: utility power arrives as high-voltage AC, typically 13.8–69 kV depending on the country and facility size, and a dedicated substation or transformer yard steps it down to medium voltage. The largest ML facilities require their own substation, which takes 18–24 months to build and requires coordination with the local utility. The grid connection is the ultimate bottleneck: no amount of engineering inside the building can deliver more power than the grid provides.



Substation lead time can run 4× longer than GPU procurement.

> Systems Perspective 15.3: The interconnection queue
>
> Deploying 10,000 GPUs and deploying the electrical substation required to power their IT load (9.6 MW, after the 7 MW GPU subtotal is expanded for rack support power) run on very different clocks. On the silicon path, GPU supply chains are volatile, but typical enterprise lead times are 6 months. On the infrastructure path, permitting, EPC (Engineering, Procurement, Construction), and grid connection for a new 10+ MW substation averages 24 months. Infrastructure therefore takes 4× longer to deploy than the accelerators themselves.
>
> **Systems insight**: In the era of the ML Fleet, the primary bottleneck is not the supply chain of silicon, but the interconnection queue of the grid. As of 2024, there are over 2000 GW of capacity waiting for grid connection in the US alone. An engineer who optimizes for GPU utilization without a 2-year power roadmap will find their fleet "electrically stranded": expensive silicon sitting in a dark building waiting for a transformer.

Inside the facility, an Uninterruptible Power Supply (UPS) conditions incoming power and provides battery backup during brief outages. Modern online double-conversion UPS systems convert AC to DC, charge a battery bank, and then convert back to AC, which ensures clean power but loses 3–5 percent efficiency. High-efficiency eco-mode designs bypass that double conversion during normal operation, achieving 98–99 percent efficiency with slightly less protection against input anomalies. The Power Distribution Unit (PDU) then distributes conditioned power from the UPS to racks, often providing the final AC step-down from 480 V to 208/240 V for servers.

Some ML facilities use **48 V DC distribution**, which eliminates one conversion stage and improves efficiency by 2–3 percent. This improvement is not merely an incremental gain in a generic data center setting; for dense ML accelerator baseboards, it addresses a hard physical constraint imposed by the current demands of the hardware itself. A training node with eight H100 GPUs drawing 700 W each requires over 5,600 W just for the accelerators. At traditional 12 V delivery, meeting that demand requires pushing nearly 470 A across the baseboard busbars. At that current level, I^2R losses in the copper conductors themselves generate substantial heat and produce voltage drop that undermines the tight voltage tolerances of VRMs. Moving to 48 V reduces the delivered current by a factor of four, which reduces I^2R distribution losses by a factor of sixteen. For tightly integrated ML baseboards—such as NVIDIA's HGX—48 V DC is less a design preference than a requirement for operating at rated power density without melting power connectors. Google pioneered 48 V DC distribution in their data centers, and the Open Compute Project (OCP) has standardized rack-level 48 V DC power buses for high-density compute.

At ML rack power densities, the difference between 95 percent and 98 percent distribution efficiency is meaningful. For a 33 kW rack, a 3 percent efficiency improvement saves approximately 0.99 kW of power per rack. Across a facility with 300 racks, this saves 297 kW, enough to power roughly 36 additional GPU nodes, or about 9 four-node racks under this rack-power mix. Over a 3-year lifecycle at \$0.07/kWh, the efficiency improvement saves ~\$546,361 in electricity costs.

The final conversions happen at the server and baseboard. Each server contains power supply units (PSUs) that convert rack-level voltage to 12 V DC or directly to the multiple voltages needed by baseboard components. A DGX H100, for example, uses multiple high-efficiency PSUs rated for 10 kW total, with N+1 redundancy so one failed PSU does not take the node offline. Voltage regulator modules (VRMs) then convert 12 V DC to the 0.7–1.0 V required by the GPU core and the 1.1–1.2 V

required by HBM. These regulators must respond to load changes within microseconds as the GPU moves between idle and full-load computation, and they operate at 90–95 percent efficiency.

Checkpoint 15.3: Power delivery physics

Verify your understanding of the data center power path:

- ☐ **Grid stress**: Explain why synchronous ML training creates more stress on the power grid than asynchronous web traffic.
- ☐ **Power ramp**: Define the "Power Ramp" problem and explain how supercapacitor banks address it.
- ☐ **Power budgeting**: Explain why a node with 8 GPUs at 700 W each is budgeted at ~7 kW rather than 5.6 kW.
- ☐ **AC-DC conversion**: Evaluate the true-or-false claim that converting high-voltage AC to low-voltage DC at the rack level is more efficient than doing it at the server level.

At 700 W per GPU, the VRM dissipates 35–70 W of heat, which must be cooled along with the GPU itself. This VRM heat is sometimes overlooked in thermal design: in a liquid-cooled system where cold plates cover the GPUs, the VRMs are typically still air-cooled by small fans, creating a thermal management challenge for the remaining components that do not have direct liquid cooling contact.

The cumulative efficiency across all five stages is typically 85–90 percent, meaning that for every 100 W of useful computation, 10–15 W is lost as heat in the power delivery chain itself. This overhead is part of the PUE calculation and represents an irreducible cost of converting utility power to useful computation.

To make this concrete, consider the power budget for a single rack containing four DGX H100 nodes:

Table 15.6 itemizes the full rack budget so facility sizing includes host, network, conversion, and cooling overhead rather than only GPU TDP.

Table 15.6: Power Budget for a Four-Node DGX H100 Rack: GPUs consume two thirds of total rack power, but the remaining one third (host systems, networking, power conversion, and cooling) cannot be eliminated and must be budgeted for when sizing the facility's electrical infrastructure. The power conversion losses (8 percent) represent the cumulative inefficiency of the five-stage delivery chain.

Component	Power (kW)	% of Rack Total
GPU compute	22.4 kW	67%
Host CPUs and DRAM	3.2 kW	10%
NVSwitch fabric	1.6 kW	5%
InfiniBand HCAs	0.8 kW	2%
Power conversion losses	2.8 kW	8%
Cooling overhead (PUE ~1.09)	2.7 kW	8%
Total	**33.5 kW**	**100%**

As table 15.6 shows, 32 GPUs at 700 W each deliver 22.4 kW, but that GPU subtotal alone understates the true rack power requirement by roughly 50 percent. Infrastructure planners who size their facility based on GPU TDP alone will underestimate the electrical load and may discover during commissioning that their power capacity is insufficient.

ML training workloads impose a unique challenge on this power chain: **synchronous transients**. In traditional web-serving data centers, thousands of servers handle independent requests with uncorrelated power draws. The aggregate load is smooth and predictable, varying by perhaps 10–20 percent over the course of a day.

In a training cluster, the picture is radically different. All accelerators execute the same computation in lockstep. When a large matrix multiplication begins, thousands of Tensor Cores across the cluster

activate simultaneously, and power demand surges by 40–60 percent within microseconds. When the computation pauses for gradient synchronization, demand drops just as sharply. These **power ramps** stress every component in the delivery chain, from the VRMs on the baseboard to the transformers in the substation.

Example 15.4: Power-ramp sizing

Scenario: A 512-GPU training cluster experiences intermittent node resets during synchronized training steps. The failures appear unrelated to the model code.

Failure mode: Hardware diagnostics show no component defects. The likely root cause is the power delivery chain: synchronous training steps can create a few-hundred-kilowatt load step – roughly 300 kW for a 600 W idle-to-peak swing across 512 GPUs before facility overhead – that exceeds the response time of upstream power-conditioning equipment.

Consequence: Voltage sags can trigger accelerator undervoltage protection circuits and force hard resets. The mitigation is local ride-through capacity, such as supercapacitor banks, that smooths the transient until slower UPS and facility systems respond.

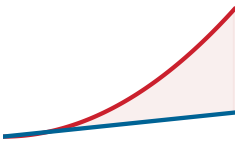
Systems insight: ML clusters do not behave like traditional data center workloads: the *temporal correlation* of power demand across thousands of chips creates a qualitatively different electrical engineering challenge.

Modern ML data centers address power transients through a layered defense strategy, with each layer covering a different timescale. Supercapacitor banks provide the first line of defense, delivering hundreds of kilowatts within microseconds to smooth the initial surge. Unlike batteries, which have response times measured in milliseconds, supercapacitors store energy electrostatically and can discharge instantaneously. A typical installation places 50–100 kJ of supercapacitor storage per rack, enough to sustain a 100 kW transient for 0.5–1.0 seconds.

Battery-backed UPS systems with fast inverter response (under 10 ms switching time) handle longer transients and provide ride-through capability during brief grid disturbances lasting up to several minutes.

Dedicated electrical substations with custom transformer designs serve the largest installations. These transformers are rated for the high di/dt (rate of current change) characteristic of ML workloads, with custom winding configurations that can handle rapid load swings without voltage distortion. Standard utility transformers are designed for slowly varying loads and can experience magnetic saturation when subjected to the rapid load changes that ML training creates.

To appreciate the magnitude of these transients, consider a 1024-GPU cluster transitioning from communication phase (400 W per GPU average) to matrix multiplication phase (700 W per GPU). The power delta is 300 W $\times$ 1024 = 307 kW, and this transition occurs in approximately 100 µs. The rate of power change is therefore 307 kW/100 µs = 3.07 GW/s. No passive electrical component can respond at this rate; only energy storage devices (supercapacitors) positioned physically close to the load can absorb the transient before it propagates into the building's electrical distribution.



Training phase transitions create microsecond power shocks.

The power delivery chain itself introduces inefficiencies at each stage. Utility-to-medium-voltage transformation loses 1–2 percent. The UPS loses 3–5 percent (modern double-conversion designs) or 1–2 percent (eco-mode designs that bypass the inverter during normal operation). The PDU loses 2–3 percent. Voltage regulation on the baseboard loses another 5–8 percent. Cumulatively, 10–15 percent of the power drawn from the grid is dissipated as heat in the delivery chain before it ever reaches a transistor. This overhead is part of the PUE calculation: a PUE of 1.10 implies that the delivery chain and cooling together consume 10 percent above the IT load.

At the largest scales, the data center's power draw represents a significant fraction of the local electrical grid's capacity. A 100,000-GPU cluster at 700 W per GPU has a 70 MW accelerator subtotal, but rack-profile support power raises the IT load to 96.2 MW before PUE. With PUE, the total facility draw approaches 107.8 MW. This is equivalent to powering a small city. Such installations require dedicated feeds from the electrical grid, often with purpose-built substations and transmission lines. The lead time for grid interconnection can exceed two years, making power availability one of the longest-lead-time constraints in building ML infrastructure.

At large-cluster scale, the same delivery limit becomes a grid-procurement problem rather than an electricity bill. GPT-3's 1,287 MWh training run is the chapter's roughly 120-household-year anchor. Later large systems have not disclosed comparable training-energy accounts, which is precisely why the engineering discipline must track energy at the workload level instead of relying on public model cards alone.

This grid-procurement problem worsens when model and data growth multiply each other. The scaling of energy consumption with model size is approximately linear: doubling the model parameters roughly doubles the training energy (assuming the same number of training tokens per parameter). When both model size and dataset size increase simultaneously, total training energy grows super-linearly. A model with $10\times$ the parameters, trained on $10\times$ the data, requires approximately $100\times$ the energy.

The electricity consumed by a single large-model training campaign can become nonnegligible relative to the output of a power plant. A 100 MW training facility operating at full capacity for one year consumes 876 GWh, which is comparable to the annual output of a 100 MW wind farm (at 30 percent capacity factor, a wind farm produces about 262.8 GWh per year, so the training facility would require the equivalent of approximately 3.3 large wind farms).

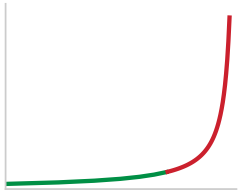
Power availability and power source therefore become the same infrastructure decision. Organizations training large models may seek to match their electricity consumption with renewable energy generation, either by locating data centers near renewable sources (hydro, wind, solar) or by purchasing renewable energy certificates (RECs) to offset their grid consumption.

Direct investment in clean generation is the strongest form of this decision because it adds capacity rather than only reallocating credits. Microsoft, for example, has signed agreements to purchase nuclear energy from restarted reactors, recognizing that the scale and consistency of ML training loads require baseload power sources that renewable intermittent sources alone cannot provide. Google has similarly invested in geothermal energy projects, which provide consistent power output independent of weather conditions.

The carbon intensity of the energy grid dictates the true environmental cost of a training run. A facility powered by hydroelectric dams in the Pacific Northwest emits approximately 50 g CO_2/kWh, while a coal-heavy grid can produce 400 g CO_2/kWh or more. A single training run for our 175B model, consuming approximately 1,287 MWh, implies a carbon impact ranging from 64 tonnes to 515 tonnes depending solely on location – an $8\times$ variance that makes site selection a first-order decarbonization lever. This pairing of Pacific Northwest hydro against a moderate coal grid sits at the conservative end of the geographic span established in section 15.1.1; a cleaner hydro grid widens it further. This environmental calculus can drive infrastructure decisions: organizations that can locate training clusters in low-carbon regions may achieve both lower electricity costs (hydroelectric power is often cheaper than fossil-fuel generation) and lower carbon footprints, a rare alignment of economic and environmental incentives.

15.4.6 Cooling

Every watt of electrical power delivered to a GPU is ultimately converted to heat. The first law of thermodynamics guarantees this: the electrical energy is converted to computational work (switching transistors), but the "work" product is just bit flips in memory, which themselves have negligible energy. All of the input energy exits the system as thermal energy that must be physically removed from the chip, transported out of the rack, and rejected to the environment. The fundamental physics of heat transfer establishes an unavoidable constraint: the rack-level electrical load represented by 33.5 kW must be absorbed by a cooling medium and carried away at the same rate, continuously. If cooling falls behind even briefly, chip temperatures rise, triggering thermal throttling that reduces clock speeds and throughput. At extreme temperatures, the silicon can sustain permanent damage.



Below the thermal limit the chip runs safely; cross it and throttling climbs steeply.

Air cooling, the dominant technology for decades, works by blowing room-temperature air across heat sinks attached to the chips. The air absorbs heat at a rate determined by its specific heat capacity, roughly 1.0 kJ/kg/K. The heated air is exhausted from the rear of the rack, typically 15–20 degrees Celsius warmer than the inlet, and directed to a computer room air conditioning (CRAC) unit that cools it before recirculating.

The fundamental problem is that air is a poor thermal conductor. Its thermal conductivity is only 0.026 W/m/K, compared to 0.6 W/m/K for water and 400 W/m/K for copper. To remove

100 kW from a rack using air alone, the fans must move enormous volumes of air at high velocity, consuming 30–40 percent of the rack's total power budget just for cooling.

The physics can be made precise with a simple calculation. The heat removal capacity of a fluid flow is:

$$Q = \dot{m} \times c_p \times \Delta T$$

where Q is the heat removed (watts), $\dot{m}$ is the mass flow rate (kg/s), c_p is the specific heat capacity (J/kg/K), and ΔT is the temperature difference between outlet and inlet. For air with c_p of 1,005 J/kg/K and a typical ΔT of 15 K (inlet at 20 degrees C, outlet at 35 degrees C), removing 100 kW requires a mass flow rate of 100,000 $\div$ (1,005 $\times$ 15) $\approx$ 6.6 kg/s. At sea-level air density of 1.2 kg/m^3, this corresponds to a volumetric flow rate of 5.5 m^3/s, or approximately 11,713 CFM (cubic feet per minute). Moving this much air through the confined space of a server rack requires powerful fans that themselves consume substantial power.

At higher power densities (above 30 kW per rack), the fan power begins to approach or exceed the compute power, at which point the cooling system is consuming more energy than the computation it supports. The Power Usage Effectiveness (PUE)[28] metric captures this overhead: a PUE of 1.5 means that for every watt consumed by compute, an additional 0.5 watts is consumed by cooling and power distribution overhead.

Reducing PUE is a primary engineering objective for ML data centers because the cooling overhead is wasted energy that produces no useful computation. At the scale of a 10,000-GPU cluster consuming 9.62 MW of IT power after rack support load, the difference between PUE 1.5 and PUE 1.1 is 3.85 MW of wasted power, costing approximately $2.4M per year in electricity and requiring proportionally more cooling infrastructure to dissipate.

Water has a specific heat capacity of 4.18 kJ/kg/K, over four times that of air, and a thermal conductivity roughly 25$\times$ higher. These physical properties make water an inherently superior heat transfer medium. To appreciate the magnitude of the difference, consider how much fluid must flow to remove 700 W from a single GPU. Air at a 15-degree temperature rise requires tens of liters per second of airflow (a small wind tunnel). Water at the same temperature rise requires only about 0.01 liters per second (a thin stream). This thousands-fold difference in volumetric flow rate is why air cooling requires massive fans and carefully designed airflow paths, while liquid cooling requires only thin pipes and small pumps.

Direct-to-chip liquid cooling routes chilled water (or a specialized dielectric coolant) through machined copper cold plates mounted directly on each GPU package. The cold plate makes physical contact with the GPU's heat spreader through a thin layer of thermal interface material, creating a thermal path with a resistance of less than 0.1 K/W. The coolant absorbs heat within millimeters of the die surface and carries it via manifolds and pipes to a Coolant Distribution Unit (CDU) at the rack or row level.

The CDU transfers heat from the chip-level coolant loop (a closed loop using deionized water or dielectric fluid) to the building's chilled water loop, which rejects the heat to the outside environment via cooling towers or dry coolers. This two-loop design isolates the chip-level coolant (which must be ultra-pure to avoid mineral deposits on the cold plates) from the building-level water (which is less rigorously filtered).

Because liquid coolant is far more effective at absorbing heat per unit volume, the server-level fans are eliminated entirely (or reduced to small units for auxiliary components like DIMMs and VRMs). The cooling power overhead drops to 3–8 percent of IT power, yielding PUE values of 1.03–1.08. An additional benefit is noise reduction: liquid-cooled data centers are dramatically quieter than their air-cooled counterparts, which matters for facilities co-located with offices or in noise-regulated areas.

A more aggressive approach, immersion cooling, submerges entire server boards in a tank of nonconductive dielectric fluid. The fluid absorbs heat through direct contact with every component surface, eliminating the need for cold plates, fans, and even heat sinks. The principle is simple: if every surface of the board is in contact with coolant, the heat has nowhere to accumulate and is removed uniformly across the entire assembly.

Single-phase immersion cooling uses a fluid that remains liquid throughout the process, with heat carried away by convection currents in the tank. The heated fluid rises to the surface, is pumped

28 **PUE (Power Usage Effectiveness)**: Defined by The Green Grid consortium in 2007 as $P_{\text{total}}/P_{\text{IT}}$. Google's fleet-wide PUE averages 1.10; the industry average is ~1.58. For a 10,000-GPU cluster consuming 9.62 MW of IT power after rack support load, reducing PUE from 1.58 to 1.10 saves 4.62 MW of cooling overhead – roughly $2.8M per year in electricity and the equivalent of removing 3,800 residential homes from the grid. ML-specific facilities with direct-to-chip liquid cooling have demonstrated PUE values of 1.03–1.08.

through a heat exchanger to reject the heat to the building's chilled water loop, and returns to the bottom of the tank.

Two-phase immersion cooling takes this further: the fluid boils at the chip surface, absorbing the latent heat of vaporization (roughly 100× more energy per gram than a simple temperature change), condenses on a cold surface at the top of the tank, and drips back down. This cycle is self-sustaining and remarkably efficient, removing over 200 kW per rack with PUE values approaching 1.02.

The trade-off is serviceability: accessing a failed component requires draining or partially submerging in fluid, and the dielectric fluids themselves are expensive ($20–50 per liter). A single immersion tank holding four server boards may contain 500-1,000 liters of fluid, representing $10,000-50,000 in coolant cost alone. The operational procedures for immersion-cooled facilities differ sharply from air-cooled ones, requiring specialized training for technicians and different approaches to cable management, since all connectors must be compatible with prolonged fluid exposure. Standard copper cables and connectors can corrode or swell when exposed to some dielectric fluids, necessitating the use of specialized fluid-resistant materials that add cost and reduce the available supply chain options. Table 15.7 compares the air and liquid regimes that determine when these operational costs become unavoidable.

Table 15.7: The Shift to Liquid Cooling: At rack power densities above 30 kW, air cooling requires fan power that approaches the power consumed by the GPUs themselves. Liquid cooling is not a premium option; it is a thermodynamic requirement for dense ML racks.

Metric	Air Cooling (Legacy)	Liquid Cooling (Modern)
Max Power Density	~20–30 kW/Rack	>120 kW/Rack
Cooling Efficiency	PUE ~1.5–2.0	PUE ~1.05–1.10
Mechanism	Forced-Air Fans	Direct-to-Chip Coolant
Heat Carrier	Air (1.0 kJ/kg/K)	Water (4.18 kJ/kg/K)
Fan Power	30–40% of IT load	<5% of IT load

The architecture comparison in figure 15.16 shows the physical reason for that threshold: air moves heat through bulk airflow, while liquid moves heat through a direct thermal path from chip to coolant.

Figure 15.16 and table 15.7 together illustrate the stark contrast between these approaches. The capital cost of these cooling technologies spans an order of magnitude. Standard air cooling infrastructure costs $2,000–5,000 per rack (fans, CRAC units, raised floor tiles). Direct-to-chip liquid cooling costs $15,000–25,000 per rack (cold plates, manifolds, CDUs, piping). Full immersion cooling costs $30,000–50,000 per tank (dielectric fluid, sealed tanks, specialized heat exchangers). The break-even analysis between air and liquid cooling depends on rack power density: at 20 kW per rack, air cooling's lower CapEx wins over a 3-year lifecycle. At 40 kW per rack, the electricity savings from liquid cooling's lower PUE (1.08 vs. 1.5) offset the higher CapEx within 18–24 months. At 60+ kW per rack, a common regime for dense ML infrastructure, air cooling is physically impossible, making the comparison moot. For our 175B model's 32-rack training cluster at 33.5 kW per rack, direct-to-chip liquid cooling is the reference choice, balancing density, serviceability, and cost. Immersion cooling offers marginal PUE improvement (1.03 vs. 1.08) but introduces operational complexity that many organizations may find unjustified at these rack densities.

Napkin Math 15.5: The cooling tax

Consider a 1,024-GPU cluster of H100s at 700 W each.

- **IT Power**: 1,024 × 700 W = 716.8 kW
- **Air cooling (PUE 1.5)**: Total facility power = 716.8 kW × 1.5 = 1068.0 kW. Cooling overhead = 351.2 kW.
- **Liquid cooling (PUE 1.08)**: Total facility power = 716.8 kW × 1.08 = 774.1 kW. Cooling overhead = 57.3 kW.

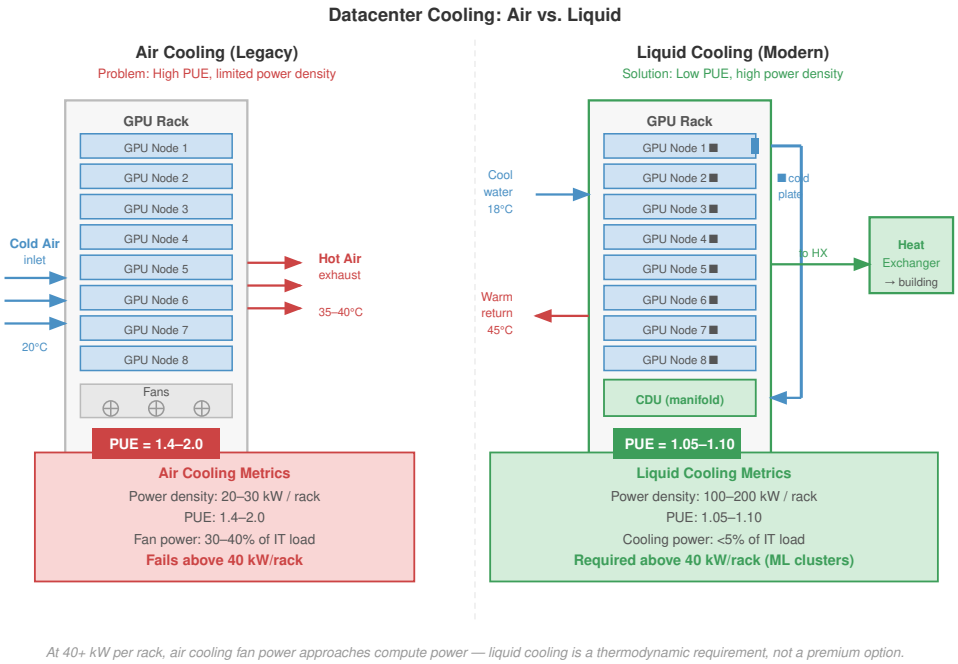


Figure 15.16: Cooling Architectures: Air vs. Liquid: Comparison of thermal management strategies for high-density ML racks. Air cooling (left) requires massive airflow and large fan overhead, reaching a physical limit near 30 kW per rack. Liquid cooling (right) uses direct-to-chip cold plates and high-thermal-capacity coolant, enabling densities exceeding 100 kW per rack with significantly lower PUE.

Savings: Liquid cooling saves 293.9 kW of continuous power. At \$0.07/kWh, the annual savings are approximately \$180,212. Over a 3-year hardware lifecycle, the cooling savings alone total \$540,636, which often exceeds the capital cost of installing the liquid cooling infrastructure.

The final link in the cooling chain is **heat rejection**: getting the heat from the building's chilled water loop to the outside environment. The dominant technology is the cooling tower, which sprays warm water over a fill medium and uses evaporation to carry heat into the atmosphere. Evaporative cooling is remarkably efficient (the latent heat of vaporization of water is 2,260 kJ/kg, compared to 4.18 kJ/kg/K for sensible heating), but it consumes water.

A 10 MW data center with evaporative cooling towers can consume well over 100 million liters of water per year, depending on climate and cooling design, which has become a significant concern in water-stressed regions. To put this in perspective, 140 million liters is roughly the annual water consumption of about 1,000 households. As ML data centers grow to 100 MW and beyond, their water footprint becomes a meaningful factor in local resource planning.

Dry coolers, which use fans to blow air over a radiator without evaporation, eliminate water consumption but work efficiently only when the ambient air temperature is well below the coolant temperature, limiting their effectiveness in hot climates. Many facilities use hybrid approaches: dry coolers during cool weather and evaporative towers during heat waves.

Waste heat reuse treats the data center's thermal output as a resource rather than a waste product. The thermal density of ML accelerator clusters creates an advantage over traditional CPU infrastructure that is easy to overlook: the grade of heat they produce. Traditional air-cooled CPU racks exhaust warm air at roughly 35 degrees Celsius, a temperature range too low for most practical reuse without energy-intensive heat pumps. A liquid-cooled ML cluster, by contrast, returns coolant from its Coolant Distribution Units at 50–65 degrees Celsius—high-grade heat well suited for district heating networks, greenhouse climate control, and industrial process applications. The same acceler-

ator thermal density that demands direct-to-chip cooling thus makes ML racks thermodynamically superior candidates for municipal waste heat programs compared to traditional IT infrastructure. Several Nordic data centers supply their waste heat to municipal heating networks, offsetting the natural gas or electricity that would otherwise be required to heat buildings during winter. A 10 MW ML data center can supply approximately 8–9 MW of useful heat (accounting for heat pump efficiency), enough to heat several thousand apartments. When the economic value of the waste heat is credited against the data center's operating costs, the effective PUE can drop below 1.0, meaning the facility produces more useful energy (computation plus heat) than it consumes from the grid.

The viability of waste heat reuse depends on the proximity of heat consumers. Urban data centers, despite their higher land and electricity costs, are often better positioned for waste heat reuse than remote facilities because they are close to residential and commercial heating loads. The result is a counterintuitive economic optimization: a data center in a Nordic city may have higher electricity costs but lower net operating costs after waste heat revenues, compared to a remote facility with cheaper electricity but no heat consumers nearby.

For our 175B model training cluster, the choice between cooling technologies is not optional. A cluster of 1,000 H100s dissipates 700 kW of heat from the GPUs alone, before accounting for CPUs, memory, networking, and power conversion losses. Only liquid cooling can remove this heat at the required density. The rack is the level at which the problem shifts from *computation* to *physics*, and the design of the cooling infrastructure often determines whether a training cluster can operate at full utilization or must be throttled to prevent thermal runaway.

15.4.6.1 Cooling system reliability

Cooling system failures have more severe consequences in ML clusters than in traditional data centers because of the higher power density. In a traditional air-cooled data center at 10 kW per rack, losing a CRAC unit causes temperatures to rise gradually over tens of minutes, providing ample time for operators to respond. In a liquid-cooled ML rack at 100+ kW, losing coolant flow causes temperatures to reach the GPU's thermal shutdown threshold within 30–60 seconds, because the heat capacity of the cold plate and the small volume of coolant in the pipes provides minimal thermal buffer.

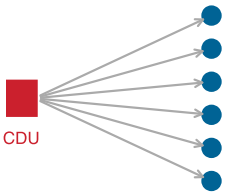
Rapid thermal runaway drives several design decisions. Coolant loops are designed with N+1 redundancy: each CDU has a backup pump, and the piping manifold includes bypass valves that can reroute coolant around a failed CDU. Temperature sensors at each cold plate trigger immediate alerts when the coolant outlet temperature exceeds a threshold (typically 65 degrees Celsius), and the GPU firmware will throttle power within milliseconds if the junction temperature approaches the 83-degree limit.

Some facilities also maintain an emergency air cooling capability as a last-resort backup. Even though air cooling cannot sustain full-power operation at ML rack densities, it can keep the hardware below damage thresholds (at reduced clock speeds) long enough for operators to repair the liquid cooling system. The defense-in-depth approach to cooling reliability reflects that a cooling failure in a 10,000-GPU cluster can simultaneously affect hundreds of GPUs, making the potential financial impact of a cooling outage far greater than the cost of the redundancy.

The failure modes of liquid cooling systems are qualitatively different from those of air cooling. Air cooling fails gracefully: a fan failure reduces airflow, causing temperatures to rise slowly over minutes, providing ample time for automated load shedding. Liquid cooling can fail catastrophically: a coolant leak can simultaneously damage hardware (if the coolant is conductive) and remove cooling capacity (if the leak drains the loop). Quick-disconnect fittings, which allow hot-swapping of server nodes without draining the entire coolant loop, are a critical design feature that reduces maintenance downtime from hours to minutes. However, these fittings are also the most common point of failure in the coolant loop, as the O-ring seals degrade over thousands of connect/disconnect cycles. Facilities that perform frequent hardware swaps (common in research environments where nodes are regularly reconfigured) must budget for quarterly O-ring replacement and maintain a stock of spare fittings.

The economics of cooling reliability shift dramatically when moving from independent inference servers to tightly coupled training clusters. In a distributed training run using synchronous parallelism, a cooling-loop failure that idles a pod-scale slice can halt the entire job. Consider a cooling

failure that triggers a thermal shutdown of a 256-GPU pod within a 10,000-GPU cluster. The direct hardware cost is negligible, but the opportunity cost is immense. If the repair time for a CDU pump is 4 hours, the immediate loss of 256 GPUs at $4 per GPU-hour is only $4,096. However, because the training algorithm requires all workers to proceed in lockstep, the remaining 9,744 GPUs also sit idle, burning electricity without making progress. This straggler effect inflates the cost to $160,000 in lost compute time. When adding the overhead of checkpoint retrieval and the rollback to the last saved state – often losing 30 to 60 minutes of computation – the total financial impact of a single cooling component failure easily exceeds $200,000. The nonlinear scaling of failure costs makes N+1 redundancy in cooling loops a mathematical necessity for training economics.



One cooling failure can idle the whole training job.

Maintaining the physical integrity of the liquid loop requires managing complex hydro-chemical dynamics. The fluid circulating through direct-to-chip systems is typically **deionized water** mixed with specific corrosion inhibitors, not simple tap water. The conductivity must be rigorously maintained below 1 microsiemens per centimeter (μS/cm) to prevent **galvanic corrosion**, where the electrical potential difference between dissimilar metals in the loop (copper cold plates and stainless steel manifolds) eats away at the cooling surfaces. This chemical balance is unstable: inhibitors are consumed over time and dissolved gases accumulate, necessitating monthly quality testing and annual full-volume replacement. Biological contamination poses an equally severe threat. **Biofilm** growth on the internal micro-fins of a cold plate acts as a thermal insulator; a mere 50-micron layer of organic growth can degrade heat transfer coefficients by 30 percent, forcing pumps to run at maximum power to compensate. Regular biocide treatments and periodic system flushing are therefore as critical to cluster performance as driver updates or firmware patches.

15.5 Training vs. Inference Energy Analysis

Cooling infrastructure manages the intense thermal load of the cluster, but the total magnitude of that load depends on whether the fleet is executing a concentrated training run or a globally distributed inference workload. Training a massive language model is a spectacular, highly visible energy event, akin to launching a rocket. Deploying that same model to serve a billion daily queries is like operating an international airline fleet. *Training* burns thousands of megawatt-hours in a single, concentrated burst over several months; *inference* burns energy continuously, query by query, year after year. Understanding where the majority of the energy budget goes dictates where optimization efforts must concentrate.

Optimization opportunities differ across lifecycle phases. Training optimizations focus on computational efficiency and hardware utilization, while inference optimizations emphasize latency, throughput, and edge deployment strategies. Matching the sustainability intervention to the dominant energy consumer for each application yields the greatest returns.

15.5.1 Training energy demands

Training very large AI models can require computational infrastructure with hundreds of thousands of cores and specialized AI accelerators operating continuously for months. Microsoft's 2020 disclosure of the OpenAI dedicated supercomputer, built specifically for large-scale AI training at the time, reported 285,000 CPU cores, 10,000 GPUs, and network bandwidth exceeding 400 gigabits per second per server (Langston 2020). This 2020 figure remains useful as a calibrated reference point rather than a claim about the largest infrastructure available in later generations.

The intensive computational loads generate heat that cooling infrastructure must continuously remove, the overhead quantified for the cooling treatment in section 15.4.6. Reducing it requires co-optimization of hardware architecture, parallelism strategy, and algorithmic efficiency.

Training energy costs occur once per model, but that one-time cost still determines facility sizing, checkpoint storage, and carbon accounting for the run. The primary sustainability challenge often emerges during deployment, where inference workloads continuously serve millions or billions of users and can overtake training energy when request volume is large enough.

15.5.2 Inference energy costs

Inference workloads execute every time an AI model responds to queries, classifies images, or makes predictions. Unlike training, inference scales dynamically and continuously across applications such as search engines, recommendation systems, and generative AI models. Although each individual

inference request consumes far less energy compared to training, the cumulative energy usage from high-volume deployed services can rival or exceed training-related consumption (Wu et al. 2022).

For example, AI-driven search engines handle billions of queries per day, recommendation systems provide personalized content continuously, and generative AI services such as ChatGPT or DALL-E have substantial per-query computational costs. The inference energy footprint is high in transformer-based models due to high memory and computational bandwidth requirements.

Early market forecasts anticipated this shift: a 2017 McKinsey projection (figure 15.17) expected the data-center inference market to roughly double from 4-5 to 9-10 billion dollars and edge inference to climb from near zero to 4-4.5 billion dollars by 2025, both outpacing the slower-growing training market. The stronger evidence is physical rather than economic. The Meta lifecycle measurements in figure 15.14 show inference serving at scale rivaling or exceeding training emissions for deployed recommendation models, and the chapter's own accounting shows continuous serving overtaking a one-time training run once request volume is large enough.

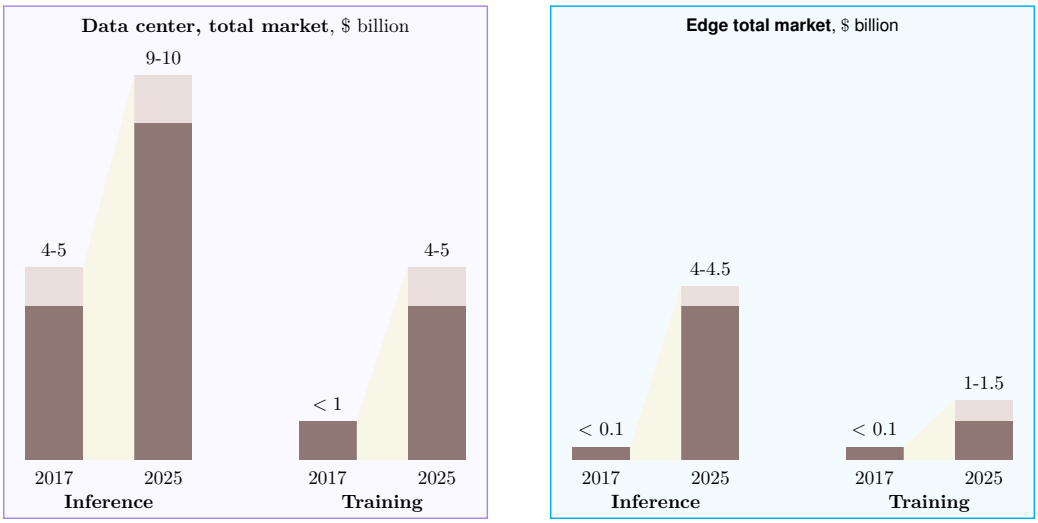


Figure 15.17: AI Hardware Market Growth: McKinsey analysis comparing 2017 market estimates with then-projected 2025 data center and edge markets. Inference workloads dominated the projected growth, with edge inference treated as a significant new segment while training markets grew more gradually in the projection.

Unlike traditional software applications with fixed energy footprints, inference workloads dynamically scale with user demand. AI services like Alexa, Siri, and Google Assistant rely on continuous cloud-based inference, processing millions of voice queries per minute, necessitating uninterrupted operation of energy-intensive data center infrastructure.

15.5.2.1 The energy inefficiency of the decode phase

Chapter 10 introduced prefill and decode as latency phases; sustainability reuses the same split as an energy model. Prefill tends to saturate compute, while decode repeatedly streams model and KV-cache state through memory for each generated token. The serving footprint grows because decode wastes energy differently from prefill, and the gap between the two inference phases is striking (figure 15.18).

The prefill/decode distinction summarized in table 10.20 extends beyond latency into energy efficiency. Recent analysis (Ma and Patterson 2026) reveals that autoregressive generation is inherently energy-wasteful compared to batch processing because the two phases stress different hardware limits. During prefill, high arithmetic intensity allows the GPU to perform thousands of operations for every byte read from memory, achieving near-peak energy efficiency in pJ/FLOP. During decode, the model must read the entire weight set from HBM to generate a single token; arithmetic intensity is low, so the compute units sit idle for much of the cycle.

The result is Static Power Waste: the GPU draws significant leakage and clock power while waiting for memory transfers. Generating 1,000 tokens through 1,000 sequential decode steps can

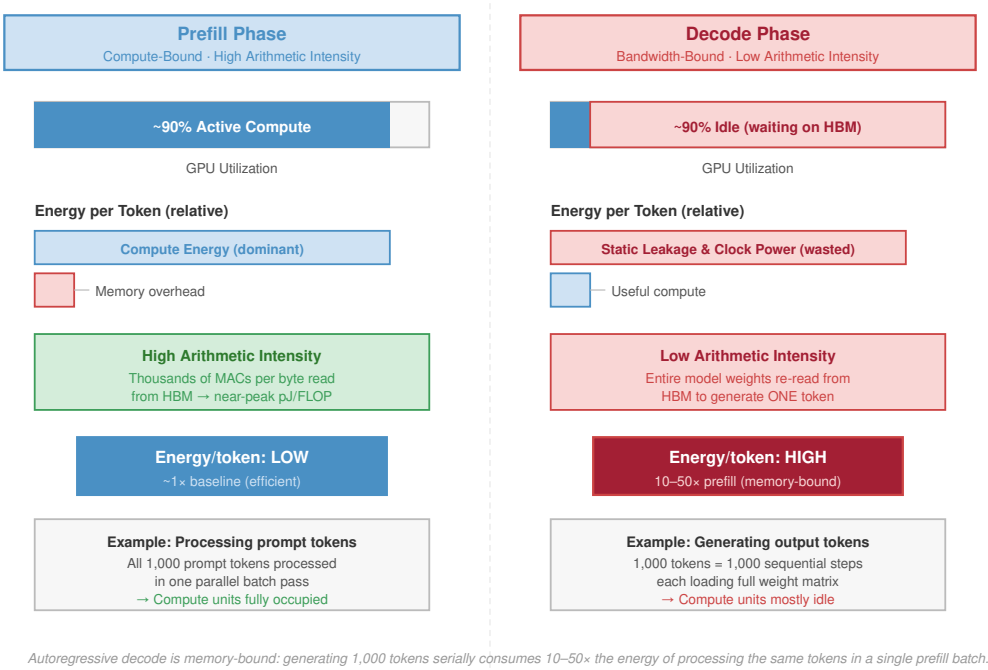


Figure 15.18: Prefill vs. Decode Energy Intensity: Two side-by-side panels comparing prefill and decode. The left panel shows GPU utilization bars across the two phases — high during prefill (compute-saturated) and low during decode (memory-bandwidth bound). The right panel decomposes energy per token into compute and memory components. Decode is 10×–50× less energy-efficient than prefill per operation because compute units sit idle drawing static power while waiting for memory.

therefore consume 10–50× more energy than processing the same 1,000 tokens in a single prefill batch. The inefficiency drives demand for specialized, memory-optimized NPUs and TPUs examined in Chapter 2, which prioritize bandwidth-per-watt over raw TFLOP/s.

15.5.2.2 Edge AI impact

The edge intelligence architectures from Chapter 11 enable inference beyond centralized data centers. This distributed approach offers unique sustainability advantages by reducing data transmission energy costs and lowering dependency on high-power cloud infrastructure. Instead of routing every AI request to centralized cloud servers, models can be deployed directly on user devices or at edge computing nodes.

However, running inference at the edge does not eliminate energy concerns, especially when AI is deployed at scale. Autonomous vehicles, for instance, require millisecond-latency AI inference, meaning cloud processing is impractical. Instead, vehicles use onboard AI accelerators that function as "data centers on wheels" (Sudhakar et al. 2023). These embedded computing systems process real-time sensor data equivalent to small data centers, consuming significant power even without relying on cloud inference.

Similarly, consumer devices such as smartphones, wearables, and IoT sensors individually consume milliwatts to watts of power but collectively add terawatt-hours to global energy use due to their sheer numbers. Therefore, the efficiency benefits of edge computing must be balanced against the extensive scale of device deployment.

Edge deployment can be more sustainable than cloud deployment when designed correctly. The combination of eliminated data transmission, local processing efficiency, and duty-cycled operation can reduce total system energy consumption by orders of magnitude compared to always-connected cloud inference.

15.5.2.3 Edge and mobile power budgets

ARM-based edge devices operate under fundamentally different power constraints than data center accelerators. The engineering choice is to match each inference workload to the smallest power tier that still satisfies latency and accuracy.

Power budgets reflect the physical constraints of battery capacity, thermal dissipation, and deployment environment. Table 15.8 groups edge AI power budget categories and shows how these constraints propagate: TinyML devices operating from coin cells or energy harvesting cannot exceed milliwatt average power, mobile devices must balance user experience with battery life, and automotive systems face thermal constraints within enclosed vehicle compartments despite having access to vehicle power.

Table 15.8: Edge AI Power Budget Categories: Edge platforms span five orders of magnitude in power consumption, from sub-milliwatt TinyML systems to automotive compute platforms approaching data center power levels. Sustainable deployment requires matching workload requirements to appropriate power tiers.

Platform Category	Idle Power	Active Power	Peak Power	Example Devices
TinyML (MCU)	1–100 W	1–50 mW	100 mW	Arduino Nano 33, STM32H7, Nordic nRF5340
Mobile NPU	10-100 mW	0.5–5 W	10 W	Pixel Tensor, Apple Neural Engine, Snapdragon NPU
Edge GPU/TPU	1-5 W	5–30 W	75 W	NVIDIA Jetson Orin NX (10–25 W) and AGX Orin (15–60 W), Google Edge TPU, RPi AI Kit
Autonomous Vehicle	10–50 W	50–200 W	500 W	Tesla FSD Computer, Mobileye EyeQ, NVIDIA Drive

15.5.2.4 TinyML power state dynamics

While Chapter 11 examines TinyML from a systems architecture perspective, the energy efficiency of on-device inference is equally a sustainability consideration: each of the billions of edge inference calls aggregates into measurable carbon footprint at fleet scale. TinyML efficiency depends heavily on duty cycling, where devices alternate between deep sleep and active inference. Equation 15.14 expresses average power as a weighted sum of active and sleep power:

$$P_{\text{average}} = P_{\text{active}} \times \frac{t_{\text{inference}}}{T_{\text{period}}} + P_{\text{sleep}} \times \frac{T_{\text{period}} - t_{\text{inference}}}{T_{\text{period}}} \tag{15.14}$$

For a keyword-spotting model running on a Cortex-M4 microcontroller (Archetype C (Federated MobileNet) regime, section 1.6.1):

- Active inference power: 15 mW for 20 ms per detection cycle
- Deep sleep power: 10 microamps at 3.3V (33 microwatts)
- Detection period: 1 second (continuous listening)

$$P_{\text{average}} = 15\ \text{mW} \times \frac{20\ \text{ms}}{1000\ \text{ms}} + 0.033\ \text{mW} \times \frac{980\ \text{ms}}{1000\ \text{ms}}$$

$$P_{\text{average}} = 0.30\ \text{mW} + 0.032\ \text{mW} = 0.33\ \text{mW}$$

At this average power, a 250 mAh coin cell battery (at 3.0V nominal) provides approximately 2,270 hours of operation, nearly 95 days of continuous always-on AI inference. This calculation demonstrates how TinyML enables sustainable AI deployment scenarios impossible with higher-power platforms. These power-aware design principles carry directly into practical industrial deployment scenarios.

Example 15.5: Battery life for TinyML

Consider deploying an anomaly detection model on a factory sensor node:

System parameters:

- Model: Autoencoder for vibration anomaly detection
- MCU: ARM Cortex-M4 at 80 MHz
- Inference latency: 5 ms per sample
- Sampling rate: 10 Hz (100 ms period)
- Active power: 12 mW during inference
- Sleep power: 5 microamps at 3.3V (16.5 microwatts)
- Battery: two AA cells (3000 mAh at 3.0V)

Step 1: Calculate duty cycle and average power.

$$\delta_{\text{duty}} = \frac{5 \text{ ms}}{100 \text{ ms}} = 0.05 \text{ (5\% duty cycle)}$$

$$P_{\text{avg}} = 12 \text{ mW} \times 0.05 + 0.0165 \text{ mW} \times 0.95 = 0.60 + 0.016 = 0.616 \text{ mW}$$

Step 2: Calculate battery life.

$$E_{\text{battery}} = 3000 \text{ mAh} \times 3.0 \text{ V} = 9000 \text{ mWh}$$

$$t_{\text{life}} = \frac{9000 \text{ mWh}}{0.616 \text{ mW}} = 14,610 \text{ hours} \approx 1.7 \text{ years}$$

Systems insight: The deployment achieves continuous AI-powered monitoring for nearly two years on standard batteries, demonstrating the sustainability potential of TinyML systems designed with power-aware principles.

15.5.2.5 On-device learning and the battery wall

While inference on TinyML devices is highly efficient, on-device learning introduces a much steeper energy challenge. Personalizing a model to a user's specific voice or gait requires backpropagation, which demands 2–3× more compute and memory than forward inference.

The thermal design power (TDP) of mobile processors creates hard constraints that shape every aspect of on-device learning strategies. Modern smartphones typically maintain sustained processing at 2–3 W for ML workloads to prevent thermal discomfort, but can burst to 5–10 W for brief periods before thermal throttling occurs. This thermal design power determines the entire feasible space of adaptive algorithms.

Napkin Math 15.6: The energy of learning

Problem: Consider fine-tuning a small language model (1B parameters) on a user's smartphone overnight. Is this feasible within a 5 percent battery budget?

Math:

1. **Phone Battery**: Typical capacity is approximately 15 Wh, or about 54000 J.
2. **Budget**: 5 percent of 54000 J = 2700 J.
3. **Training Cost**:
   - Forward pass: $\approx$ 2 nJ/param.
   - Backward pass: $\approx$ 4 nJ/param.
   - Total per token: 6 nJ/param $\times 10^9$ params = 6 J/token.

4. **Capacity**: 2700 J / 6 J/token = 450 tokens.

Systems insight: Full fine-tuning is impossible within a reasonable daily battery budget. Sustainable on-device learning requires parameter-efficient fine-tuning (PEFT) or sparse updates to reduce the energy cost per token by 100× or more.

The fundamental physics of energy consumption reveals why local processing is almost always preferable to cloud offloading for on-device learning, provided the model is sufficiently compact.

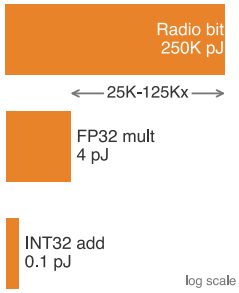
Systems Perspective 15.4: The energy hierarchy

Trade-off: The architectural choice between processing data locally and sending it to the cloud is governed by an energy budget. The physics of energy consumption provides a clear answer based on the energy-to-communication ratio.

Energy cost per operation (approximate):

- **32-bit integer add**: 0.1 pJ
- **32-bit float mult**: 4 pJ
- **Wireless transmit (1 bit)**: 100,000–500,000 pJ (Bluetooth/Wi-Fi)

Systems insight: Transmitting a single bit of data costs roughly the same energy as performing 25,000 to 125,000 FP32 multiplies, or 1 million to 5 million 32-bit integer adds, under these operation-cost assumptions. When insight can be extracted from data using fewer than roughly 100,000 floating-point operations per bit, local processing is usually more energy efficient than cloud offloading. This ratio drives the architecture of federated learning: *compute is cheap; radio transmission is expensive*.



Radio transmission dwarfs local arithmetic in energy.

15.5.2.6 Energy harvesting for autonomous edge AI

With sufficient optimization, TinyML enables energy-autonomous operation where devices harvest ambient energy rather than relying on batteries:

Consider the energy harvesting power budgets in table 15.9: a keyword spotting model optimized to 0.5 mW average power can operate indefinitely on approximately 5 cm² of indoor solar harvesting only under bright indoor conditions near the top of the listed range. Typical indoor deployments need additional area, energy storage, duty cycling, or a lower average-power model to leave margin for conversion losses and dim lighting. This perpetual operation model represents the ultimate sustainable edge AI deployment, where operational energy comes entirely from ambient sources.

Table 15.9: Energy Harvesting Power Budgets: Ambient energy harvesting enables batteryless TinyML deployments when average power consumption remains within harvesting capacity. Solar harvesting provides the highest power density for most deployments.

Harvesting Source	Typical Power	Viable TinyML Applications
Indoor solar (1 cm²)	10-100 microwatts	Periodic sensor classification
Outdoor solar (1 cm²)	1-10 milliwatts	Continuous keyword spotting
Thermoelectric (body heat)	10-100 microwatts	Wearable gesture recognition
RF harvesting (Wi-Fi)	1-10 microwatts	Ultra-low-duty sensor nodes
Vibration piezoelectric	100 microwatts-1 mW	Industrial monitoring

15.5.2.7 Cascade inference architecture

Beyond individual device efficiency, architectural patterns determine total system energy consumption across edge-cloud boundaries. A cascade architecture deploys a small edge model (under 100

KB) to filter inputs before cloud inference. Equation 15.15 expresses total energy as the sum of local processing plus probabilistically-triggered cloud costs:

$$E_{\text{cascade}} = E_{\text{edge}} + p_{\text{escalate}} \times (E_{\text{transmit}} + E_{\text{cloud}}) \tag{15.15}$$

where p_{escalate} is the probability of requiring cloud inference (typically 5–20 percent for well-designed cascades).

For a visual inspection system:

- Edge model (MobileNet-v3 tiny): 0.5 mJ per image classification
- Cloud model (ResNet-152): 50 mJ per classification
- Transmission energy: 10 mJ per image (cellular)
- Escalation rate: 10 percent (only ambiguous cases sent to cloud)

$$E_{\text{cascade}} = 0.5 + 0.10 \times (10 + 50) = 0.5 + 6.0 = 6.5 \text{ mJ/image}$$

Compared to always-cloud inference at 60 mJ per image, the cascade architecture achieves 89 percent energy reduction while maintaining accuracy through selective cloud escalation.

15.5.2.8 Wake-word triggered systems

Always-on systems use hierarchical wake detection to minimize average power:

1. Ultra-low-power analog front end: 10 microwatts continuous voice activity detection
2. Tiny neural network wake detector: 100 microwatts when speech detected
3. Full model inference: 10 mW for 50 ms when wake word confirmed

With typical speech activity rates of 5 percent and wake word occurrence of 0.1 percent:

$$P_{\text{average}} = 0.01 + 0.05 \times 0.1 + 0.001 \times 10 \times 0.05 = 0.0155mW$$

The hierarchical approach achieves 15.5 µW average power compared to 10 mW for always-active full inference, a 645.2× reduction enabling battery-powered voice assistants with multi-year operation.

15.5.2.9 Federated learning energy analysis

Training at the edge eliminates data transmission but increases local compute. Equation 15.16 contrasts the energy trade-offs between federated and centralized approaches:

$$\begin{aligned} E_{\text{federated}} &= N_{\text{clients}} \times E_{\text{local\_train}} + E_{\text{aggregation}} \\ E_{\text{centralized}} &= N_{\text{clients}} \times E_{\text{transmit}} + E_{\text{cloud\_train}} \end{aligned} \tag{15.16}$$

Federated learning becomes more energy-efficient when data sizes exceed model update sizes. For privacy-sensitive applications with rich sensor data, federated approaches often achieve both privacy *and* energy benefits, as transmitting model weight updates (megabytes) requires less energy than transmitting raw data (gigabytes) for applications like on-device personalization. The edge analysis leaves one more lifecycle term: the physical supply chain that produces the devices and accelerators.

15.5.3 Water, chemicals, and critical materials

AI's environmental footprint extends beyond electricity consumption to include physical resources—water, hazardous chemicals, and critical materials—that require different assessment approaches. Comprehensive assessment requires measuring additional ecological impacts including water consumption, hazardous chemical usage, rare material extraction, and biodiversity disruption that often receive less attention despite their ecological significance. Modern semiconductor fabrication plants producing AI chips require millions of liters of water daily and use over 250 hazardous substances in their processes. In regions already facing water stress, such as Taiwan, Arizona, and Singapore, this intensive usage threatens local ecosystems and communities. AI hardware also relies heavily

on scarce materials like gallium, indium, arsenic, and helium, which face both geopolitical supply risks and depletion concerns (Jha 2014; Chen 2006).

Semiconductor fabrication is an exceptionally water-intensive process (Cooper et al. 2011). TSMC's fab in Arizona is projected to consume 34 million liters of water per day[29] (Reuters 2024), accounting for nearly 3 percent of the city's total water production. A single 300mm silicon wafer requires over 8,300 liters of water throughout the complete fabrication process. Figure 15.19 illustrates the typical fab water cycle, where advanced recycling can reclaim 60–80 percent of water but still leaves a substantial consumption footprint.

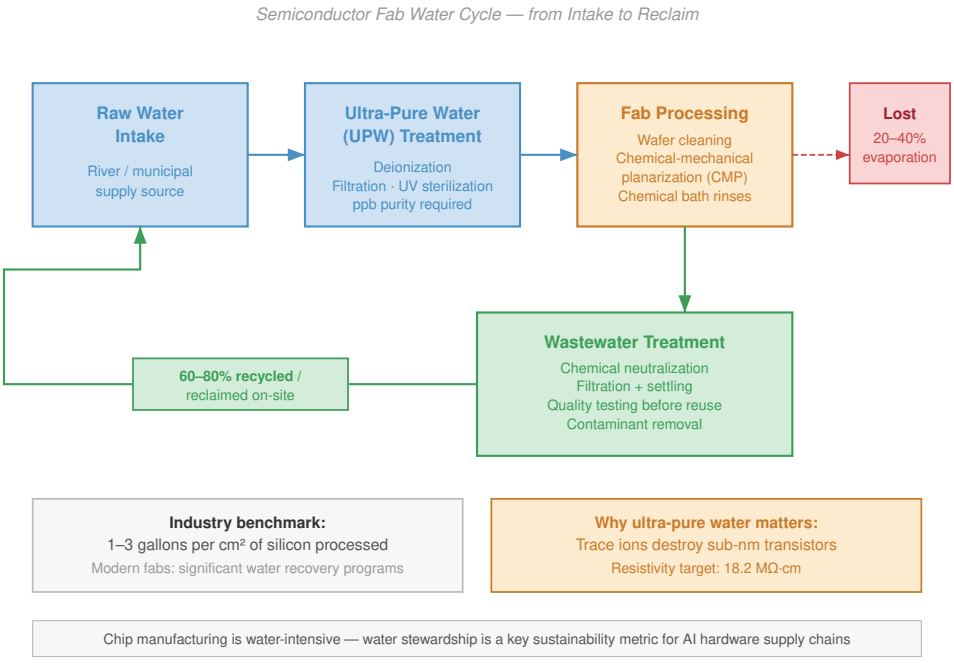


Figure 15.19: Semiconductor Water Cycle: Modern fabs consume millions of liters of water daily. To mitigate this, advanced facilities implement closed-loop recycling. Raw water is purified to Ultra-Pure Water (UPW) for processing. Wastewater is treated and recycled back into the UPW system, reducing net consumption by 60–80 percent.

The critical takeaway from figure 15.19 is that even with 60–80 percent reclamation rates, the absolute volume of ultra-pure water consumed by advanced-node fabs remains enormous, creating a hard physical constraint on where AI chip manufacturing can sustainably operate.

Fabrication is also heavily reliant on hazardous chemicals for etching, doping, and cleaning. Strong acids (hydrofluoric, sulfuric), volatile organic compounds like xylene, and highly toxic gases (arsine, phosphine) are used in massive quantities—a large fab may consume over 2,000 metric tons of acids annually (S. Kim et al. 2018). These substances create hazardous waste streams requiring extensive treatment to prevent ecological harm.

AI hardware depends on a suite of scarce and geopolitically sensitive **critical materials**. While silicon is abundant, high-performance chips require rare elements like gallium, indium, tantalum, and helium. Materials such as indium appear in critical-materials and endangered-elements analyses because supply can depend on byproduct extraction, substitution options, and recycling constraints (Rhodes 2019). The geographic concentration of rare earth refining creates significant supply chain vulnerabilities. Table 15.10 quantifies the scope of this material dependency challenge.

29 **Semiconductor Water Scale**: TSMC's Arizona fab will consume 12 billion liters annually (about 4,800 Olympic pools), and advanced-node AI chips require 5–10× more water per die than older process nodes due to additional EUV and cleaning steps. This water dependency creates a direct sustainability constraint: fabs compete with municipal water supplies in drought-prone regions like Arizona and Taiwan, where semiconductor water demand can reach 3 percent of a city's total allocation.

Table 15.10: Critical Materials for AI Hardware: Semiconductor manufacturing relies on specific materials, including silicon, gallium, germanium, indium, tantalum, cobalt, tungsten, copper, helium, and rare earth elements, that face increasing supply constraints and geopolitical risks, potentially impacting AI hardware production and innovation. The table details these materials, their applications in AI systems, and the associated supply vulnerabilities requiring proactive mitigation strategies.

Material	Application in AI Semiconductor Manufacturing	Supply Concerns
Silicon (Si)	Primary substrate for chips, wafers, transistors	• Processing constraints • Geopolitical risks
Gallium (Ga)	GaN-based power amplifiers, high-frequency components	• Limited availability • Byproduct of aluminum and zinc production
Germanium (Ge)	High-speed transistors, photodetectors, optical interconnects	• Scarcity • Geographically concentrated
Indium (In)	Indium Tin Oxide (ITO), optoelectronics	• Limited reserves • Recycling dependency
Tantalum (Ta)	Capacitors, stable integrated components	• Conflict mineral • Vulnerable supply chains
Rare Earth Elements (REEs)	Magnets, sensors, high-performance electronics	• High geopolitical risks • Environmental extraction concerns
Cobalt (Co)	Batteries for edge computing devices	• Human rights issues • Geographical concentration (Congo)
Tungsten (W)	Interconnects, barriers, heat sinks	• Limited production sites • Geopolitical concerns
Copper (Cu)	Interconnects, barriers, heat sinks	• Limited high-purity sources • Geopolitical concerns
Helium (He)	Semiconductor cooling, plasma etching, EUV lithography	• Nonrenewable • Irretrievable atmospheric loss • Limited extraction capacity

The construction and operation of fabs and data centers also directly impacts natural ecosystems through habitat disruption, water stress, and pollution from chemical discharge. Semiconductor-industry effluents can contaminate nearby fluvial sediments with heavy metals and trace elements (Hsu et al. 2016). Waste generation from fabrication—including gaseous emissions, VOC-laden air, and metal-contaminated wastewater—requires advanced treatment systems, and the end-of-life disposal of AI hardware contributes to a growing e-waste crisis, with only 17.4 percent of global e-waste properly recycled (Singh and Ogunseitan 2022).

The environmental toll of computational demand extends far beyond atmospheric carbon, manifesting as severe water stress and ecological disruption around manufacturing hubs. These supply-chain costs converge on the next design lever: how long massive, resource-intensive hardware clusters remain useful before they become waste.

15.6 Hardware Lifecycle and E-Waste

The environmental cost of an AI accelerator begins long before its first FLOP is calculated (Gupta et al. 2022; Luccioni et al. 2023; NVIDIA Corporation 2025). The per-H100 manufacturing footprint quantified in section 15.3.1.1 is incurred entirely at fabrication.[30] The fleet of thousands of such processors required to train our 175B parameter model—consuming 1,287 MWh of electricity—represents a significant upfront carbon investment before any computation occurs. A comprehensive **Life Cycle Assessment (LCA)** quantifies the cumulative environmental impact across four key phases: design, manufacture, use, and disposal. LCA can reveal manufacturing as a major share of lifecycle impact, making it a critical sustainability lever that operational efficiency improvements alone cannot address.

30 **Life Cycle Assessment (LCA)**: Standardized by ISO 14040/14044, LCA traces environmental impact from raw material extraction through disposal (International Organization for Standardization 2006a, 2006b). For AI hardware, LCA separates embodied manufacturing emissions from operational energy use, making hardware refresh cycles and accelerator lifespan extension first-order sustainability levers that operational efficiency alone cannot substitute.

Checkpoint 15.4: The training-inference flip

Consider a vision model where training requires 2,000 GPU-hours at an average power draw of 300 W. Once deployed, the model serves 1 million requests per day, with each request taking 50 ms at an average draw of 100 W.

- ☐ **Training energy**: Calculate the total energy used for training.
- ☐ **Inference energy**: Calculate the total energy used for inference over a 2-year product lifespan.
- ☐ **Optimization focus**: Use the "Inference-to-Training Ratio" to select the optimization target that maximizes sustainability.

Life Cycle Assessments can show that discarding functional hardware purely for modest efficiency gains causes more environmental harm through embodied carbon than it saves in operational power. To evaluate the tipping point where new hardware becomes environmentally justified, we must estimate the intersection of training costs, inference scale, and hardware lifespans.

Each of the four primary lifecycle stages contributes to an AI system's total environmental footprint. Figure 15.20 visualizes this progression from design through disposal, highlighting the interdependencies between phases and the environmental impact categories associated with each stage.

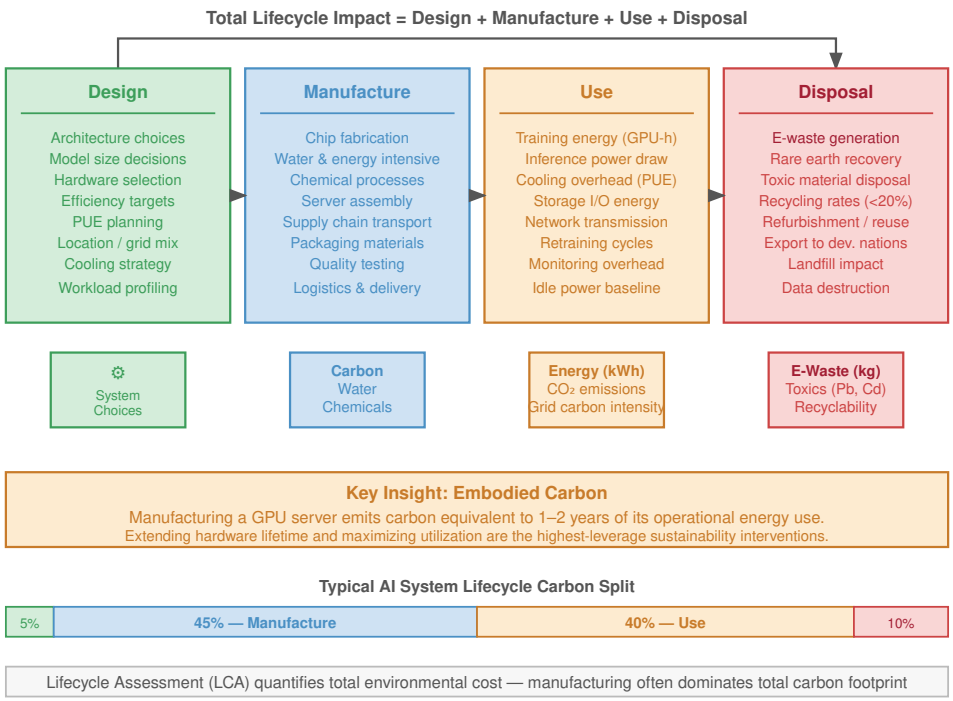


Figure 15.20: AI System Lifecycle: Analyzing AI systems across design, manufacture, use, and disposal stages exposes the full environmental impact beyond operational energy consumption, encompassing resource depletion and electronic waste. This lifecycle assessment allows targeted interventions to improve sustainability throughout the entire AI system's existence.

The lifecycle sequence is not a static taxonomy. The binding sustainability problem shifts as the system matures: design creates experimental waste, manufacturing locks in embodied carbon, use couples the workload to grid and cooling constraints, disposal externalizes e-waste, and lifespan extension becomes the lever that amortizes all earlier emissions over more useful work.

15.6.1 Design and experimentation phase

The design phase encompasses the research, development, and optimization of ML models before deployment—iterating on architectures, tuning hyperparameters, and running training experiments.

The environmental cost of this phase is often underestimated because reported training energy (such as GPT-3's 1,287 MWh) reflects only the final run, not the extensive trial-and-error that preceded it. Automated architecture search techniques evaluate hundreds or thousands of configurations, but their carbon footprint depends on what is counted: proxy search, full training runs, data-center efficiency, grid mix, and reuse across downstream models (Strubell et al. 2019; Patterson et al. 2021). Later efficient NAS methods use weight sharing, continuous relaxations, and hardware-aware search to reduce the search budget (Elsken et al. 2019). Table 15.11 reveals stark differences in model carbon footprint across model scales.

Table 15.11: Model Carbon Footprint: Training large AI models can generate substantial carbon emissions, but the final footprint depends on computation, hardware efficiency, data-center overhead, and grid carbon intensity. GPT-3's training emissions are equivalent to driving 1.9 million km under the accounting assumptions used here.

AI Model	Training FLOPs	Estimated CO_2 Emissions (kg)	Equivalent Car Distance
GPT-3	3.1×10^{23}	502,000 kg	1.9 million km
T5-11B	2.3×10^{22}	85,000 kg	338,000 km
BERT (Base)	3.3×10^{18}	650 kg	2,400 km
ResNet-50	2.0×10^{17}	35 kg	129 km

Addressing the design phase's sustainability challenges requires innovations in training efficiency: sparse training, low-precision arithmetic, weight-sharing, and energy-aware NAS approaches. Transfer learning and fine-tuning pretrained models reuse pretrained representations instead of requiring every task to be trained from scratch (Raffel et al. 2020).

15.6.2 Manufacturing phase

The manufacturing of AI hardware is enormously resource-intensive, carrying the per-H100 embodied carbon established in section 15.3.1.1 before any computation occurs. Semiconductor fabrication requires extreme precision through processes such as EUV lithography—each tool consuming approximately 1 MW of continuous power—chemical vapor deposition, and ion implantation. The resource demands detailed in section 15.5.3 reveal the scale: TSMC's Arizona fab consumes 34 million liters of water daily, fabrication relies on over 250 hazardous substances, and the supply chain depends on geopolitically concentrated critical materials.

Two structural properties of AI accelerators amplify this manufacturing footprint relative to conventional chips. First, high-performance AI chips are typically fabricated at or near the reticle limit—the maximum die area a single lithography exposure can print. Larger dies yield fewer chips per wafer and are disproportionately vulnerable to random defects: a defect density that kills 5 percent of small dies may kill 20 percent or more of a reticle-limit die, effectively wasting the water, chemicals, and EUV energy consumed in fabricating every defective unit. Second, the memory bandwidth requirements of large-scale inference and training demand advanced 2.5D and 3D packaging—such as TSMC's Chip-on-Wafer-on-Substrate (CoWoS) process—that bonds High Bandwidth Memory stacks directly to the accelerator die. This integration introduces additional fabrication stages, chemical cleaning cycles, and precision baking steps that a conventional monolithic CPU does not require. The result is that each viable AI accelerator embodies substantially more water consumption, hazardous chemical use, and process energy than its wafer area alone would suggest, making hardware longevity and high utilization first-order sustainability levers rather than secondary concerns.

The energy required to manufacture AI hardware is substantial, and in clean-grid regions embodied manufacturing impacts can rival operational impacts over useful life. Research on eco-friendly electronics points to lower-toxicity materials, improved recycling, and greener device and manufacturing approaches as sustainability directions for electronics production (Cenci et al. 2021; Irimia-Vladu 2014).

15.6.3 Use phase

The operational energy consumed during training and inference is detailed in section 15.5. What merits attention here is the pattern of this consumption and its interaction with grid infrastructure.

The 1,287 MWh required to train our 175B model represents a high, continuous power draw—but the character of that draw is not uniform across workload types, and that distinction shapes what scheduling interventions are practical.

A large distributed training run draws constant, correlated power across thousands of accelerators and is often described as inflexible. This characterization understates the scheduling flexibility that distributed training already builds in for fault tolerance. Because a single hardware failure in a thousand-node cluster can corrupt a run, production training systems checkpoint state to persistent storage every few minutes to hours. That checkpointing infrastructure is exactly the mechanism that enables carbon-aware scheduling: a training job can be paused during the late-afternoon grid peak (when solar generation drops and fossil peaker plants come online), with cluster state saved to checkpoint, and resumed when overnight wind generation raises renewable availability. The same engineering that protects against hardware failures thus doubles as a scheduling lever for grid decarbonization—a coupling of fault tolerance and sustainability that has no parallel in synchronous inference serving, which cannot be paused mid-request.

This flexibility stands in contrast to the duck curve problem that training clusters do exacerbate when they run continuously. The **duck curve** describes the steep ramp that grid operators must cover as solar power drops in the late afternoon: a data center pulling constant megawatts through the transition period deepens that ramp and increases reliance on fossil peaker plants. Cooling systems compound the problem, adding the overhead quantified in section 15.4.6 on top of the computational draw. Carbon-aware schedulers that exploit checkpointing to pause training through the high-carbon window—typically the two to four hours straddling the solar-to-peaker transition—can shift a meaningful fraction of training energy consumption to periods when the marginal grid carbon intensity is lower. Geographic optimization, as discussed in section 15.3, addresses the baseline, but temporal scheduling addresses the variation within a given grid region.

15.6.4 Disposal, e-waste, and embedded AI

The rapid pace of innovation in AI hardware creates a relentless upgrade cycle (Slade 2007), contributing to a growing global crisis of electronic waste (e-waste). Globally, humanity generates over 50 million metric tons of e-waste annually, of which only 17.4 percent is formally documented as collected and properly recycled (Singh and Ogunseitan 2022). The high-performance servers used for training large models have a typical service life of just three to five years before they are considered obsolete. Discarded AI hardware contains toxic materials—lead, mercury, cadmium, and beryllium—that can leach into soil and groundwater when disposed of in landfills or informal recycling facilities (Grossman 2007).

Two mechanisms specific to ML infrastructure make this obsolescence faster and more wasteful than in traditional server environments. AI accelerators rarely reach the end of their physical silicon lifespan; instead, they become obsolete due to memory bandwidth and interconnect bottlenecks that new model architectures expose. A cluster of GPUs connected by PCIe Gen 4 may have perfectly functional compute silicon but fall below the minimum interconnect bandwidth required to sustain the collective communication patterns of a newer, larger model. Because the interconnect is embedded in the baseboard rather than the chip, the entire node—not just the accelerator—must be replaced. Unlike conventional CPUs that socket into standardized ATX or OCP motherboards, modern AI nodes mount accelerators on proprietary baseboards engineered around a specific generation of NVLink, NVSwitch, and HBM. When a fleet upgrade targets a new model architecture, the baseboard, networking host channel adapters, and often the host server are replaced together, rather than simply swapping a PCIe card. This architectural tight-coupling multiplies the mass of e-waste generated per upgrade cycle far beyond what the per-chip silicon accounts suggest.

The problem is compounded by the rise of **embedded AI**, where machine learning capabilities are integrated into billions of consumer devices. Figure 15.21 traces the connected-device population from 8.6 billion in 2019 to a projected 29.42 billion by 2030, a more than threefold rise over the decade with the historical-to-projected boundary falling around 2024 (Statista 2022). That trajectory creates a distributed, low-value, and exceptionally difficult-to-recycle form of e-waste. Many AI-powered IoT sensors, wearables, and smart appliances are built with short lifespans and limited upgradability, making them difficult or impossible to repair or recycle (Baldé et al. 2017). Nonreplaceable lithium-

ion batteries, sealed enclosures, and proprietary components ensure that even minor failures lead to complete device replacement.

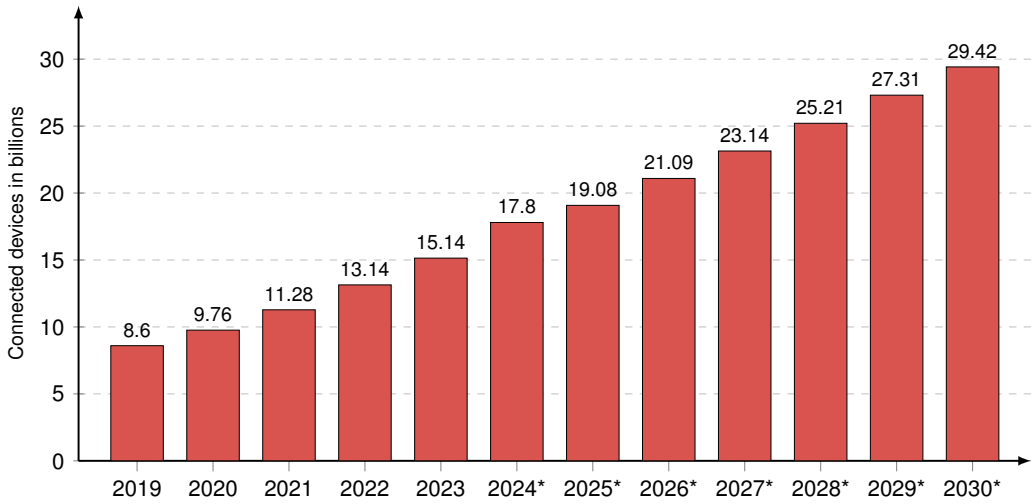


Figure 15.21: IoT Device Growth: Rapid expansion in the number of connected devices amplifies the environmental impact of embedded AI systems, as short device lifecycles contribute to escalating electronic waste. Projections approaching 30 billion devices by 2030 necessitate sustainable design and improved recycling infrastructure to mitigate the growing e-waste crisis.

Short product lifecycles accelerate the cycle: limited software support windows, proprietary components that prevent repair, and sealed designs that make disassembly difficult all push devices toward replacement instead of reuse. A disproportionate share of this e-waste burden falls on developing nations, which often receive shipments of discarded electronics from wealthier countries, leading to significant environmental and social costs for populations least equipped to manage them.

15.6.5 Extending hardware lifespan

Countering the linear "take-make-dispose" model requires a shift toward a **circular economy** (Stahel 2016) that prioritizes reuse, refurbishment, and recycling. When embodied carbon dominates the lifecycle account, extending the functional lifespan of AI hardware is one of the largest reduction levers because it amortizes high manufacturing emissions over a longer period. Extending server life from three to five years reduces embodied carbon per year of service by 40 percent, a gain that can exceed many local software optimizations.

Four lifecycle interventions extend hardware service life by making systems repairable, upgradeable, supported, and reusable:

- **Right-to-repair**: Legislative and regulatory movements push back against repair restrictions by emphasizing access to parts, tools, diagnostics, and service information (Federal Trade Commission 2021).
- **Modular design**: AI hardware designs that allow independent upgrade of accelerators, memory, or networking interfaces prevent entire systems from being discarded when only one component is obsolete, following the principle demonstrated by companies like Framework in consumer laptops (Incorporated 2022).
- **Extended support cycles**: Longer software and firmware support keeps usable hardware secure and operational for longer, delaying its entry into the e-waste stream (Forti et al. 2020).
- **Secondary-use programs**: Moving older accelerators into research, batch, or lower-priority workloads further amortizes embodied carbon instead of sending hardware directly to disposal.

These interventions move sustainability from disposal management to lifecycle engineering. Once the scale of the hardware, energy, and carbon footprint generated by AI systems is quantified, the question becomes what specific engineering techniques can reduce this impact.

15.7 Mitigation Strategies

When a data center hits its absolute power ceiling, the operator *cannot* simply buy more GPUs. The only path forward is extracting more intelligence from every watt through algorithmic intervention: quantizing FP32 weights down to INT4, pruning inactive neural pathways, and scheduling training runs to execute precisely when the local power grid is flooded with excess solar energy. Mitigation is the process of treating energy efficiency as a core algorithmic constraint.

The measurement frameworks developed in preceding sections revealed where environmental costs concentrate: training dominates for research workloads, inference dominates for deployed services, and manufacturing contributes a baseline that operational efficiency cannot eliminate. The findings guide implementation strategy along three axes: algorithmic optimization reduces per-operation costs, infrastructure choices determine whether those savings translate to actual emissions reduction, and policy frameworks ensure industry-wide adoption.

Implementation must account for Jevons Paradox[31] (principle 19): making models 10× more efficient can increase total usage enough to erase or even exceed the expected energy savings, because cheaper computation enables entirely new applications that were previously economically infeasible. This **rebound effect** is why sustainability strategies must focus on absolute limits (carbon budgets, renewable sourcing) rather than just rate efficiency (FLOP/s per watt), combining technical optimization with usage governance that prevents efficiency gains from being offset by exponential growth in deployment scale.

[31] **Jevons Paradox**: Named after Jevons (1865), who observed that James Watt's more efficient steam engine increased total coal consumption by making steam power economically viable for new applications. The pattern recurs in AI: making inference 10× cheaper enables 100× more applications (chatbots, code assistants, real-time translation), producing a net increase in total energy. This is why per-query efficiency alone cannot guarantee sustainability without usage governance.

15.7.1 Multi-layer mitigation strategy framework

The most counterintuitive obstacle to sustainable AI is not inefficiency but success, which is why mitigation must choose both which layer owns each reduction and which absolute budget prevents rebound. Energy-efficient model design, optimized hardware deployment, sustainable infrastructure operations, and carbon-aware scheduling each attack a different term in the lifecycle footprint. Framework selection matters only insofar as it changes computation, memory movement, utilization, or reporting. Lifecycle-aware design keeps the optimization from ending at deployment by checking whether savings survive training, inference, manufacturing, and use.

Figure 15.22 captures this effect: as the cost per unit of computation drops, usage rises faster than efficiency brings the per-unit cost down, so total consumption, and environmental impact, rises rather than falls.

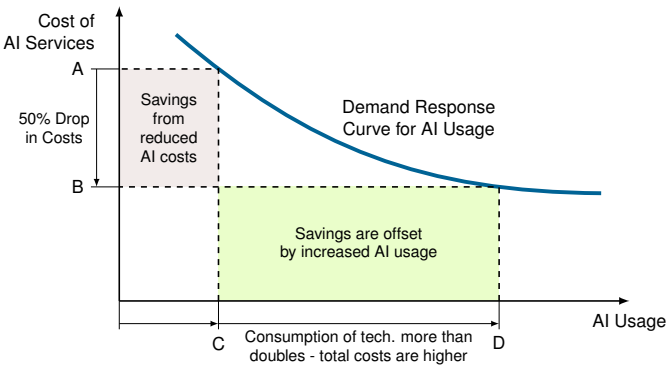


Figure 15.22: Jevons Paradox: Decreasing computation costs drive increased AI usage, potentially offsetting efficiency gains and leading to higher overall resource consumption; the figure maps this effect, showing how a cost reduction (A to B) fuels demand growth (C to D). This counterintuitive relationship underscores the importance of considering systemic effects when evaluating the environmental impact of AI advancements.

The paradox has profound implications for sustainable AI strategy because total energy depends on both per-query cost and demand elasticity.

Checkpoint 15.5: The efficiency trap (Jevons Paradox)

Your team optimizes a translation service, reducing the computational cost per query by 50 percent (2× efficiency gain).

- ☐ **Inelastic demand**: Calculate the change in total energy consumption when demand is inelastic and price change does not affect usage.
- ☐ **Elastic demand**: Calculate the net change in total energy consumption when a 50 percent cost reduction leads to a 300 percent increase in query volume.
- ☐ **Rebound effect**: Define how this "rebound effect" challenges the assumption that "efficient models are automatically green models."

Jevons Paradox does not invalidate efficiency as a strategy; it simply means that efficiency must be paired with governance and capacity planning. At the level of individual systems, efficiency remains a central lever because it directly reduces cost, latency, and energy per useful operation.

Systems Perspective 15.5: Efficiency as sustainability

Every model optimization technique is simultaneously a sustainability tool. Pruning reduces computational complexity and energy consumption by eliminating unnecessary parameters. Quantization decreases memory requirements and accelerates inference while cutting power consumption. Knowledge distillation enables smaller models to achieve competitive performance with lower resource demands.

Performance engineering and environmental responsibility converge on the same objective. Optimizing a model to run faster or use less memory simultaneously reduces its carbon footprint. Designing efficient architectures or implementing hardware-software co-design produces systems that are both high-performing and environmentally sustainable.

The fundamental insight is that sustainable AI engineering overlaps strongly with efficient AI engineering, but extends beyond it. The engineering principles that enable systems to scale, perform better, and cost less to operate also make them more environmentally responsible, but sustainability adds lifecycle accounting, carbon-aware placement, absolute resource budgets, water and materials constraints, and governance against rebound effects. Sustainability is an integral part of good systems engineering, not a synonym for efficiency alone.

15.7.2 Lifecycle-aware development methodologies

Lifecycle-aware development starts from the largest environmental term in the workload and then selects the intervention that changes that term. Algorithmic design, infrastructure optimization, operational practice, and governance reduce impact only when they are sequenced around the measured bottleneck rather than applied as a generic checklist (Uddin and Rahman 2012).

15.7.2.1 Energy-efficient algorithmic design

Many deep learning models rely on billions of parameters, requiring trillions of FLOPs during training and inference.[32] While these large models achieve top benchmark scores, research indicates that much of their computational complexity is unnecessary. Many parameters contribute little to final predictions, leading to wasteful resource consumption. Sustainable AI development treats energy efficiency as a design constraint rather than an optimization afterthought, requiring hardware-software co-design approaches that simultaneously optimize algorithmic choices and their hardware implementation for maximum efficiency per unit of computational capability.

When the measured bottleneck is unused structure, bit width, or serving scale, the sustainable design lever changes. Table 15.12 maps the bottleneck to the intervention that changes the energy term.

32 **FLOP/s vs. FLOPs**: FLOP/s denotes a rate (operations per second); FLOPs denotes an operation count. The distinction matters for sustainability because energy scales with FLOPs (count), not FLOP/s (rate). GPT-3 required 3.1×10^{23} FLOPs total, and the energy cost per operation spans a 1000× range: CPUs at ~100 pJ/FLOP, GPUs at ~10 pJ/FLOP, TPUs at ~1 pJ/FLOP, and custom ASICs approaching 0.1 pJ/FLOP.

Table 15.12: Algorithmic Energy Levers by Bottleneck: Pruning, quantization, and distillation reduce energy through different mechanisms, so the right choice depends on the measured bottleneck rather than on a generic compression goal.

Technique	Measured bottleneck	Energy mechanism	Representative evidence
Pruning	Unused model structure	Removes redundant weights, reducing model size, compute, and memory movement during inference	Structured pruning can remove up to 90% of weights in models such as ResNet-50 while maintaining comparable accuracy
Quantization	Bit width	Lowers numerical precision so arithmetic units and memory transfers move fewer bits	INT8 operations consume about 16× less energy than FP32, 4-bit operations can reach 64× reductions, and Q8BERT reduces BERT size by 4× with minimal degradation (Zafrir et al. 2019)
Knowledge distillation (Hinton et al. 2015)	Serving scale	Moves repeated inference cost into a one-time teacher-student training process	DistilBERT retains 97% of BERT accuracy with 40% fewer parameters and 60% faster inference (Sanh et al. 2019)

The table's three levers have different deployment caveats: pruning[33] depends on sparsity structure and hardware support, quantization[34] compounds savings across arithmetic and memory movement, and knowledge distillation[35] amortizes its extra training cost across repeated serving.

Pruning, quantization, and distillation form the core toolkit for sustainable AI development, but their sustainability value depends on the measured bottleneck. In this chapter, the design question is not how to implement each compression method from first principles; it is how to rank these levers against memory movement, serving volume, carbon intensity, and lifecycle cost.

While model compression, efficient architectures, and carbon-aware scheduling provide the technical mechanisms for efficiency, deploying them haphazardly yields diminishing returns. To achieve maximum impact, engineering teams must synthesize these isolated techniques into a coherent, prioritized strategy that attacks the largest sources of emissions first.

Checkpoint 15.6: Prioritizing decarbonization strategy

You are deploying a 70B LLM for a latency-sensitive application.

- ☐ **Technique ranking**: Rank INT4 quantization, unstructured pruning, carbon-aware scheduling, and knowledge distillation by their potential to reduce total energy consumption.
- ☐ **Systems justification**: Justify the order using the principle that "memory movement costs more than arithmetic."

15.7.2.2 TinyML optimization stack

TinyML makes the lifecycle argument concrete because the environmental budget appears as a hard physical envelope rather than a reporting category. A microcontroller deployment must fit the model and its peak activations into kilobytes of SRAM, finish inference before the sensor or user-facing deadline expires, and remain within a milliwatt or microwatt power budget. Standard INT8 quantization provides a 4× memory reduction and often lowers energy substantially; structured pruning can add further savings when sparsity maps to hardware-visible work removal. Those techniques often get a model into the right range, but energy-harvesting devices require a stricter sequence: first make the memory plan feasible, then reduce switching activity, and only then search for an architecture that uses the harvested-energy budgets in table 15.9 well. Table 15.13 summarizes extreme TinyML optimization techniques at the end of that sequence.

33 **Pruning Energy Impact**: Structured pruning at 90 percent sparsity reduces inference energy by 2–10× because eliminated weights require neither storage nor computation, directly reducing both memory bandwidth and arithmetic. SparseGPT achieves 60 percent unstructured sparsity on LLMs with less than 1 percent accuracy loss, though realizing energy savings from unstructured sparsity requires hardware with native sparse execution support (for example, NVIDIA's Sparse Tensor Cores).

34 **Quantization Energy Savings**: INT8 multiply-accumulate consumes roughly 16× less energy than FP32 because both the arithmetic unit area and memory bandwidth shrink proportionally with bit-width. GPTQ enables 4-bit LLM quantization (64× energy reduction per operation) with only 2 percent perplexity increase, reducing LLaMA-65B from 130 GB to 32 GB and enabling consumer-GPU deployment. The sustainability implication is multiplicative: lower precision reduces energy in both compute and memory movement simultaneously.

35 **Knowledge Distillation**: Introduced by Hinton et al. (2015), distillation trains a compact "student" model on soft probability targets from a larger "teacher," capturing inter-class relationships that hard labels discard. DistilBERT retains 97 percent of BERT's accuracy with 40 percent fewer parameters and 60 percent faster inference. The sustainability arithmetic is decisive: the one-time cost of training teacher plus student is amortized across millions of inference queries, making distillation one of the highest-ROI sustainability interventions for deployed services.

Table 15.13: Extreme TinyML Optimization Techniques: For energy-harvesting devices operating on microwatt budgets, these techniques push beyond conventional INT8/pruning approaches, trading model quality, search cost, and deployment specificity for the dramatic efficiency gains required for truly autonomous operation (Courbariaux et al. 2016; Lin et al. 2020; Prakash et al. 2023).

Technique	Typical Accuracy Impact	Memory Reduction	Energy Reduction
Binary Neural Networks	task-dependent	up to 32×	order-of-magnitude when bit operations dominate
Neural Architecture Search for MCUs	varies	task-dependent	2–5× vs. baseline

That sequence has three passes:

1. **Memory fit**: Microcontrollers operate with 64 KB to 2 MB SRAM, so the memory pass starts with peak activation analysis, not just parameter count. A model that has small weights can still fail if one intermediate tensor exceeds the tensor arena, so TinyML runtimes rely on in-place operations, tensor-arena planning, and operator fusion to reuse buffers and avoid fragmentation.
2. **Switching energy**: Once the memory plan fits, the energy pass asks whether ordinary INT8 arithmetic is still too expensive for the deployment. On devices powered by solar, vibration, or RF harvesting, Binary Neural Networks (BNNs) may be justified because XNOR-style operations replace multiply-accumulate work with bit operations (Courbariaux et al. 2016). That trade is not free: the accuracy loss is task-dependent, so BNNs belong in applications where always-on sensing within the harvested-energy budgets in table 15.9 matters more than full-precision classification margins.
3. **Architecture search**: Automated design becomes useful only after the SRAM, latency, and harvested-energy constraints are explicit. MCUNet jointly searches the network and inference schedule for memory-limited microcontrollers and demonstrated ImageNet-scale accuracy on 256 KB SRAM devices (Lin et al. 2020); Once-for-All Networks amortize search by training a supernet from which device-specific subnetworks can be extracted (Cai, Gan, Wang, et al. 2020); and ProxylessNAS optimizes directly against hardware latency and energy (Cai et al. 2019).

These methods should not be read as a menu of sustainable techniques. They are responses to a specific failure: the hand-designed model cannot simultaneously satisfy SRAM, latency, and harvested-energy constraints. The lifecycle budget must record the resulting trade-offs because accuracy loss, extra search or distillation training, and rebound-driven deployment growth can erase nominal per-inference savings.

15.7.2.3 Lifecycle-aware systems

Many AI deployments operate with a short-term mindset: train a model, deploy it, replace it a few months later, and treat the discarded training run or device generation as yesterday's cost. Lifecycle-aware systems treat that churn as part of the footprint. If the model will be updated repeatedly, the first sustainability question is whether the next update requires full retraining. Incremental learning and transfer learning reduce this waste because fine-tuning pretrained models on new datasets can cut computational cost by orders of magnitude compared with training from scratch (Raffel et al. 2020).

The deployment boundary matters just as much as the training boundary. Edge deployment can reduce communication energy by running inference on specialized low-power hardware at the point of use (Xu et al. 2020), but the lifecycle account must include the embodied carbon of manufacturing and replacing those devices. Embedding LCA methodologies into AI workflows allows teams to see this trade-off early: a cloud model, an edge model, and a hybrid cascade may have different winners depending on query volume, device lifetime, grid carbon intensity, and retraining frequency (International Organization for Standardization 2006a, 2006b). Henderson et al. (2020) argue for systematic reporting of ML energy and carbon footprints precisely so these comparisons can be made consistently. As Jevons Paradox warns, the accounting must also include

usage growth, because optimizing one stage may increase total impact if lower costs enable wider deployment.

15.7.2.4 Benchmarks and operating metrics

Benchmarks matter when they make efficiency visible at procurement and design time. The ML.ENERGY Leaderboard (ML.ENERGY Initiative et al. 2023) ranks models by energy efficiency and carbon footprint, encouraging researchers to optimize for sustainability alongside accuracy. MLCommons extends the same idea into standardized measurement: the MLPerf benchmark suite defines power-measurement protocols for data center and edge deployments, making sustainability claims comparable across hardware, software stacks, and workload classes.

The right efficiency metric depends on the serving regime. Batch inference is naturally expressed as samples per joule, latency-sensitive serving as queries per joule, and generative workloads as joules per token because output length varies across requests. Standardization does not make one platform universally green; it makes the workload, measurement window, and energy denominator visible enough for procurement and architecture decisions to be argued quantitatively.

For sub-watt TinyML deployments, the MLPerf Tiny benchmark suite provides the same discipline at microcontroller scale. Table 15.14 summarizes benchmark tasks and typical energy requirements spanning from sub-millijoule to multi-millijoule ranges. The measurement methodology requires external power monitors such as INA219, INA226, or Joulescope-class instruments and specifies warm-up periods, measurement windows, and statistical reporting requirements so that tiny efficiency claims can be reproduced across submissions.

Table 15.14: MLPerf Tiny Benchmark Suite: Standardized benchmarks for TinyML systems measure accuracy, latency, and energy consumption on microcontroller-class hardware. Reference model sizes indicate minimum viable deployments; optimized implementations often achieve 2–10× better energy efficiency.

Benchmark	Task	Reference Model	Typical Energy (mJ/inference)
Visual Wake Words	Image Classification (person detection)	MobileNetV1 0.25 (250 KB)	0.1-1.0 mJ
Keyword Spotting	Audio Classification (12 keywords)	DS-CNN (19 KB)	0.05-0.5 mJ
Anomaly Detection	Time Series (machine health)	Deep Autoencoder (5 KB)	0.01-0.1 mJ
Image Classification	Visual Recognition (CIFAR-10)	ResNet-8 (70 KB)	0.5-5.0 mJ

Energy and latency still have to be read together. As equation 15.17 shows, the Energy Delay Product (EDP) balances energy consumption against response time by penalizing solutions that save power only by taking too long:

$$\text{EDP} = E \times T = P_{\text{average}} \times T^2 \tag{15.17}$$

where E is energy consumed, T is execution time, and P_{average} is average power. The quadratic delay term penalizes solutions that achieve low energy through excessive delays. Lower EDP indicates better efficiency, enabling comparison of systems with different energy-latency trade-offs.

For TinyML deployments, EDP helps identify optimal operating points. A microcontroller running at reduced clock frequency consumes less power but takes longer to complete inference. The EDP-minimizing configuration often operates at moderate frequencies where voltage can be reduced (exploiting the quadratic voltage term in CMOS power) without excessive latency penalties.

Sustainability metrics complement traditional performance benchmarks by creating evaluation frameworks that account for both capability and environmental impact. The EU AI Act's 2024 requirements for providers of general-purpose AI models to document known or estimated model energy consumption illustrate how these metrics can move from voluntary reporting practice toward compliance requirements.

15.7.3 Infrastructure optimization

Algorithmic optimizations reduce per-operation energy, but the operational environment determines whether those savings translate to actual emissions reduction. Infrastructure-level innovations

address the physical context where computational efficiency gains are realized: renewable energy integration, carbon-aware workload scheduling, and AI-driven cooling optimization each target a different layer of the data center stack.

15.7.3.1 Green data centers

A single hyperscale data center can consume over 100 MW of power—comparable to a small city[36]. Reducing this footprint requires three complementary strategies: renewable energy integration, advanced cooling, and AI-driven optimization.

Major cloud providers have announced renewable-energy commitments, but intermittency remains a challenge. AI infrastructure must incorporate energy storage solutions and intelligent scheduling that shifts workloads to times of peak renewable availability. Google's published 2030 target for 24/7 carbon-free energy[37] illustrates the harder version of this goal: matching every unit of electricity consumed with renewable generation in real time rather than relying on annual carbon offsets.

Cooling claims a large share of data center electricity (the cooling-overhead figure established for the PUE treatment in section 15.4.6)[38]. Liquid cooling, which transfers heat directly from accelerators using specially designed coolants, is significantly more effective than traditional air cooling and is used in high-density AI clusters. Software control of that cooling is the other lever: the DeepMind optimization in section 15.2.2.4 reclaimed a substantial fraction of cooling energy without any hardware change, demonstrating AI improving the sustainability of its own infrastructure.

15.7.3.2 Carbon-aware scheduling

Grid carbon intensity fluctuates dramatically based on the mix of power sources available at any given time—from 50 g CO_2/kWh in nuclear-heavy France to 820 g/kWh in coal-dependent Poland. **Carbon-aware scheduling** dynamically shifts AI computations to times and locations where low-carbon energy is available. For deadline-tolerant workloads with geographic or temporal flexibility, it can be one of the highest-leverage emissions levers.

Carbon-aware scheduling is fundamentally a **load shifting software problem**. The scheduler queries real-time grid carbon intensity APIs (for example, ElectricityMap, WattTime) to pause nonurgent training jobs during carbon-intensive periods, such as the evening peak, and migrate workloads to geographic regions with excess renewable energy, such as solar peaks in California or wind peaks in Iowa.

Google's carbon-intelligent computing platform[39] demonstrated this approach at scale, achieving a 40 percent reduction in carbon footprint under its global workload-shifting assumptions (Radovanovic et al. 2021). Within the broader energy-gap cascade of figure 15.23, carbon-aware scheduling is one of the systemic-stage steps, contributing a more conservative 1.3× average reduction for mixed production fleets, where only some jobs are deadline-tolerant enough to move across time or geography.

The effectiveness of carbon-aware scheduling depends on accurate real-time grid emissions data. Average grid intensity is useful for retrospective reporting because it estimates the emissions associated with energy already consumed. Marginal emissions are more useful for scheduling because they estimate which generator responds when the workload adds or removes demand. The Electricity Maps API provides real-time CO_2 emissions data for power grids worldwide[40], while WattTime provides marginal emissions data showing which power plants turn on or off next. Figure 15.24 demonstrates the scheduling opportunity: shifting training jobs to low-carbon hours in lower-carbon regions reduces emissions by up to 8× without changing a single line of model code.

Renewable energy variability presents a key challenge for carbon-aware scheduling. Figure 15.25 captures European grid dynamics: solar energy peaks at midday, wind shows distinct peaks in mornings and evenings, and fossil generation fills the gaps. This temporal pattern determines when AI workloads can run on clean energy.

Energy-aware AI frameworks complement scheduling when they optimize the workload rather than only shifting its location. Zeus (You et al. 2023) achieves 75 percent energy savings on BERT training by automatically finding optimal energy-performance trade-offs, while Perseus (Chung et al. 2023) reduces large-model training energy consumption by up to 30 percent by mitigating energy bloat. These tools, alongside CodeCarbon for emissions tracking (Schmidt et al. 2021), democratize energy optimization beyond hyperscale companies.

36 **PUE Gap**: The industry-average PUE of 1.67 means 40 percent of electricity powers cooling and infrastructure rather than computation, while highly optimized Google facilities have reported PUE near 1.08 (only 7.4 percent overhead). For a 100 MW AI data center, this gap represents 59 MW of wasted power, enough to run 47,000 homes. Each 0.1 PUE improvement at hyperscale saves millions in annual electricity costs and tens of thousands of tonnes of CO_2 per year on an average U.S. grid.

37 **24/7 Carbon-Free Energy (CFE)**: Google's published 2030 target requires matching every hour of consumption with real-time carbon-free generation, far harder than annual-average offsets. Closing the remaining gap requires substantial storage, transmission, and clean-generation investment. The distinction matters: annual-average carbon neutrality allows fossil-fuel hours offset by renewable credits, while hourly CFE forces genuine elimination of carbon-emitting generation from the supply chain.

38 **Cooling Energy Density**: AI accelerator racks can exceed 100 kW per cabinet, roughly 10× the density of traditional servers, making air cooling physically inadequate. Direct liquid cooling reduces cooling energy from 38 percent to under 10 percent of total facility power by transferring heat at 3,000× the volumetric efficiency of air. For AI data centers, the cooling system is no longer infrastructure overhead but an active constraint on how many accelerators can be physically co-located.

39 **Carbon-Aware Scheduling at Scale**: Google's carbon-aware data center study reports 15 percent carbon reduction through intra-region temporal shifting alone, and 40 percent globally by routing nonurgent batch compute across time zones to chase renewable peaks (Radovanovic et al. 2021). The key insight is that delay-tolerant workloads gain access to dramatically different grid carbon intensities without any model or infrastructure changes.

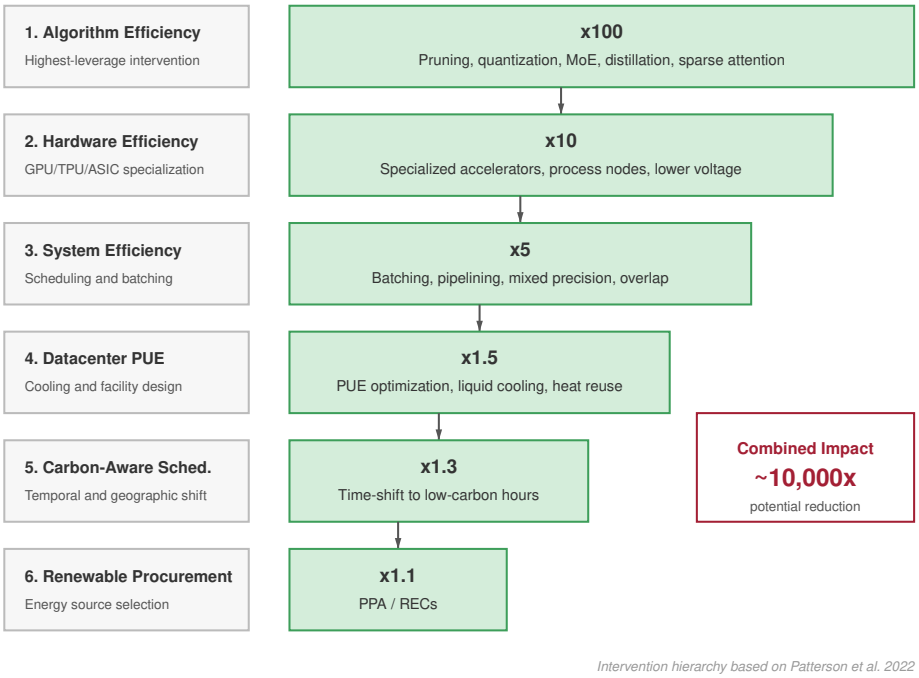


Figure 15.23: Carbon-Aware Workload Scheduling in the Intervention Cascade: The six-step intervention cascade from figure 15.5, recalled here because carbon-aware scheduling is one of its systemic-stage steps, contributing a conservative 1.3× reduction for mixed fleets via temporal and geographic shifting.

AI-driven cooling optimization is a software-deployable lever for reducing data center energy consumption when the facility has sufficient sensing and controllable cooling equipment. Traditional cooling systems rely on fixed control policies with predefined temperature thresholds, often consuming more energy than necessary; a learned controller that adapts to real-time conditions reclaims much of that waste, as the DeepMind deployment analyzed in section 15.2.2.4 showed without any hardware change.

Complementing software optimization, liquid cooling and immersion cooling change the thermal design space for dense accelerator clusters. Liquid cooling transfers heat directly from accelerator chips using specially designed coolants, achieving 3,000× better heat transfer than air. Immersion cooling submerges entire server racks in nonconductive liquid coolants, eliminating traditional air-based systems entirely. These approaches enable higher compute densities with lower power consumption—critical for accelerators whose thermal design power reaches hundreds of watts per chip.

40 **Marginal vs. Average Emissions**: WattTime's marginal emissions data identifies which power plant turns on next when load increases, enabling 2–5× better carbon optimization than grid-average intensity. The distinction is critical: average intensity smooths out peaks, but marginal data reveals that adding 1 MW of load at the wrong hour can activate a coal peaker plant at 900 g/kWh even on a nominally "clean" grid.

15.7.4 Case study: Google's framework

The value of Google's case study is that it decomposes mitigation across the same layers this chapter has tracked: model, machine, mechanization, and map. Table 15.15 summarizes the "4 Ms" that Google engineers identified for reducing the carbon footprint of rapidly expanding AI workloads (Patterson, Gonzalez, Holzle, et al. 2022).

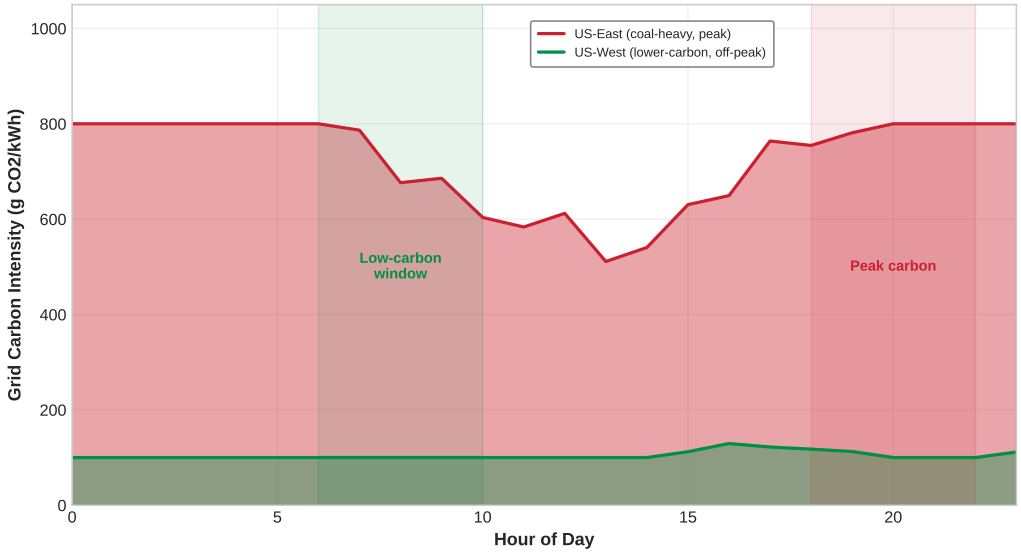


Figure 15.24: Carbon-Aware Scheduling Opportunity: Grid carbon intensity varies by region and time of day. Training during off-peak hours in lower-carbon regions (US-West) produces up to 8× less carbon than peak hours in coal-heavy regions (US-East). This geographic and temporal flexibility gives schedulers a large emissions lever when workloads can move.

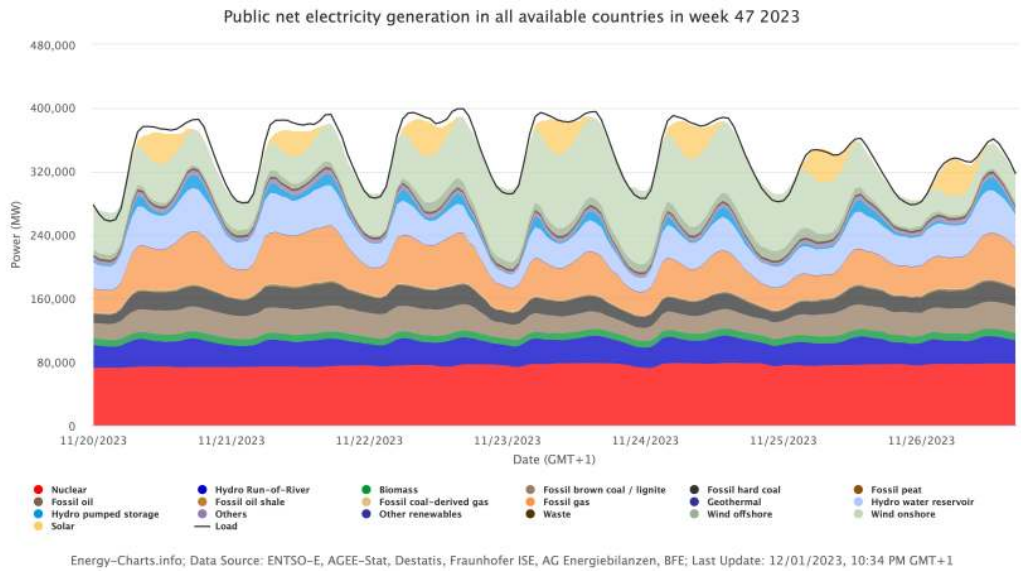


Figure 15.25: European Energy Mix: Renewable energy sources exhibit significant temporal variability, necessitating fossil fuel supplementation to meet consistent demand. Understanding this fluctuation is important for effectively scheduling AI workloads to periods of high renewable energy availability. Source: Uenergy charts.

Table 15.15: Google's 4 Ms Sustainability Framework: The framework combines model architecture, hardware specialization, infrastructure efficiency, and carbon-aware placement so improvements compound across the AI stack.

Lever	Intervention	Reported efficiency effect	Example mechanism
Model	Select efficient AI architectures such as sparse models or neural-architecture-search-derived designs	5–10× lower computation requirements without compromising model quality	Evolved Transformer and Primer

Lever	Intervention	Reported efficiency effect	Example mechanism
Machine	Use AI-specific hardware rather than general-purpose systems	2–5× performance-per-watt improvement; TPUs show 5–13× greater carbon efficiency relative to nonoptimized GPUs	Tensor Processing Units
Mechanization	Run optimized cloud infrastructure at high utilization	1.4–2× energy reduction compared to conventional on-premise data centers	Lower facility PUE than the industry-average baseline used in the 2021 study
Map	Place data centers in regions with low-carbon electricity supplies	5–10× lower gross emissions	Real-time monitoring of renewable energy usage across infrastructure

The combined effect of these practices produces multiplicative efficiency gains. For instance, implementing the optimized Transformer model on TPUs in strategically located data centers reduced energy consumption by a factor of 83 and CO_2 emissions by a factor of 747.

In the period studied by Patterson, Gonzalez, Holzle, et al. (2022), systematic efficiency improvements constrained energy consumption growth even as AI deployment expanded across Google's product ecosystem. A significant indicator of this progress is the observation that AI workloads maintained a less-than-15-percent proportion of Google's total energy consumption over the reported period. As AI functionality expanded across Google's services, corresponding increases in compute cycles were offset by advancements in algorithms, specialized hardware, infrastructure design, and geographical optimization.

Empirical case studies demonstrate how sustainable-AI engineering can improve both capability and environmental impact. For example, Patterson, Gonzalez, Holzle, et al. (2022) compare GPT-3 with Google's GLaM and report improved quality metrics alongside reduced training computation and lower-carbon energy sources, while the GLaM model paper explains the mixture-of-experts architecture behind that efficient scaling (Du et al. 2022).

The strategy in the case study—combining systematic measurement, carbon-aware development, transparency in reporting, and renewable energy transition—is useful as a framework for sustainable AI scaling. The analysis also argues that earlier extrapolations overstated ML energy requirements by large factors because they did not account for efficiency improvements and workload measurement boundaries, underscoring the importance of empirical measurement over theoretical projections.

15.7.5 Engineering guidelines for sustainable AI development

Measurement, optimization, and scheduling frameworks provide the analytical foundation, but implementation requires prioritization. The following checklist should be read as a decision aid: first measure the dominant lifecycle term, then choose the lever that changes it.

- **Measure first**: Tools like CarbonTracker and CodeCarbon track the emissions of training runs. Teams cannot improve what they do not measure, and establishing baseline metrics is essential for validating the effectiveness of optimization efforts (Anthony et al. 2020; Schmidt et al. 2021).
- **Choose the region**: Train models in data centers powered by renewable energy. As established in section 15.1.1, grid carbon intensity spans the 8 to 40 times range for representative region pairs; scheduling workloads where clean energy is most abundant yields immediate reductions.
- **Optimize the model**: Avoid training the largest model possible by default. Pruning, quantization, and knowledge distillation find the smallest model that meets accuracy targets. A 90 percent accurate model requiring 10 percent of the resources often provides better real-world value than a 95 percent accurate model requiring full resources.
- **Avoid retraining from scratch**: Transfer learning and fine-tuning reduce computational requirements by orders of magnitude compared to full retraining.
- **Select efficient hardware**: Energy-efficient accelerators, such as TPUs or specialized inference chips, reduce deployment costs. The full hardware lifecycle and workload-specific platform selection matter as much as raw throughput.

- **Account for the full lifecycle**: Longer hardware refresh cycles and responsible e-waste policies reduce total environmental impact. Manufacturing often exceeds operational energy consumption, making hardware longevity a critical sustainability factor.

The cumulative impact of individual technical choices depends on systemic, industry-wide adoption. Without external pressure, market forces prioritize speed and scale over efficiency. Policy and regulatory frameworks translate engineering possibilities into industry-wide practice by making sustainable choices a financial and legal imperative.

15.8 Policy, Regulation, and the Path Forward

If a company can slash its cloud computing bill by relocating its training cluster to a region powered entirely by cheap, high-emission coal, the market alone will not prevent them from doing so. Engineering ingenuity can provide the tools for efficient computation, but it requires policy, regulation, and carbon pricing to ensure that using those tools becomes a financial and legal imperative rather than just a corporate public relations talking point.

For a systems reader, policy is the control plane that changes the objective function. Reporting rules make hidden energy and embodied-carbon costs measurable, carbon pricing turns location into a scheduling variable, and procurement standards make lifecycle accounting part of infrastructure design. The mechanisms below matter because they determine which optimizations become economically rational at fleet scale.

15.8.1 Regulatory mechanisms

Effective AI sustainability governance operates through a combination of mandatory reporting, emission restrictions, and financial incentives, though global policy fragmentation presents a significant implementation challenge. The European Union has taken a leading role with mandatory approaches, notably the EU AI Act[41] and the **Corporate Sustainability Reporting Directive (CSRD)**.[42] The AI Act creates separate obligations for general-purpose AI models and for general-purpose AI models with systemic risk, including documentation of computational resources and known or estimated model energy consumption. The CSRD mandates that large companies disclose their environmental impacts, including Scope 1, 2, and 3 emissions from AI operations, according to standardized, audited reporting frameworks. This regulatory shift transforms energy monitoring from an optional optimization into a legal necessity.

Beyond measurement mandates, governments are exploring direct restriction mechanisms. These include setting limits on computational power available for training large AI models, mirroring **Emissions Trading Systems (ETS)**[43] used in environmental policy. Such "cap-and-trade" systems for compute would force organizations to operate within predefined energy budgets or procure additional capacity, creating a market for computational carbon credits. The expansion of carbon pricing and **Carbon Border Adjustment Mechanisms (CBAM)** is converting the geographic location of compute into a direct financial variable—the carbon intensity of regional electricity grids varies across the span established in section 15.1.1, making carbon-aware scheduling a key compliance strategy.

To balance these restrictions, government incentives play a proactive role. Financial support, tax benefits, and grants for Green AI research can make sustainability a competitive advantage. Governments can also use their public procurement power, mandating that vendors meet sustainability benchmarks such as operating on carbon-neutral data centers or using energy-efficient models. Broader corporate reporting frameworks—the Greenhouse Gas Protocol, TCFD, and ISSB—scrutinize Scope 3 emissions, encompassing the substantial embodied carbon of GPU procurement and data center construction alongside operational emissions of outsourced cloud compute.

15.8.2 Industry self-regulation and standards

Alongside government mandates, the AI industry is driving significant environmental improvements through self-regulation and common standards. The most visible commitments from major cloud providers—Google, Microsoft, and Amazon—focus on matching data center electricity consumption with renewable energy procurement and increasing direct clean-energy supply. Going further, the push for 24/7 Carbon-Free Energy (CFE) aims to match every hour of energy consumption with

41 **EU AI Act (2024)**: The EU AI Act, adopted in 2024, is an early broad AI regulatory framework that creates obligations for general-purpose AI models and additional obligations for models with systemic risk, including models presumed to have high-impact capabilities above 10^{25} FLOPs of training compute. Technical documentation must include computational resources and known or estimated model energy consumption. For violations by providers of general-purpose AI models, Article 101 permits fines up to 3 percent of annual worldwide turnover or EUR 15 million; the separate 7 percent maximum applies to prohibited-practice violations.

42 **CSRD (Corporate Sustainability Reporting Directive)**: Effective 2024, this EU regulation requires 50,000+ companies to disclose audited Scope 1, 2, and 3 emissions using standardized ESRS frameworks. For AI infrastructure, CSRD forces disclosure of previously hidden costs: the embodied carbon of GPU procurement, energy from outsourced cloud training, and end-of-life hardware disposal that collectively constitute the majority of an AI system's Scope 3 footprint.

43 **Emissions Trading for Compute**: The EU ETS (2005) pioneered cap-and-trade for industrial emissions; applying this model to AI compute would set aggregate energy budgets for training clusters and let organizations trade surplus capacity. The mechanism converts sustainability from a voluntary optimization into a priced constraint: organizations that invest in efficiency can sell unused allocation to less efficient competitors, creating a financial incentive aligned with the iron law's utilization term (η_{hw}).

real-time clean energy procurement, moving beyond annual averages and carbon offsets that can obscure actual emissions from fossil-fuel-reliant grids (Monyei and Jenkins 2018).

Internal carbon pricing is another effective self-regulatory tool. By assigning a "shadow price" to carbon emissions, companies integrate environmental costs directly into financial decision-making for AI projects, naturally prioritizing investments in energy-efficient hardware and low-emission models. Voluntary checklists and open-source tools promote accountability when they feed those same project decisions: projects like **CodeCarbon** and **ML CO_2 Impact** provide frameworks that allow developers to estimate and track model carbon footprints directly within their workflows (Schmidt et al. 2021; Lacoste et al. 2019).

Standardized benchmarks provide the objective data needed to validate these efforts. **MLCommons**, through its MLPerf benchmark suite, has incorporated power measurement protocols for both data center and edge deployments. By establishing metrics like "samples per Joule" and "Joules per token," MLCommons enables fair, transparent comparison of AI system efficiency across different hardware and software platforms. These benchmarks, combined with independent sustainability audits from organizations like the Green Software Foundation, create a measurable mechanism for holding the industry accountable and driving competition toward genuinely greener AI.

15.8.3 Public engagement and environmental justice

Effective AI sustainability governance requires public support, which depends on transparency, clear communication, and equitable access. Currently, public understanding of AI's environmental impact is limited and often polarized between narratives of technological salvation and ecological disaster. Fostering informed discourse requires moving beyond **greenwashing**[44]—the practice of making misleading claims about environmental responsibility—toward genuine, verifiable transparency.

Pledge-style disclosure is useful only when it creates auditable data rather than reputational cover. The **Montréal Carbon Pledge**, originally a commitment by institutional investors to measure and disclose carbon footprints annually, is a useful model precisely because its value lies in the disclosed data, not the pledge itself; the same standard applies to an AI organization, whose sustainability claims are only as credible as the workload-level measurements behind them.

Transparency establishes the evidence base; environmental justice asks how the burdens and benefits revealed by that evidence are distributed. As section 15.1.2 established, an ML fleet allocates electricity demand, water use, land pressure, and e-waste across communities that often do not share in its economic benefits, which is why site selection is part of the engineering design space. At the policy layer, this distributional question becomes a reporting requirement: social impact assessments and equitable-access provisions for large-scale AI projects turn the fairness concern into auditable obligations rather than aspirations.

15.8.4 Future research directions

The research agenda follows the same engineering logic as the mitigation framework: remove data movement, close the measurement gap, and avoid redundant computation. One major direction is the development of **non-von Neumann computing architectures**[45], such as **neuromorphic computing** and **in-memory computing**. By processing data where it is stored, these paradigms aim to eliminate the "von Neumann bottleneck"—the energy-intensive shuttling of data between memory and processing units that can account for 60–80 percent of a system's power consumption. Successful implementation could yield energy efficiency improvements of 100–1000× for certain AI workloads.

A critical implementation barrier is the "measurement gap": AI teams need standardized, workload-level reporting of energy use and carbon emissions rather than relying only on coarse proxy metrics (Henderson et al. 2020). Coarse methods often rely on proxies such as GPU-hours multiplied by average grid intensity, which fail to capture the real-world dynamics required by reporting regimes that care about time, location, and workload boundary. Developing and standardizing granular, real-time energy and carbon accounting tools is essential for both compliance and effective optimization.

A second research direction reduces redundant computation before it reaches the accelerator. Research shows that the predictive value of training data often decays, meaning models are frequently trained on vast datasets with diminishing returns (Wu et al. 2022). Smarter data sampling, active

44 **Greenwashing in AI**: Manifests as claiming "carbon neutrality" through offsets while expanding data center capacity, or highlighting per-query efficiency gains while total compute grows 10×. Green-claims rules can require verifiable evidence for environmental claims. For ML engineers, the technical litmus test is whether sustainability reporting covers all three GHG Protocol scopes, or conveniently omits Scope 3 (hardware manufacturing, cloud supply chain) where much of AI's carbon can reside.

45 **Von Neumann Bottleneck**: John von Neumann's 1945 stored-program architecture separates processing from memory, requiring constant data shuttling that consumes 60–80 percent of system power. For AI workloads dominated by matrix multiplications with low arithmetic intensity, this bottleneck means most energy moves data rather than computes results. In-memory and neuromorphic architectures attack this directly, with potential 100–1,000× energy reductions for inference by eliminating the memory-processor round trip.

learning, and data valuation techniques can optimize training processes to use only the most informative data, reducing computational waste without sacrificing accuracy. Ultimately, an integrated approach combining algorithmic efficiency, hardware innovation, renewable energy adoption, and transparent governance is necessary to ensure AI's trajectory aligns with global sustainability goals.

Minimizing redundant computation through smarter data curation directly aligns regulatory compliance with operational efficiency. The most dangerous obstacles to sustainable AI are not technical limitations but incorrect assumptions—miscalculations that cause well-intentioned teams to inadvertently increase their environmental footprint.

15.9 Fallacies and Pitfalls

Sustainability involves counterintuitive physics where efficiency improvements can increase total consumption and geographic choices dominate all other optimizations. These fallacies and pitfalls capture errors that waste compute budgets and planetary resources through misallocated optimization effort.

Fallacy: *Cloud computing automatically makes AI systems more environmentally sustainable.*

Engineers assume cloud providers operate efficiently and sustainably. In production, geographic region dominates all other factors through grid carbon intensity differences. Training a 7B model on 64 A100s for 14 days produces 5.7 t CO_2 on the US average grid (429 g/kWh) but only 264.9 kg CO_2 on Quebec's hydroelectric grid (20 g/kWh operational)—a roughly 21.5× difference for this US-average-versus-Quebec pairing, which sits inside the 8 to 40 times geographic range established in section 15.1.1. Coal-powered grids emit 800–1000 g CO_2/kWh while well-managed hydroelectric sources emit 10–50 g CO_2/kWh. As demonstrated in section 15.3, teams that deploy to default cloud regions without checking grid carbon intensity waste a large multiple of the carbon budget necessary, turning "cloud sustainability" into a geographic lottery rather than an inherent advantage.

Pitfall: *Focusing only on operational energy consumption while ignoring embodied carbon and lifecycle impacts.*

Teams optimize training efficiency while ignoring manufacturing emissions. In low-carbon grids, embodied carbon can dominate fleet-level footprint accounting. As quantified in section 15.3.1.1, an A100 accelerator embodies roughly 150 kg CO_2 from manufacturing (Luccioni et al. 2023); for the 14-day, 64-A100 training run above, the unamortized upfront burden is roughly 9.6 metric tons CO_2. Amortized over a 4-year service life, the share attributable to this specific 14-day job is about 90 kg, but the unamortized fleet-level number is what dominates total footprint accounting: it exceeds the job's operational emissions on Quebec's clean grid where operational emissions are minimal. Extending hardware lifetime from three to five years reduces amortized embodied carbon by 40 percent. Organizations focusing exclusively on operational efficiency miss this procurement and depreciation lever while optimizing marginal gains in PUE or compute efficiency.

Fallacy: *TDP is actual power consumption.*

Thermal Design Power (TDP) is the maximum sustained draw the cooling system must handle, not the wattage the accelerator actually consumes under a given workload. Real power varies with utilization, memory access pattern, and clock frequency: an H100 idles around 50–80 W, runs inference workloads at 250–400 W, and approaches its 700 W TDP only during sustained training with high tensor-core occupancy. Using TDP for energy calculations overestimates carbon for inference fleets (which rarely sustain peak power) and underestimates it for sustained training on newer hardware with dynamic boost above the published envelope. Carbon and electricity-cost estimates that drive geographic-placement decisions (section 15.3) should use measured average power per workload class, not datasheet TDP.

Pitfall: *Using PUE as a complete environmental metric.*

Data-center operators report PUE in sustainability disclosures and engineers treat the figure as a single number that summarizes efficiency. PUE measures only the ratio of total facility power to IT power; it says nothing about water consumption, embodied carbon in manufacturing, or the carbon intensity of the electricity the facility consumes. A data center with PUE 1.06 running on a coal-heavy grid (820 g CO_2/kWh) has a far larger operational carbon footprint than a data center with PUE 1.40 running on hydroelectric power (10 g CO_2/kWh)—roughly a 60× difference that PUE alone hides. Reporting total environmental impact requires PUE, WUE (water-usage effectiveness), grid carbon intensity, and embodied-carbon amortization together. No single metric is sufficient.

Fallacy: *Efficiency improvements automatically reduce total environmental impact.*

Engineers assume that halving inference cost cuts environmental impact in half. Jevons Paradox warns that efficiency improvements can increase total consumption by enabling expanded usage. In a rebound scenario, reducing token cost from $0.06 to $0.002 per 1,000 tokens (a 30× improvement) while inducing a 100× increase in query volume grows total emissions despite per-query efficiency gains. A quantization change that reduces inference energy by 4× can still increase total energy if relaxed cost constraints expand deployment by more than 4×. Teams that optimize efficiency without usage governance can therefore transform sustainability wins into consumption growth, requiring carbon budgets and usage caps of the kind motivated by the Jevons analysis in figure 15.22.

Pitfall: *Treating carbon offsets as a substitute for reducing actual emissions.*

Organizations purchase offsets to neutralize emissions without validating offset quality. In reality, analysis of voluntary carbon markets reveals that 60–90 percent of credits fail to deliver claimed reductions due to inflated baselines, nonpermanent sequestration, or projects that would have occurred regardless. A company training models on coal grids (1000 g CO_2/kWh) and buying offsets spends 2–3× more than directly migrating to renewable regions (20–50 g CO_2/kWh) while achieving inferior environmental outcomes. Offset projects take 5-20 years to sequester carbon while compute emissions are immediate. Teams that prioritize offsets over actual reduction miss the geographic leverage established in section 15.1.1 and delay renewable energy transitions that deliver permanent improvements.

Fallacy: *Component-level optimization guarantees lifecycle improvement.*

Teams reduce training cost to improve sustainability without analyzing deployment scale. In production, training-inference trade-offs often invert total emissions. A model pruned by 40 percent to save training energy but requiring 2× inference compute increases total lifecycle emissions if it serves more than 100 million queries—a crossover point reached in three to six months for production systems. Edge deployment that reduces data center energy by 60 percent but requires manufacturing 10,000 specialized devices adds 1,500-2,000 kg embodied carbon (10× the cloud training emissions). Extending GPU lifetime from three to five years reduces amortized embodied carbon by 40 percent but may sacrifice 15–25 percent operational efficiency; the lifecycle break-even depends on grid carbon intensity, with lifetime extension dominating on clean grids and efficiency winning on dirty grids. Effective sustainability requires holistic analysis across section 15.3.1.2 rather than local optimization.

Pitfall: *Evaluating sustainability at the component boundary instead of the lifecycle boundary.*

A model aggressively pruned to save training energy, only to require massive computational overhead during inference to compensate for lost accuracy, illustrates the danger of localized optimization. The same mistake appears when hardware teams report accelerator efficiency without procurement carbon, platform teams report PUE without grid intensity, or model teams report training emissions without expected serving volume. Avoiding these systemic pitfalls requires the lifecycle boundary developed in section 15.3.1.2: training, serving, embodied carbon, regional grid mix, hardware lifetime, and demand growth must be evaluated together before a change can be called sustainable.

15.10 Summary

The lifecycle boundary is the through-line for the final synthesis: a sustainable architecture must account for training, serving, embodied carbon, grid mix, hardware lifetime, and demand growth together. Sustainable AI represents the "physical limit" of the Machine Learning Fleet. High-performing ML systems can optimize logic, exploit specialized hardware, launch global services, and harden their defensive perimeter while still failing the environmental test. The final gating constraint is whether these systems can exist within the energy, water, and material boundaries of the planet.

Sustainability is a core engineering requirement, not a discretionary "nice-to-have." The lifecycle carbon footprint spans from the per-H100 embodied carbon quantified in section 15.3.1.1 to the thousands of megawatt-hours consumed during training. Decode-phase bandwidth-bound inefficiency, where autoregressive generation leaves accelerators idling on memory rather than computing, explains why the shift to specialized, memory-optimized accelerators is a survival strategy for both

the cloud and the edge. The Jevons rebound completes the picture: efficiency alone cannot solve the crisis if it leads to exponential increases in usage.

Sustainability is an *engineering discipline*, not a *public relations exercise*. Carbon budgets, power delivery constraints, and cooling capacity impose hard limits on fleet expansion that no amount of marketing language can circumvent. The Jevons Paradox makes this especially clear: efficiency gains that reduce per-query cost routinely trigger demand explosions that overwhelm the original savings, meaning that technical optimization without governance is self-defeating. Organizations that treat sustainability as a solved problem after adopting a few efficiency techniques are repeating the same mistake that drove industrial energy consumption upward for two centuries.

The practitioner who can quantify lifecycle carbon across training, inference, and embodied manufacturing emissions, and who can design carbon-aware scheduling policies that respect grid carbon intensity, brings sustainability into the same engineering discipline as latency budgets or memory capacity. These skills transform sustainability from an abstract corporate goal into a measurable engineering constraint. When regulation, procurement, or carbon pricing makes environmental impact part of the operating envelope, this accounting becomes as fundamental to ML systems engineering as fault tolerance or security.

Key Takeaways: Efficiency alone is not enough

- **Power is a hard ceiling**: A fleet cannot compute past the megawatts, cooling, water, and grid capacity its site can supply. Sustainability is therefore an existence constraint on model scale, not a public-relations layer around an otherwise finished architecture.
- **Demand can outrun efficiency**: In the 2012–2019 scaling window, AI compute demand grew about 6.2× per year against a 1.5× annual hardware-efficiency curve. Without algorithmic and governance limits, the power wall arrives even as each operation gets cheaper.
- **Decode wastes energy structurally**: Autoregressive serving spends long periods bandwidth-bound, leaving accelerators drawing static power while waiting on memory. Sustainable serving needs quantization, sparsity, batching discipline, and memory-optimized hardware because FLOP efficiency alone misses the dominant loss.
- **Carbon starts before boot**: Up to 30 percent of lifecycle emissions can be embodied in hardware manufacturing before the first query runs. Procurement, hardware lifetime, reuse, and e-waste policy are MLOps decisions when lifecycle accounting is the boundary.
- **Location changes the footprint**: Carbon-aware scheduling can cut emissions by the chapter's 8 to 40 times representative regional factors when flexible jobs move across grids. Efficiency must be paired with workload placement and demand governance or Jevons rebound spends the savings.

For most of this book, efficiency has been the lever that made everything else possible, each chapter spending fewer bytes, fewer FLOPs, fewer joules for the same result. Sustainability is where that lever meets a wall it cannot move. A data center has a fixed power envelope, and no quantity of compute, communication, or coordination can draw more than the facility can deliver; the thermodynamic limit is the one constraint that closes over all three. Efficiency does not escape that ceiling, and the Jevons paradox shows it can hasten the approach, because cheaper computation is bought in greater quantity until the savings are spent. Staying beneath the limit therefore takes more than efficiency: it takes a deliberate decision about how much of the newly cheap capacity to actually use.

What's Next: From sustainability to responsibility

The environmental footprint of the ML fleet is quantifiable and constrained, which makes physical viability part of the system design problem. Security, robustness, and sustainability together form the engineering foundation of production AI. A system that is technically sound, however, can still cause social harm. In Chapter 16, we turn to the governance frameworks,

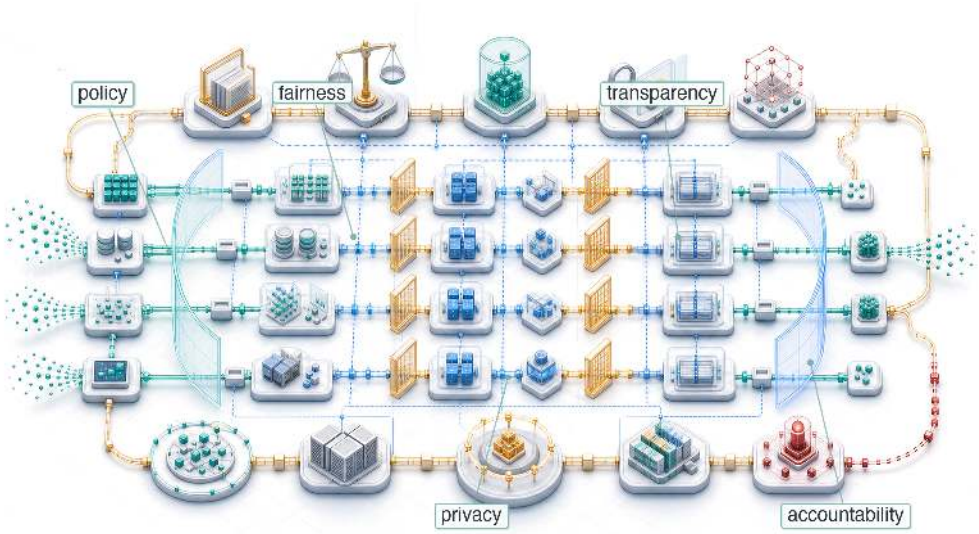
fairness requirements, and ethical guardrails that ensure our fleet serves the values of the society that built it, completing the transition from *how to build* the machine to *whom it serves*.

Research Questions: For further inquiry

- What lifecycle boundary is needed to decide whether an ML system is actually sustainable?
- How should a fleet design change when facility power, cooling, or grid capacity becomes the binding constraint?
- When does inference energy dominate training energy, and how should that change model and serving design?
- How should carbon-aware scheduling balance emissions reductions against latency, reliability, and security constraints?

16

Responsible AI

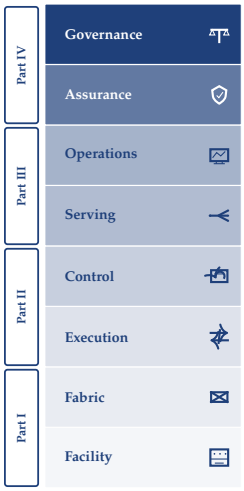


16.1 The Governance Imperative
16.2 Core Principles and the ML Lifecycle
16.3 Responsible AI Across Deployment Environments
16.4 Bias Detection and Fairness Monitoring
16.5 Privacy, Unlearning, and Robustness Mitigations
16.6 Explainability and Interpretability
16.7 Sociotechnical Dynamics
16.8 Implementation Challenges and AI Safety
16.9 Fallacies and Pitfalls
16.10 Summary

Purpose

Why do the systems that fail responsibility requirements fail to deploy at all, regardless of their technical capabilities?

In high-risk regulated settings, a model that cannot explain its decisions, document its behavior, or support audit can face deployment gates regardless of its technical capability. A model that exhibits demographic bias can create legal, operational, and reputational risk where discrimination rules apply. A model that cannot be audited cannot satisfy many enterprise governance requirements. These are not soft preferences but can become hard gates in regulated or enterprise settings: systems that fail them may be blocked regardless of accuracy, latency, or any other technical metric. The shift from responsible AI as ethics to responsible AI as engineering reflects this reality: for high-risk AI systems, legal and governance requirements can make transparency, oversight, documentation, and risk management part of deployment readiness. Organizations that treat responsibility as optional may discover their systems blocked at deployment by legal, regulatory, or reputational constraints that no amount of technical excellence can overcome. Responsibility has become infrastructure, not aspiration. In C^3 terms, responsibility is a coordination constraint on the entire fleet: safety and fairness guarantees must be established before compute is allowed to run.



Learning Objectives

- Translate fairness, transparency, accountability, privacy, and safety into measurable deployment gates
- Calculate fairness metrics and diagnose incompatibilities caused by base rates or imperfect prediction
- Design bias detection and fairness monitoring across demographic groups, drift, and fleet-scale feedback loops
- Select explanation methods under latency, compute, auditability, and user-contestability constraints
- Evaluate privacy, unlearning, and robustness mitigations against accuracy, compute, and governance requirements
- Analyze human-AI feedback loops, automation bias, and value conflicts in deployment contexts
- Assess governance structures that sustain documentation, accountability, safety review, and regulatory compliance

16.1 The Governance Imperative

In 2018, Reuters reported that Amazon had abandoned an experimental hiring algorithm trained on historical resume data after discovering it systematically penalized female candidates (Dastin 2018). The system satisfied many conventional operational requirements: it could be built, evaluated, and integrated into a recruiting workflow. Yet it had learned that past successful applicants were predominantly male, encoding historical bias rather than merit-based qualifications. The model was statistically optimized yet ethically disastrous, demonstrating that technical excellence can coexist with profound social harm.

In the fleet stack shown in figure 1.13, Responsible AI is the Governance Layer, the point where the system meets the real world. The lower layers supply compute, data movement, distributed execution, serving infrastructure, security controls, robustness mechanisms, and sustainability budgets. The Governance Layer adds the constraints that determine *why* the machine runs and *whom* it serves, ensuring that an otherwise correct system does not impose societal harms. If the iron law defines efficiency, Responsible AI defines stability: ensuring that the system's output does not destabilize the society it operates in. The instability is concrete: a model that outputs toxic content erodes user trust (feedback-loop instability), a model that discriminates degrades its own future training data (distributional instability), and a model that leaks privacy invites regulatory shutdown (operational instability). Responsible AI therefore belongs in the *objective function*, not only as a *constraint* checked on the finished solution.

Production AI has its own systems agenda, spanning architecture, infrastructure, data processing, security, and robustness challenges (Stoica et al. 2017). Security and privacy, robustness under distribution shift, and environmental sustainability sit directly beneath the Governance Layer. Those capabilities explain how to build and operate ML systems that are secure, robust, and sustainable. Responsible AI addresses the question that technical excellence alone cannot answer: whether those systems operate responsibly toward the people they affect.

The Amazon hiring incident reveals the central challenge of responsible AI: systems can be algorithmically sound while perpetuating injustice. The problem extends beyond individual bias to encompass transparency, accountability, privacy, and safety in systems affecting billions of lives daily. This is not the same problem as resilience. Resilient AI addresses threats to system integrity through adversarial attacks and hardware failures; responsible AI asks whether a properly functioning system produces outcomes consistent with human values and collective welfare. That distinction turns abstract ethical principles into engineering constraints. Fairness, transparency, and accountability must become quantifiable mechanisms and verifiable system properties, not final review items attached after the serving path is complete.

Software engineering provides precedent for this evolution. Early systems prioritized functional correctness alone. As complexity grew, the field developed methodologies for reliability engineering,

security assurance, and maintainability analysis. ML deployment creates a comparable maturation pressure, but at larger social scale: models can mediate credit allocation, medical diagnosis, educational assessment, and criminal justice. Unlike conventional software failures that manifest as crashes or data corruption, responsible AI failures can perpetuate systemic discrimination, compromise democratic institutions, and erode public confidence in beneficial technologies.

Definition 16.1: Responsible AI

Responsible AI is the practice of designing, auditing, and operating ML systems to measurable fairness, safety, privacy, and accountability standards—translating ethical principles into verifiable system properties that constrain model training, deployment decisions, and operational monitoring.

1. **Significance**: Responsible AI constraints impose real costs: fairness-aware training can lengthen training runs, real-time bias monitoring can consume serving-path latency, and on-demand explainability can require additional model evaluations or asynchronous worker capacity. At fleet scale, these safeguards must be budgeted like reliability or security controls. Conversely, a facial recognition system with a persistent subgroup error gap can affect millions of users before detection, creating regulatory and liability costs that dwarf the monitoring investment.
2. **Distinction**: Unlike AI ethics (which defines normative principles about what systems should do), responsible AI engineering defines technical mechanisms that enforce those principles—bias detection algorithms, differential privacy implementations, audit trails, and architectural guardrails that make compliance measurable and verifiable rather than aspirational.
3. **Common pitfall**: A frequent misconception is that responsible AI is a final compliance review applied to a finished model. Responsible constraints that are not designed in from the data collection stage typically require fundamental retraining to fix: a model trained on biased labels cannot be fairly calibrated by post-hoc threshold adjustment alone, as the learned representations themselves encode the bias.

Responsible AI therefore constitutes a systematic engineering discipline with four interconnected dimensions: translating ethical principles into measurable system requirements, detecting and mitigating harmful algorithmic behaviors, addressing **sociotechnical dynamics**[1] that extend beyond individual systems, and navigating implementation challenges within organizational and regulatory contexts. Public frameworks such as the National Institute of Standards and Technology (NIST) AI Risk Management Framework, ISO/IEC 42001 and 23894, the Organisation for Economic Co-operation and Development (OECD) AI Recommendation, and the United Nations Educational, Scientific and Cultural Organization (UNESCO) Recommendation on AI ethics all express this shift from abstract principles toward governable risk management, accountability, transparency, and human oversight (Tabassi 2023; International Organization for Standardization and International Electrotechnical Commission 2023b, 2023a; Organisation for Economic Co-operation and Development 2024; UNESCO 2022). Privacy mechanisms, robustness techniques, and sustainability metrics provide the technical foundation for that integration. The responsible-AI framework combines those capabilities with bias detection algorithms, privacy preservation mechanisms, organizational governance structures, and stakeholder engagement processes, treating responsible AI as fundamental to sound engineering practice rather than as supplementary constraints applied to finished systems.

That integration is a significant infrastructure investment. Section C.2 supplies the production-scale fleet baselines and serving-capacity figures against which this overhead can be estimated, letting an engineer size the tax relative to the same per-accelerator throughput numbers used for raw serving. This cost is essential infrastructure to be provisioned, analogous to how security (Chapter 13) and redundancy (Chapter 7) are budgeted in distributed systems.

Treating fairness, transparency, accountability, and privacy as rigorous engineering specifications rather than abstract ideals transforms responsible AI from aspiration into practice. The systematic approach that follows maps these core ethical principles directly onto the mechanical stages of the

1 **Sociotechnical System**: Coined by the Tavistock Institute in the 1950s to describe the interdependent relationship between humans and technology in the workplace. ML fleets are the ultimate sociotechnical systems: their "performance" is not merely a benchmark score but an emergent property of how model outputs interact with user behavior, legal frameworks, and physical resource constraints.

machine learning lifecycle, turning each into a concrete design constraint with measurable criteria. These four concerns must be read in order, because each depends on the one before it: technical solutions alone cannot resolve value conflicts, ethical principles without technical implementation remain aspirational, and isolated interventions fail without organizational support.

16.2 Core Principles and the ML Lifecycle

A continuous integration pipeline that detects a memory leak automatically blocks deployment. The same pipeline must block a deployment in which the system detects a 15 percent drop in accuracy specifically for elderly users. Responsible AI translates ethical principles into hard engineering invariants. Just as unit tests prevent logic regressions, the continuous integration/continuous deployment (CI/CD) pipeline must embed fairness, privacy, and accountability checks, treating a demographic bias exactly as a fatal software exception would be treated.

Fairness operates as a stability constraint. In control theory terms, fairness ensures that the system's error distribution is invariant across population subgroups. A system that violates this constraint is unstable: it will degrade its own training data through feedback loops (for example, predictive policing) and lose user trust, leading to eventual system collapse. This principle encompasses both statistical metrics and broader normative concerns about equity, justice, and structural bias. The fairness section below examines the formal mathematical definitions in detail.

The same control-plane view makes **Explainability** function as system observability: it is the mechanism by which the control plane exposes internal state to human operators (Phillips et al. 2020). Without explainability, the system is a black box running open loop, making it impossible to debug failure modes or verify safety constraints. Explanation mechanisms must support both individual decisions and overall model behavior. They may be generated after a decision is made to detail the reasoning process, known as post hoc explanations, or built into the model's design for transparent operation. Neural network architectures vary significantly in their inherent interpretability, with deeper networks generally being more difficult to explain. For this reason, explainability supports error analysis, regulatory compliance, and user trust.

That observability must be paired with **Transparency**: openness about how AI systems are built, trained, validated, and deployed. Transparency includes disclosure of data sources, design assumptions, system limitations, and performance characteristics. While explainability focuses on understanding outputs, transparency addresses the broader lifecycle of the system.

Once system behavior is visible, **Accountability** provides the mechanisms by which individuals or organizations are held responsible for AI outcomes. It involves traceability, documentation, auditing, and the ability to remedy harms. Accountability ensures that AI failures are not treated as abstract malfunctions but as consequences with real-world impact.

At the objective-function layer, **Value alignment**[2] requires AI systems to pursue goals that are consistent with human intent and ethical norms. In practice, this involves technical challenges, including reward design and constraint specification, and broader questions about whose values are represented and enforced.

The lifecycle also needs **Human oversight**: the role of human judgment in supervising, correcting, or halting automated decisions. This includes humans in the loop[3] during operation, as well as organizational structures that ensure AI use remains accountable to societal values and real-world complexity.

Privacy, robustness, and human oversight make the same point from different angles: principles alone do not ensure responsible systems. Translation from abstract ideals to concrete practice requires systematic integration across the ML lifecycle, where each principle manifests differently in data collection, model training, evaluation, deployment, and monitoring. The critical question is *how* these principles interact when they compete for priority.

16.2.1 Integrating principles across the ML lifecycle

Fairness, transparency, accountability, privacy, and safety define what it means for an AI system to behave ethically and predictably. Translating these principles into concrete constraints that guide how models are trained, evaluated, deployed, and maintained is the central engineering challenge.

Implementing these principles in practice requires understanding how each sets specific expectations for system behavior:

2 | **Value Alignment**: The problem of ensuring AI systems optimize for human values rather than proxy objectives. Stuart Russell formalized this in 2015, arguing that specifying objectives is harder than optimizing them. The engineering consequence: YouTube's pre-2017 recommendation algorithm optimized for click-through rate (a proxy for satisfaction), inadvertently promoting conspiracy content that maximized clicks while degrading user welfare—a misalignment that required redesigning the entire reward pipeline.

3 | **Human-in-the-Loop (HITL)**: A design pattern where humans actively participate in model decisions rather than being replaced by automation. The systems trade-off is latency vs. safety: HITL adds 100 ms to 30+ seconds per decision depending on domain. Large content-moderation systems have historically paired automated triage with thousands of human reviewers, demonstrating that the pattern scales only with proportional human infrastructure cost. In ML serving architectures, HITL requires routing logic, confidence thresholds for escalation, and queue management that fundamentally reshape the inference pipeline.

- **Fairness**: Models must treat different subgroups in ways that account for historical biases.
- **Explainability**: Model decisions must be understandable to developers, auditors, and end users.
- **Privacy**: Data collection and use must respect consent, purpose, and access boundaries.
- **Accountability**: Responsibilities must be assigned, tracked, and enforced throughout the system lifecycle.
- **Safety**: Models must behave reliably even in uncertain or shifting environments.

The principles overlap, but each creates a distinct engineering obligation that must be designed into the system lifecycle.

Table 16.1 maps the responsible AI lifecycle across the major phases of ML system development: data collection, model training, evaluation, deployment, and monitoring. Fairness and privacy constraints begin at data collection; robustness and accountability become most critical during deployment and oversight. Explainability spans the full lifecycle, supporting model debugging at design time and user-facing justification at serving time. The mapping reinforces that responsible AI is a multiphase architectural commitment, not a post-hoc compliance step.

Table 16.1: Responsible AI Lifecycle: Embedding fairness, explainability, privacy, accountability, and robustness throughout the ML system lifecycle, from data collection to monitoring, ensures these principles become architectural commitments rather than post hoc considerations. The table maps these principles to specific development phases, revealing how proactive integration addresses potential risks and promotes trustworthy AI systems.

Principle	Data Collection	Model Training	Evaluation	Deployment	Monitoring
Fairness	Representative sampling	Bias-aware algorithms	Group-level metrics	Threshold adjustment	Subgroup performance
Explainability	Documentation standards	Interpretable architecture	Model behavior analysis	User-facing explanations	Explanation quality logs
Transparency	Data source tracking	Training documentation	Performance reporting	Model cards	Change tracking
Privacy	Consent mechanisms	Privacy-preserving methods	Privacy impact assessment	Secure deployment	Access audit logs
Accountability	Governance frameworks	Decision logging	Audit trail creation	Override mechanisms	Incident tracking
Robustness	Quality assurance	Robust training methods	Stress testing	Failure handling	Performance monitoring

16.2.1.1 Resource requirements and equity implications

Implementing responsible AI principles requires computational resources that vary significantly across techniques and deployment contexts. These resource requirements create multifaceted equity considerations that extend beyond individual organizations to encompass broader social and environmental justice concerns. Organizations with limited computing budgets may be unable to implement comprehensive responsible AI protections, potentially creating disparate access to ethical safeguards. Large-scale model deployments that depend on accelerator clusters and low-latency connectivity can exclude users whose networks or devices cannot meet those assumptions.

Environmental justice concerns compound these access barriers through the engineering reality that responsible AI techniques impose energy and capacity costs. The concrete numbers later in the chapter are sizing assumptions for methods the chapter unpacks: differential privacy may require extra training iterations, real-time fairness monitoring consumes serving-path resources, and Shapley-value explanations can require many additional model evaluations unless they are approximated, cached, or generated asynchronously. These computational requirements translate directly into infrastructure demands: a high-traffic system serving responsible AI features to 10 million users requires substantial additional data-center capacity compared to unconstrained models.

The geographic distribution of this computational infrastructure creates systematic inequities that engineers must consider in system design. Data centers supporting AI workloads concentrate in regions with low electricity costs and favorable regulations, areas that often correlate with lower-income communities that experience increased pollution, heat generation, and electrical grid strain

while frequently lacking the high-bandwidth connectivity needed to access the AI services these facilities enable. This creates a feedback loop where computational equity depends not only on algorithmic design but on infrastructure placement decisions that affect both system performance and community welfare. The resource-overhead comparison later in this chapter quantifies the performance characteristics of specific responsible AI techniques.

16.2.2 Transparency and explainability

Machine learning systems are frequently criticized for their lack of interpretability. In many cases, models operate as opaque "black boxes," producing outputs that are difficult for users, developers, and regulators to understand or scrutinize. This opacity presents a significant barrier to trust, particularly in high stakes domains such as criminal justice, healthcare, and finance, where accountability and the right to recourse are important. For example, the COMPAS algorithm, used in the United States to assess recidivism risk, was found to exhibit racial bias[4]. The proprietary nature of the system, combined with limited access to interpretability tools, hindered efforts to investigate or address the issue.

Explainability is the capacity to understand how a model produces its predictions. It includes both **local explanations**, which clarify individual predictions, and **global explanations**, which describe the model's general behavior. Transparency, by contrast, encompasses openness about the broader system design and operation. This includes disclosure of data sources, feature engineering, model architectures, training procedures, evaluation protocols, and known limitations. Transparency also involves documentation of intended use cases, system boundaries, and governance structures.

The importance of explainability and transparency extends beyond technical considerations to legal requirements. In many jurisdictions, these principles are legal obligations rather than optional guidance. Under GDPR, solely automated decisions with legal or similarly significant effects trigger Article 22 safeguards and related transparency duties, including meaningful information about the logic involved under access and notice provisions, subject to statutory exceptions (European Parliament and Council of the European Union 2016; European Data Protection Board 2018).[5] Similar regulatory pressures in other domains reinforce the need to treat explainability and transparency as core architectural requirements.

Implementing these principles requires anticipating the needs of different stakeholders, whose competing values and priorities are examined comprehensively in section 16.7.3. Designing for explainability and transparency therefore necessitates decisions about how and where to surface relevant information across the system lifecycle.

Transparency and explainability also support system reliability over time. As models are retrained or updated, mechanisms for interpretability and traceability allow detection of unexpected behavior, enable root cause analysis, and support governance. Embedded into the structure and operation of a system, these mechanisms provide the foundation for trust, oversight, and alignment with institutional and societal expectations.

While transparency and explainability enable stakeholders to understand system behavior, they do not guarantee that this behavior is equitable. A model can be fully transparent about how it makes decisions while still systematically disadvantaging certain groups. This distinction motivates the examination of fairness as a separate, complementary principle.

16.2.3 Fairness in machine learning

Automated decisions in hiring, lending, and healthcare affect millions of people, and historical data used to train these systems encodes decades of structural inequality. The engineering question is how to formalize "equitable treatment" in a way that a system can measure and enforce.

Definition 16.2: Algorithmic fairness

Algorithmic Fairness is the measurable property that a model's error distribution or outcomes are invariant (or bounded in variation) across protected demographic groups.

1. **Significance**: It transforms fairness from an intuition into a multi-objective optimization problem. Within the iron law, achieving fairness often requires trading off total accuracy

4 **COMPAS (Correctional Offender Management Profiling for Alternative Sanctions)**: ProPublica's 2016 analysis found Black defendants were falsely flagged as future criminals at nearly twice the rate of white defendants (45 percent vs. 23 percent false positive rate). The proprietary, black-box nature of the system limited full inspection of model internals and training data, demonstrating a compounding failure: bias in the model coupled with opacity in the serving architecture made the system harder to audit and harder to debug.

5 **GDPR Article 22**: Article 22 gives data subjects protections against decisions based solely on automated processing that produce legal or similarly significant effects, with exceptions for contract necessity, authorization by law, or explicit consent plus safeguards (European Parliament and Council of the European Union 2016). Related notice and access provisions, including Article 15, require meaningful information about the logic involved for qualifying automated decision-making, and EDPB guidance treats those duties as part of a broader profiling and automated-decision governance regime (European Data Protection Board 2018). For ML systems, this creates an architectural constraint: high-stakes automated-decision systems need explanation, recourse, and governance infrastructure rather than opaque serving-only APIs.

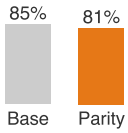
(Accuracy) for group-specific calibration, where calibration is checked separately within each protected group, ensuring that the system's benefits and harms are distributed equitably.

2. **Distinction**: Unlike Average Accuracy (which hides disparities in the aggregate), Algorithmic Fairness focuses on the Subgroup Distribution $(p(Y \mid X, \text{Group}))$, identifying where the model fails for minority populations.
3. **Common pitfall**: A frequent misconception is that there is a single "fair" solution. In reality, different fairness definitions (for example, demographic parity vs. equalized odds) are often mathematically incompatible and not jointly satisfiable except in special cases: satisfying one necessitates violating another, requiring explicit policy choices by the engineer.

Fairness in machine learning presents complex challenges that extend beyond transparency. The core requirement is that automated systems not disproportionately disadvantage protected groups. Because these systems are trained on historical data, they are susceptible to reproducing and amplifying patterns of systemic bias embedded in that data. Without careful design, machine learning systems may unintentionally reinforce social inequities rather than mitigate them.

A widely studied example comes from the healthcare domain. An algorithm[6] used to allocate care management resources in US hospitals was found to systematically underestimate the health needs of Black patients (Obermeyer et al. 2019). The model used healthcare expenditures as a proxy for health status, but due to longstanding disparities in access and spending, Black patients were less likely to incur high costs. As a result, the model inferred that they were less sick, despite often having equal or greater medical need. This case illustrates how seemingly neutral design choices such as proxy variable selection can yield discriminatory outcomes when historical inequities are not properly accounted for. Enforcing fairness constraints on such models incurs a measurable cost, a phenomenon known as the **fairness tax**.

The fairness-tax calculation isolates one criterion: demographic parity, which requires equal positive decision rates across groups. It does not represent every fairness goal; it shows the system cost of enforcing that criterion before the section compares demographic parity with equalized odds and equality of opportunity.



Fairness constraints cost a few points of accuracy, the responsibility tax.

6 **Healthcare Algorithm Scale**: The Optum algorithm affected approximately 200 million Americans annually, using healthcare expenditure as a proxy for health need. Because Black patients historically incurred lower costs due to access disparities, the model systematically underestimated their severity, reducing Black enrollment in high-risk care programs by 50 percent. Correcting the proxy would have increased Black patient identification from 17.7 percent to 46.5 percent, quantifying the cost of a single proxy variable choice at population scale.

Napkin Math 16.1: The fairness tax

Problem: Consider a credit model with 85 percent accuracy. Group A (majority) has a 20 percent default rate. Group B (minority) has a 40 percent default rate due to systemic factors. If demographic parity (equal approval rates) is enforced, what happens to accuracy?

Math:

The approval-rate facts define the policy change; the accuracy values below are measured on the held-out validation set for this scenario, where the additional Group B approvals carry higher default risk.

1. **Unconstrained**: Model approves everyone with predicted default prob < 30 percent.
   - Group A approval: 80 percent.
   - Group B approval: 60 percent.
   - Total Accuracy: 85 percent.
2. **Constrained (parity)**: Must approve Group B at 80 percent rate.
   - New threshold for Group B: Approve default prob < 50 percent.
   - This forces the model to approve many risky applicants in Group B.
   - New Total Accuracy: 81 percent.

Systems insight: Fairness is not free. Enforcing parity cost 4 percentage points of accuracy (about a 4.7 percent relative accuracy drop in this credit-scoring scenario). This is the fairness tax, the explicit cost of correcting for historical bias.

Practitioners need formal methods to evaluate fairness given these risks of perpetuating bias. A range of formal criteria have been developed that quantify how models perform across groups defined by sensitive attributes. Before introducing these definitions, it helps to frame the notation as a way to express competing fairness goals, not as a proof exercise.

Systems Perspective 16.1: Fairness notation as engineering vocabulary

Before examining formal definitions, the fundamental challenge is that fairness can mean different operational constraints. A system may treat everyone identically, account for different baseline conditions, optimize equal outcomes, preserve equal opportunity, or enforce equal treatment. These choices lead to different mathematical criteria, each capturing a different aspect of fairness.

The next subsections introduce formal fairness definitions using probability notation. These metrics (demographic parity, equalized odds, equality of opportunity) appear throughout ML fairness literature and shape regulatory frameworks. Focus on understanding the intuition: what each metric measures and why it matters, rather than mathematical proofs. The concrete examples after each definition illustrate practical application. If probability notation is unfamiliar, start with the verbal descriptions and return to the formal definitions later.

Suppose a model $h(x)$ predicts a binary outcome, such as loan repayment, and let A represent a sensitive attribute with subgroups a and b. The field uses three widely adopted fairness definitions:

16.2.3.1 Demographic parity

Definition 16.3: Demographic parity

Demographic Parity is the fairness constraint where a model's positive prediction rate is independent of group membership ($\Pr(h(x) = 1 \mid A = a) = \Pr(h(x) = 1 \mid A = b)$).

1. **Significance**: It is the simplest and most restrictive fairness metric. It requires the model to produce Equal Outcomes across groups, regardless of the underlying base-rate differences in the dataset.
2. **Distinction**: Unlike Equalized Odds (which focuses on error rates like False Positives), Demographic Parity focuses only on the Final Prediction, ignoring the relationship between the prediction and the ground truth.
3. **Common pitfall**: A frequent misconception is that Demographic Parity ensures "fairness." In reality, it can force the model to sacrifice Calibration: to meet the parity constraint, the model may have to intentionally misclassify qualified individuals in one group or unqualified individuals in another.

Demographic parity asks whether favorable outcomes, such as loan approval or treatment referral, occur at equal rates across subgroups defined by a sensitive attribute A. Formally, the model satisfies demographic parity if:

$$\Pr\big(h(x) = 1 \mid A = a\big) = \Pr\big(h(x) = 1 \mid A = b\big)$$

In the healthcare example, this criterion would ask whether Black and white patients were referred for care at the same rate, regardless of their underlying health needs. That may sound like equal access, but it ignores real differences in medical status and risk, potentially overcorrecting when needs are not evenly distributed. The limitation of ignoring base-rate differences motivates more nuanced fairness criteria.

16.2.3.2 Equalized odds

Equalized odds requires that the model's predictions are conditionally independent of group membership given the true label. Specifically, the true positive and false positive rates must be equal across groups:

$$\Pr(h(x) = 1 \mid A = a, Y = y) = \Pr(h(x) = 1 \mid A = b, Y = y), \quad \text{for } y \in \{0, 1\}.$$

That is, for each true outcome $Y = y$, the model should produce the same prediction distribution across groups $A = a$ and $A = b$. The model should therefore behave similarly across groups for individuals with the same true outcome, whether they qualify for a positive result or not. It ensures that errors (both missed and incorrect positives) are distributed equally.

Applied to the medical case, equalized odds would ensure that patients with the same actual health needs (the true label Y) are equally likely to be correctly or incorrectly referred, regardless of race. The original algorithm violated this by under-referring Black patients who were equally or more sick than their white counterparts, highlighting unequal true positive rates.

16.2.3.3 Equality of opportunity

Equality of opportunity is a less stringent relaxation of equalized odds that focuses only on the true positive rate (Hardt et al. 2016). It requires that, among individuals who should receive a positive outcome, the probability of receiving one is equal across groups:

$$\Pr(h(x) = 1 \mid A = a, Y = 1) = \Pr(h(x) = 1 \mid A = b, Y = 1).$$

Equality of opportunity ensures that qualified individuals, who have $Y = 1$, are treated equally by the model regardless of group membership.

In our running example, this measure would ensure that among patients who do require care, both Black and white individuals have an equal chance of being identified by the model. In the case of the US hospital system, the algorithm's use of healthcare expenditure as a proxy variable led to a failure in meeting this criterion: Black patients with significant health needs were less likely to receive care due to their lower historical spending. The worked example below uses the same loan decisions to calculate all three criteria, showing why a single model can fail each fairness test in a different way.

Example 16.1: Calculating fairness metrics

Consider a simplified loan approval model evaluated on 200 applicants, evenly split between two demographic groups (Group A and Group B). The model makes predictions, and we later observe actual repayment outcomes:

Group A (100 applicants):

- Model approved: 70 applicants (40 actually repaid, 30 defaulted)
- Model rejected: 30 applicants (5 actually would have repaid, 25 would have defaulted)

Group B (100 applicants):

- Model approved: 40 applicants (30 actually repaid, 10 defaulted)
- Model rejected: 60 applicants (20 actually would have repaid, 40 would have defaulted)

Calculating demographic parity:

$$\Pr(h(x) = 1 \mid A = a) = \frac{70}{100} = 0.70$$
$$\Pr(h(x) = 1 \mid A = b) = \frac{40}{100} = 0.40$$

Disparity: $0.70 - 0.40 = 0.30$ (30 percentage point gap)

The model violates demographic parity by approving Group A applicants at substantially higher rates, regardless of actual repayment ability.

Calculating equality of opportunity (true positive rate):
Among applicants who *would actually repay* ($Y = 1$):

$$\Pr(h(x) = 1 \mid A = a, Y = 1) = \frac{40}{40+5} = \frac{40}{45} \approx 0.89$$
$$\Pr(h(x) = 1 \mid A = b, Y = 1) = \frac{30}{30+20} = \frac{30}{50} = 0.60$$

Disparity: $0.89 - 0.60 = 0.29$ (29 percentage point gap in TPR)
The model violates equality of opportunity: among qualified applicants who would repay, Group A members are correctly approved 89 percent of the time while Group B members are only approved 60 percent of the time.
Calculating equalized odds (true positive rate + false positive rate):
The TPR values are calculated above. For false positive rates among applicants who *would not repay* ($Y = 0$):

$$\Pr(h(x) = 1 \mid A = a, Y = 0) = \frac{30}{30+25} = \frac{30}{55} \approx 0.55$$
$$\Pr(h(x) = 1 \mid A = b, Y = 0) = \frac{10}{10+40} = \frac{10}{50} = 0.20$$

The model also has unequal false positive rates: it incorrectly approves 55 percent of Group A applicants who will default, but only 20 percent of Group B applicants who will default. This reveals the model is more "generous" with Group A even when they will not repay.
Systems insight: This model violates all three fairness criteria. Addressing one criterion does not automatically satisfy others. In fact, the impossibility theorems prove these criteria can conflict mathematically.

The worked example above revealed that this loan approval model violates all three fairness criteria simultaneously. This is not merely poor model design but reflects a fundamental mathematical tension that any classifier must confront when base rates differ between groups. These tensions motivate the impossibility results that constrain what any fair classifier can achieve.

These definitions capture different aspects of fairness and are generally incompatible[7] (Kleinberg et al. 2016; Chouldechova 2017). A university admissions example illustrates the tension concretely. Goal 1 (Demographic Parity) would be to admit students so that the admitted class reflects the demographics of the applicant pool, perhaps 50 percent from Group A and 50 percent from Group B. Goal 2 (Equal Opportunity) would be to ensure that among all qualified applicants, the admission rate is the same across groups, so that 80 percent of qualified Group A applicants get in and 80 percent of qualified Group B applicants get in. If one group has a higher proportion of qualified applicants, achieving demographic parity (Goal 1) requires rejecting some of their qualified applicants, violating equal opportunity (Goal 2), as figure 16.1 visualizes. *No mathematical fix exists; the choice is a value judgment about which definition of fairness to prioritize.*

The fairness impossibility law (Principle 20) generalizes exactly what the admissions example just demonstrated: except in special cases such as equal base rates or perfect prediction, calibrated risk scores generally cannot also satisfy equal error-rate conditions across groups, and demographic parity adds further conflict because it constrains selection rates rather than score calibration or error rates. Satisfying one criterion may preclude another, so engineers must treat fairness metrics like latency budgets, explicit trade-offs chosen by stakeholders, enforced by the system, and monitored for violation. Determining which metric to prioritize requires careful consideration of the application context, potential harms, and stakeholder values as detailed in section 16.7.3.

Figure 16.2 makes this trade-off concrete by sweeping a classification threshold across a synthetic scenario with differing group base rates. At every threshold, at least one fairness metric is substantially violated, illustrating the broader operational lesson: calibrated and equal-error-rate criteria

7 **Fairness Impossibility Theorems**: Kleinberg et al. (2016) and Chouldechova (2017) prove incompatibility results for calibrated risk scores and error-rate balance when base rates differ, except in constrained special cases such as perfect prediction or equal base rates. Demographic parity is a distinct constraint on selection rates that can also conflict with calibration or error-rate goals in practice. The systems consequence is fundamental: no amount of engineering can satisfy every normative fairness criterion in all settings, so fairness becomes a constrained multi-objective optimization requiring explicit policy choices about which criterion to prioritize for a given deployment context.

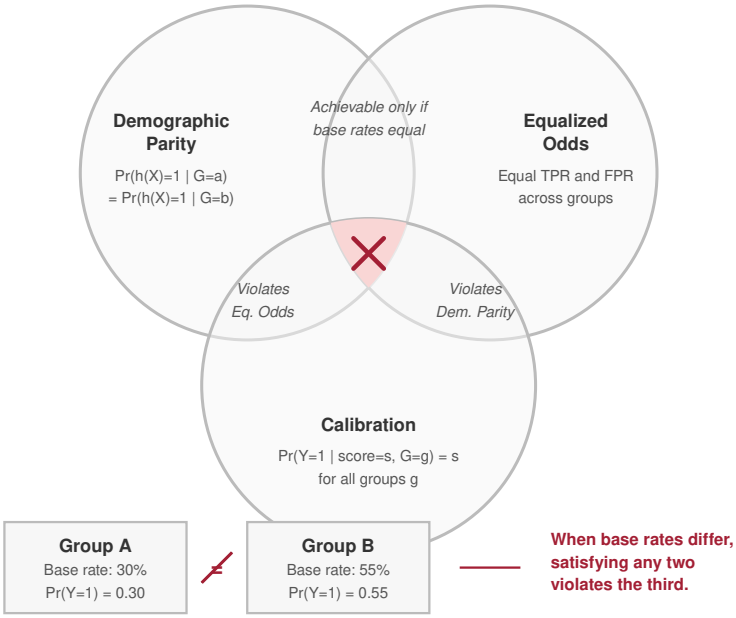


Figure 16.1: Fairness Impossibility Trade-Offs: Visualizing why common fairness criteria can conflict. Calibrated risk scores generally cannot also satisfy equalized error-rate conditions when base rates differ and prediction is imperfect; demographic parity adds a separate selection-rate constraint that may conflict with either goal. This forces engineers to make explicit normative choices based on the application context.

generally conflict under unequal base rates, and selection-rate criteria such as demographic parity add further constraints.

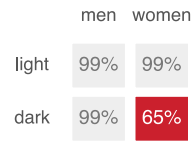
Recognizing these tensions, operational systems must treat fairness as a constraint that informs decisions throughout the machine learning lifecycle. It is shaped by how data are collected and represented, how objectives and proxies are selected, how model predictions are thresholded, and how feedback mechanisms are structured. For example, a choice between ranking vs. classification models can yield different patterns of access across groups, even when using the same underlying data.

Fairness metrics help formalize equity goals but are often limited to predefined demographic categories. In practice, these categories may be too coarse to capture the full range of disparities present in real-world data.

16.2.3.4 Intersectional fairness

A critical limitation of standard fairness analysis is that it often evaluates single axes of identity (for example, race or gender) independently. This can mask profound disparities that exist at the intersection of these attributes.

For example, a facial recognition system might have 99 percent accuracy for "Men" and 99 percent accuracy for "Light-Skinned People", but only 65 percent accuracy for "Dark-Skinned Women" (Buolamwini and Gebru 2018). If the audit only checks Race and Gender separately, the model appears fair. This phenomenon, sometimes called **Fairness Gerrymandering**, requires evaluating model performance on intersectional subgroups (for example, Race×Gender) to detect and mitigate compounded biases.



Error concentrates at the intersection: dark-skinned women.

A principled approach to fairness must account for overlapping and intersectional identities, ensuring that model behavior remains consistent across subgroups that may not be explicitly labeled in advance. Recent work in this area emphasizes the need for predictive reliability across a wide

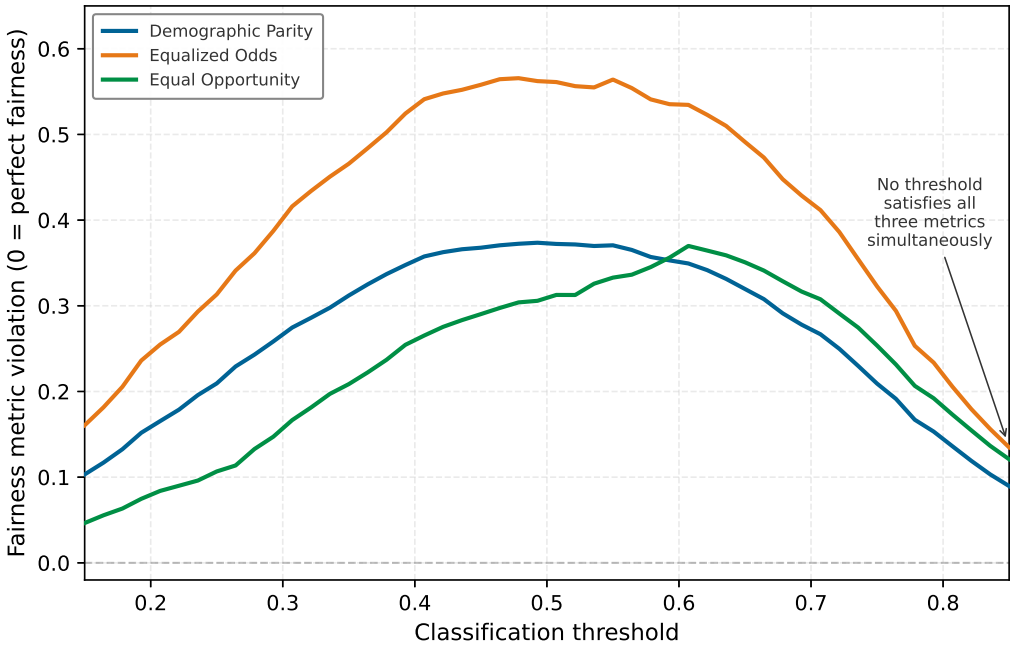


Figure 16.2: Fairness Metric Disagreement Across Thresholds: Three standard fairness metrics, demographic parity, equalized odds, and equal opportunity, computed on a classifier with differing group base rates as the classification threshold varies. At every threshold, at least one metric is substantially violated, illustrating that unequal base rates and imperfect prediction force practical trade-offs among selection-rate, error-rate, and opportunity criteria.

range of population slices (Hébert-Johnson et al. 2018), reinforcing the idea that fairness must be considered a system-level requirement, not a localized adjustment. This expanded view of fairness highlights the importance of designing architectures, evaluation protocols, and monitoring strategies that support more nuanced, context-sensitive assessments of model behavior.

16.2.3.5 Quantitative fairness measurement

Formal fairness criteria become operational only when practitioners can measure the degree of violation and establish actionable thresholds for intervention. The resulting mathematical framework quantifies disparities and determines when they warrant corrective action.

Four point metrics turn the fairness criteria from section 16.2.3 into numbers an auditor can threshold. Each one equalizes a different quantity, so each exposes a different failure of the same classifier. Table 16.2 defines them and scores them on the running loan example, where Group A is approved at 0.70 and Group B at 0.40, qualified-applicant true-positive rates are 0.89 and 0.60, and unqualified-applicant false-positive rates are 0.55 and 0.20.

Table 16.2: Fairness Metrics on the Loan Example: Four group-fairness metrics, each equalizing a different quantity, evaluated on the loan approval model. The disparate impact ratio is the legally recognized screening test; the additive differences are easier to track over time; the average odds difference captures both true-positive and false-positive error gaps.

Metric	Equalizes	Formula (a, b)	Loan value	Choose it when
Disparate impact ratio (four-fifths rule) (Feldman et al. 2015)	lower approval rate divided by higher approval rate	$\frac{\min_g \Pr(h=1 \mid g)}{\max_g \Pr(h=1 \mid g)}$	$\frac{0.40}{0.70} = 0.57$	a legally recognized screen is needed; DI < 0.8 flags disparate impact

Metric	Equalizes	Formula (a, b)	Loan value	Choose it when
Statistical parity difference (Calders and Verwer 2010)	additive gap in approval rates	$\Pr(h=1 \mid a) - \Pr(h=1 \mid b)$	$0.70 - 0.40 = 0.30$	comparing across many groups or tracking drift; common audit threshold $\lvert\text{SPD}\rvert \leq 0.10$
Equal opportunity difference (Hardt et al. 2016)	true-positive rates among qualified applicants	$\Pr(h=1 \mid a, Y=1) - \Pr(h=1 \mid b, Y=1)$	$0.89 - 0.60 = 0.29$	the harm is denying deserving individuals
Average odds difference (Hardt et al. 2016; Bellamy et al. 2019; Bird et al. 2020)	both true-positive and false-positive rates	$\frac{1}{2}\left[\lvert\Delta\text{TPR}\rvert + \lvert\Delta\text{FPR}\rvert\right]$	$\frac{1}{2}[0.29 + 0.35] = 0.32$	both error types matter; AOD $= 0$ at perfect equalized odds

The four metrics disagree by construction: the disparate impact ratio of 0.57 means the lower-approval group receives approvals at only that fraction of the higher-approval group's rate, already breaching the four-fifths rule. The additive 0.30 gap, the 0.29 opportunity gap, and the 0.32 average odds gap each quantify a distinct dimension of the same disparity. No single number is sufficient, which is why the impossibility results above force an explicit choice about which metric a deployment will prioritize.

Calibration A model satisfies **score calibration** with respect to a sensitive attribute if, among individuals assigned score s by the model, the fraction with positive outcomes is equal across groups (Kleinberg et al. 2016):

$$\Pr(Y=1 \mid s(x)=s, A=a) = \Pr(Y=1 \mid s(x)=s, A=b), \quad \forall s$$

Score calibration is a per-score condition; for hard binary predictions, one related parity condition is equal **positive predictive value** (precision) among predicted positives, while score calibration is the stronger requirement that the equality holds at every score level:

$$\text{PPV}(a) = \frac{\Pr(Y=1, h(x)=1 \mid A=a)}{\Pr(h(x)=1 \mid A=a)} = \text{PPV}(b)$$

From the loan example:

$$\text{PPV}(a) = \frac{40}{70} = 0.571$$
$$\text{PPV}(b) = \frac{30}{40} = 0.750$$

The positive predictive value gap is $0.750 - 0.571 = 0.179$. Group B's predicted positives are actually positive 75 percent of the time, while Group A's are only 57 percent accurate. This violates predictive parity for hard decisions and reveals that the model is less reliable when predicting approval for Group A. Demonstrating score calibration itself would require comparing outcome frequencies within matched score ranges across groups.

Calibration is critical for high stakes decisions where individuals rely on predicted probabilities. A miscalibrated model systematically over or underpredicts risk for specific groups, leading to misallocated resources and eroded trust.

Fairness metrics in practice Defining these metrics is the easy part; computing them continuously across a deployed fleet is where the engineering cost appears. Before turning to the mechanics of threshold trade-offs and significance testing, it is worth seeing what these metrics demand at production scale, because that cost shapes which of them a system can afford to monitor.

Measurement overhead arises because computing group specific metrics requires maintaining separate statistics for each protected group. For k groups and m metrics, this requires $\mathcal{O}(km)$

additional counters and $\mathcal{O}(km)$ statistical tests per evaluation cycle. In high throughput systems (>10K QPS), this overhead must be managed through sampling or asynchronous aggregation.

Data requirements pose challenges because fairness auditing requires ground truth labels (Y) and sensitive attributes (A) for a representative sample. In federated or privacy preserving settings, obtaining this data may conflict with privacy goals. Techniques like encrypted aggregate statistics or differential privacy for group metrics can help reconcile fairness monitoring with privacy requirements.

Sensitive-attribute availability is therefore an engineering design choice, not a background assumption. A system may collect attributes directly with consent, infer them for aggregate audit only, escrow them in a restricted service, compute encrypted or differentially private group statistics, audit offline on sampled labels, or declare a metric unmeasurable for that deployment. Each choice changes both the fairness evidence available to the organization and the privacy risk created by the monitoring pipeline.

Threshold selection demands domain expertise and stakeholder input to establish acceptable disparity thresholds. Legal thresholds (for example, four-fifths rule) provide starting points, but context-specific harm assessments should inform final values. Document threshold rationale to support audits and regulatory compliance.

Temporal stability requires monitoring fairness metrics over time to detect degradation due to distribution shift, feedback loops, or model updates. Continuous monitoring with automated alerting (for example, "alert if $|\text{SPD}| > 0.15$ for 7 consecutive days") enables proactive intervention before harms accumulate.

Threshold setting and fairness trade-offs In practice, fairness metrics can be manipulated by adjusting classification thresholds per group. Given a scoring function $s(x)$ (for example, predicted probability), define group-specific thresholds $\tau_{\text{thr},a}$ and $\tau_{\text{thr},b}$ such that $h_a(x) = \mathbb{1}[s(x) \geq \tau_{\text{thr},a}]$ for group a and similarly for group b.

To achieve demographic parity, solve:

$$\Pr(s(x) \geq \tau_{\text{thr},a} \mid A = a) = \Pr(s(x) \geq \tau_{\text{thr},b} \mid A = b)$$

To achieve equal opportunity, solve:

$$\Pr(s(x) \geq \tau_{\text{thr},a} \mid A = a, Y = 1) = \Pr(s(x) \geq \tau_{\text{thr},b} \mid A = b, Y = 1)$$

For equalized odds, both true positive and false positive rate constraints must hold simultaneously. This is a constrained optimization problem that can be solved via postprocessing (Hardt et al. 2016).

However, threshold adjustment has limitations. If base rates differ substantially between groups (that is, $\Pr(Y = 1 \mid A = a) \neq \Pr(Y = 1 \mid A = b)$), achieving one fairness criterion through thresholding will necessarily violate others due to the impossibility theorems. The resulting accuracy cost can be quantified directly.

Example 16.2: Engineering metric: The cost of fairness

Trade-off: Satisfying a fairness constraint often requires deviating from the optimal accuracy threshold. This deviation is the "fairness tax."

Scenario: A credit model scores applicants from 0 to 100, using the same held-out credit validation set as the fairness-tax example.

- **Group A (majority)**: Mean score 70, High repayment rate. Optimal Threshold $= 60$.
- **Group B (minority)**: Mean score 50, Lower repayment rate (due to systemic factors).

Unconstrained optimization (max profit):

- Threshold $= 60$ for everyone.
- $\text{Approval}_A = 80$ percent, $\text{Approval}_B = 60$ percent.
- Accuracy $= 85$ percent.

Fairness constrained (demographic parity):

- Constraint: Group B Approval must equal Group A (80 percent).
- New Threshold$_B = 50$.
- **Result**: Group B false positives increase. Overall accuracy drops to 81 percent.

Systems insight: The "Cost of Fairness" is 4 percentage points of accuracy, about a 4.7 percent relative accuracy drop in this scenario. The engineering decision requires weighing this measured accuracy trade-off against social equity gains.

The preceding example measured the aggregate accuracy drop from imposing demographic parity, but it did not show how the system enforces the constraint at decision time. In practice, equalization requires setting group-specific thresholds: one threshold for the majority group and a different, lower threshold for the minority group. Differential thresholds require access to sensitive attributes at inference time and raise concerns about explicit group-based treatment, which may itself be considered unfair or illegal in certain jurisdictions. The mechanics become clearer in a concrete threshold-setting scenario.

Example 16.3: Threshold for equal opportunity

Consider a credit scoring model that outputs a probability $s(x) \in [0,1]$. Historical data shows:
Group A: 1000 applicants, 600 would repay ($Y = 1$), 400 would default ($Y = 0$)

- Score distribution for $Y = 1$: Mean $\mu_A^+ = 0.72$, SD $\sigma_A^+ = 0.15$
- Score distribution for $Y = 0$: Mean $\mu_A^- = 0.45$, SD $\sigma_A^- = 0.18$

Group B: 1000 applicants, 400 would repay ($Y = 1$), 600 would default ($Y = 0$)

- Score distribution for $Y = 1$: Mean $\mu_B^+ = 0.65$, SD $\sigma_B^+ = 0.16$
- Score distribution for $Y = 0$: Mean $\mu_B^- = 0.40$, SD $\sigma_B^- = 0.17$

Assuming the conditional score distributions are approximately normal with the stated means and standard deviations, using a single threshold $\tau_{\text{thr}} = 0.60$ for both groups yields true positive rates:

$$\text{TPR}_a = \Pr(s(x) \geq 0.60 \mid A = a, Y = 1) \approx 0.79$$
$$\text{TPR}_b = \Pr(s(x) \geq 0.60 \mid A = b, Y = 1) \approx 0.62$$

This 17 percentage point gap violates equal opportunity. To equalize Group B's TPR to Group A's approximately 0.79 rate, we could lower Group B's threshold to $\tau_{\text{thr},b} = 0.52$ while keeping $\tau_{\text{thr},a} = 0.60$. However, this adjustment increases Group B's false positive rate from about 0.12 to about 0.24, degrading precision for Group B applicants from about 0.78 to about 0.69.

Systems insight: This illustrates the fundamental trade-off: achieving equal opportunity through threshold adjustment comes at the cost of reduced calibration and increased false positives for the group receiving the lower threshold. The decision involves weighing opportunity equity against prediction reliability.

Measuring fairness violations statistically To determine whether observed disparities are statistically significant rather than sampling noise, practitioners should compute confidence intervals and conduct hypothesis tests. For demographic parity, test the null hypothesis $H_0 : \Pr(h(x) = 1 \mid A = a) = \Pr(h(x) = 1 \mid A = b)$ using a two-proportion z-test. The test statistic is:

$$z = \frac{\hat{p}_a - \hat{p}_b}{\sqrt{\hat{p}(1-\hat{p})\left(\frac{1}{n_a} + \frac{1}{n_b}\right)}}$$

where $\hat{p}_a$ and $\hat{p}_b$ are the sample approval rates, $\hat{p} = \frac{n_a \hat{p}_a + n_b \hat{p}_b}{n_a + n_b}$ is the pooled proportion, and n_a, n_b are sample sizes.

For the loan example with $n_a = n_b = 100$, $\hat{p}_a = 0.70$, $\hat{p}_b = 0.40$:

$$\hat{p} = \frac{100(0.70) + 100(0.40)}{200} = 0.55$$
$$z = \frac{0.70 - 0.40}{\sqrt{0.55(0.45)(0.02)}} = \frac{0.30}{0.0704} = 4.26$$

With $z = 4.26$ (far exceeding critical value $z_{0.05/2} = 1.96$), we reject H_0 and conclude the demographic parity violation is statistically significant at $p < 0.001$. Similar tests can be constructed for equal opportunity and equalized odds by restricting to subpopulations where $Y = 1$ or $Y = 0$ respectively. Statistical significance does not imply practical significance; even statistically significant disparities may be acceptable if the magnitude is small. Conversely, large disparities in small samples may not reach statistical significance but still warrant intervention.

The quantitative framework developed here transforms fairness from an abstract principle into measurable engineering constraints. By establishing metrics, thresholds, and statistical tests, practitioners can systematically evaluate fairness throughout the ML lifecycle and make data-driven decisions about when intervention is required.



Napkin Math 16.2: Auditing a confusion matrix

Problem: A fraud detection model operates on two groups. What do demographic parity and equal opportunity reveal about the deployment?

Variables:

- **Group A (majority)**: TP $= 450$, FP $= 50$, FN $= 30$, TN $= 470$ ($n_{\text{records}} = 1000$).
- **Group B (minority)**: TP $= 180$, FP $= 70$, FN $= 120$, TN $= 630$ ($n_{\text{records}} = 1000$).

Math:

1. **Demographic parity (positive prediction rate)**: $\Pr(\hat{Y} = 1)$.
   - Group A: $(450 + 50)/1000 = 0.50$.
   - Group B: $(180 + 70)/1000 = 0.25$.
   - **Gap**: 0.25. (Violates four-fifths rule: $0.25/0.50 = 0.5 < 0.8$).
2. **Equal opportunity (TPR)**: $\text{TP}/(\text{TP} + \text{FN})$.
   - Group A: $450/(450 + 30) = 0.938$.
   - Group B: $180/(180 + 120) = 0.60$.
   - **Gap**: 0.338. (Severe violation).

Systems insight: Fixing TPR requires lowering the threshold for Group B to catch more fraud (reducing FNs). However, this will likely increase FPs (false alarms) for Group B, worsening predictive parity. This is a concrete demonstration of the impossibility theorem: one fairness constraint cannot be changed in isolation.

Fairness considerations extend beyond algorithmic outcomes to encompass the computational resources and infrastructure required to deploy responsible AI systems. These broader equity implications, including environmental justice concerns, arise when energy-intensive AI infrastructure is concentrated in already disadvantaged communities[8].

The computational intensity of responsible AI techniques creates a form of digital divide where access to fair, transparent, and accountable AI systems becomes contingent on economic resources. Implementing fairness constraints, differential privacy mechanisms, and comprehensive explainability tools can increase training, serving, storage, or review costs compared to unconstrained models. Compute-heavy safeguards such as continuous fairness monitoring, differentially private stochastic gradient descent, and on-demand explanations are easier for well-resourced organizations to deploy comprehensively, while resource-constrained deployments may sacrifice safeguards for efficiency. The result can be a two-tiered system where responsible AI is easier to provide to well-resourced

[8] **Data Center Environmental Justice**: A significant fraction of major U.S. cloud computing facilities sit within 16 km of low-income communities, which bear increased air pollution from backup diesel generators and heat from cooling systems. For ML fleet operators, this creates a governance constraint: data center placement decisions that optimize for power cost and latency simultaneously externalize environmental costs onto communities least able to access the AI services those data centers enable.

users and applications, potentially exacerbating existing inequalities rather than addressing them. These resource constraints create democratization challenges, while the broader implications create digital divide and access barriers affecting underserved communities.

These considerations point to the chapter's central thesis: every responsible AI property is a system-level property, not a model attribute. Fairness arises from the interaction of data engineering practices, modeling choices, evaluation procedures, and decision policies; it cannot be isolated to a single model component or resolved through post hoc adjustments alone. The same holds for privacy, explainability, robustness, and accountability examined below: each emerges from the whole pipeline rather than residing in the weights. Responsible machine learning design therefore treats fairness as a foundational constraint, one that informs architectural choices, workflows, and governance mechanisms throughout the entire lifecycle of the system. That system-level view translates each principle into concrete engineering requirements across the ML lifecycle: fairness demands group-level performance metrics and different decision thresholds across populations; explainability requires runtime compute budgets whose cost depends on whether the method uses gradients, perturbation sampling, Shapley-value approximation, or exact subset enumeration; privacy encompasses data governance, consent mechanisms, and lifecycle-aware retention policies; and accountability requires traceability infrastructure including model registries, audit logs, and human override mechanisms.

These principles interact and create tensions throughout system development. Privacy-preserving techniques may reduce explainability; fairness constraints may conflict with personalization; robust monitoring increases computational costs. As table 16.1 demonstrates, each principle manifests across data collection, training, evaluation, deployment, and monitoring phases, reinforcing that responsible AI is not a postdeployment consideration but an architectural commitment. However, the feasibility of implementing these principles depends critically on deployment context: cloud, edge, mobile, and TinyML environments each impose different constraints that shape which responsible AI features are practically achievable.

16.2.4 Privacy and data governance

Privacy and data governance present complex challenges that extend beyond threat-model perspectives, while creating fundamental tensions with the fairness and transparency principles examined above. Security-focused privacy prevents unauthorized access. Responsible privacy decides whether the system should collect a given data field at all and, if it must, how exposure is minimized throughout the lifecycle. This broader perspective creates inherent tensions: fairness monitoring requires collecting and analyzing sensitive demographic data, explainability methods may reveal information about training examples, and comprehensive transparency can conflict with individual privacy rights. Responsible AI systems must navigate these competing requirements through careful design choices that balance protection, accountability, and utility.

Machine learning systems often rely on extensive collections of personal data to support model training and allow personalized functionality. This reliance introduces significant responsibilities related to user privacy, data protection, and ethical data stewardship. The quality and governance of this data directly impacts the ability to implement responsible AI principles. Responsible AI design treats privacy not as an ancillary feature, but as a core constraint that must inform decisions across the entire system lifecycle.

One of the core challenges in supporting privacy is the inherent tension between data utility and individual protection. Rich, high-resolution datasets can enhance model accuracy and adaptability but also heighten the risk of exposing sensitive information, particularly when datasets are aggregated or linked with external sources. For example, large language models trained on broad text corpora have been shown to memorize[9] specific strings that can later be retrieved through model queries or adversarial prompting (Carlini et al. 2021).

The privacy challenges extend beyond obvious sensitive data to seemingly innocuous information. Wearable devices that track physiological and behavioral signals, including heart rate, movement, or location, may individually seem benign but can jointly reveal detailed user profiles. These risks are further exacerbated when users have limited visibility or control over how their data is processed, retained, or transmitted.

[9] **Model Memorization**: Carlini et al. (2021) demonstrated that GPT-2 could reproduce verbatim training strings, including contact information, through carefully crafted prompts. The systems implication is narrower but still serious: models that expose generative outputs to user-facing queries can create a privacy attack surface where the serving layer itself becomes a data exfiltration vector, requiring output filtering, rate limiting, and data-minimization controls as defense-in-depth measures.

Addressing these challenges requires understanding privacy as a system principle that entails robust data governance. This includes defining what data is collected, under what conditions, and with what degree of consent and transparency. Foundational data engineering practices, including data validation, schema management, versioning, and lineage tracking, provide the technical infrastructure for implementing these governance requirements. Responsible governance requires attention to labeling practices, access controls, logging infrastructure, and compliance with jurisdictional requirements. These mechanisms serve to constrain how data flows through a system and to document accountability for its use.

Figure 16.3 outlines key privacy checkpoints in the early stages of a data pipeline, highlighting where safeguards such as differential privacy, federated learning, and secure aggregation reduce attacker visibility into raw personal data. Actual implementations often involve more nuanced trade-offs and context-sensitive decisions, including separate consent and governance controls, but this diagram provides a scaffold for identifying where privacy risks arise and how they can be mitigated through responsible design choices.

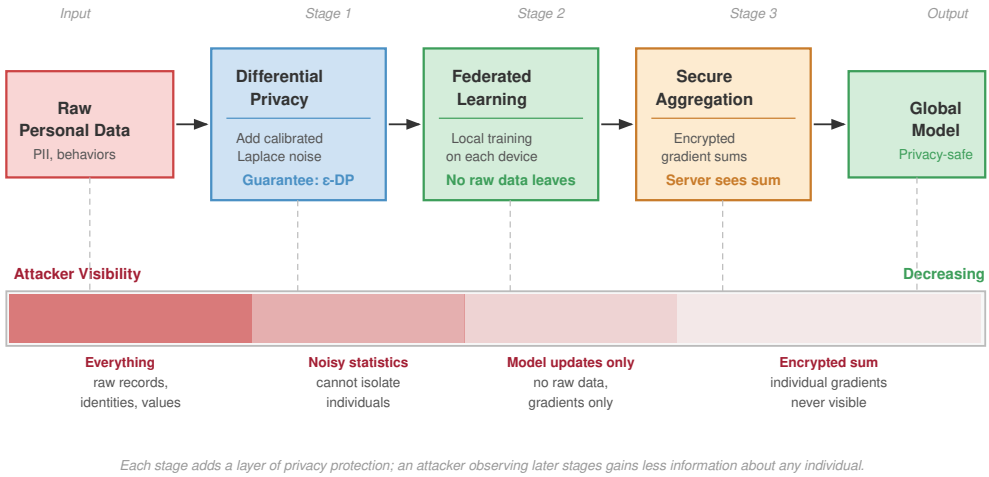


Figure 16.3: Privacy-Preserving Machine Learning Pipeline: A multi-stage architecture for training a global model while minimizing data exposure. The pipeline applies sequential safeguards—differential privacy, federated learning, and secure aggregation—that progressively reduce attacker visibility into raw personal data.

The consequences of weak data governance are well documented. Systems trained on poorly understood or biased datasets may perpetuate structural inequities or expose sensitive attributes unintentionally. In the COMPAS example introduced earlier, the lack of transparency surrounding data provenance and usage precluded effective evaluation or redress. In clinical applications, datasets frequently reflect artifacts such as missing values or demographic skew that compromise both performance and privacy. Without clear standards for data quality and documentation, such vulnerabilities become systemic.

Privacy is not solely the concern of isolated algorithms or data processors; it must be addressed as a structural property of the system. Decisions about consent collection, data retention, model design, and auditability all contribute to the privacy posture of a machine learning pipeline. This includes the need to anticipate risks not only during training, but also during inference and ongoing operation. Threats such as membership inference attacks[10] underscore the importance of embedding privacy safeguards into both model architecture and interface behavior.

10 **Membership Inference Attacks**: First demonstrated by Shokri et al. (2017), these attacks determine whether a specific individual's data was used to train a model by exploiting the confidence gap between seen and unseen inputs. The systems implication: any ML model exposed via an API can become a potential privacy oracle, and determining that someone's medical record was in a disease prediction model's training set can reveal sensitive health information. Defenses include differential privacy, regularization, confidence calibration, and interface controls such as rate limiting.

Legal frameworks reflect this understanding. Regulations such as the GDPR, CCPA[11], and Japan's Act on the Protection of Personal Information (APPI) impose specific obligations regarding data minimization, purpose limitation, user consent, and deletion or erasure rights (European Parliament and Council of the European Union 2016; California Legislature 2023; Personal Information Protection Commission, Japan 2023). These requirements translate ethical expectations into enforceable design constraints, reinforcing the need to treat privacy as a core principle in system development.

Privacy is the second instance of the system-level thesis established for fairness in section 16.2.3: it is a commitment that spans consent collection, retention policy, model design, serving, and auditability, not a property of any single stage. It requires coordination across technical and organizational domains to ensure that data usage aligns with user expectations, legal mandates, and societal norms. Rather than viewing privacy as a constraint to be balanced against functionality, responsible system design integrates privacy from the outset by informing architecture, shaping interfaces, and constraining how models are built, updated, and deployed.

Privacy preservation prevents unauthorized data exposure, but responsible systems must also ensure predictable behavior even when privacy mechanisms cannot prevent all risks. A model may satisfy every privacy constraint while still failing catastrophically when encountering unexpected inputs or adversarial conditions. Safety and robustness address this complementary concern: *how systems fail*, not just *how data is protected*.

11 **CCPA (California Consumer Privacy Act)**: CCPA grants California residents rights over covered personal information, including a deletion right subject to statutory exceptions (California Legislature 2023). For ML serving infrastructure, deletion requests create an architectural constraint that must be planned from the data pipeline forward, because raw-record deletion does not automatically remove a record's influence from trained model parameters.

16.2.5 Safety and robustness

Safety and robustness, introduced in Chapter 14 as technical properties addressing hardware faults, adversarial attacks, and distribution shifts, also serve as responsible AI principles that extend beyond threat mitigation. Technical robustness ensures systems survive adversarial conditions; responsible robustness ensures systems behave in ways aligned with human expectations and values, even when technically functional. A model may be robust to bit flips and adversarial perturbations yet still exhibit behavior that is unsafe for deployment if it fails unpredictably in edge cases or optimizes objectives misaligned with user welfare.

Safety in machine learning refers to the assurance that models behave predictably under normal conditions and fail in controlled, noncatastrophic ways under stress or uncertainty. Closely related, robustness concerns a model's ability to maintain stable and consistent performance in the presence of variation, whether in inputs, environments, or system configurations. Together, these properties are foundational for responsible deployment in safety critical domains, where machine learning outputs directly affect physical or high stakes decisions.

Ensuring safety and robustness in practice requires anticipating the full range of conditions a system may encounter and designing for behavior that remains reliable beyond the training distribution. This includes not only managing the variability of inputs but also addressing how models respond to unexpected correlations, rare events, and deliberate attempts to induce failure. For example, the NTSB investigation of Uber's 2018 Tempe crash shows how automated-driving failures can combine perception, prediction, safety-driver, and organizational-control problems rather than reduce to a single model error (National Transportation Safety Board 2019).

One illustrative failure mode arises from adversarial inputs[12]: carefully constructed perturbations that appear benign to humans but cause a model to output incorrect or harmful predictions (Szegedy et al. 2013). Such vulnerabilities are not limited to image classification; they have been observed across modalities including audio, text, and structured data, and they reveal the brittleness of learned representations in high-dimensional spaces. Addressing these vulnerabilities requires specialized approaches including adversarial defenses and robustness techniques. These behaviors highlight that robustness must be considered not only during training but as a global property of how systems interact with real-world complexity.

A related challenge is distribution shift[13]: the inevitable mismatch between training data and conditions encountered in deployment. Whether due to seasonality, demographic changes, sensor degradation, or environmental variability, such shifts can degrade model reliability even in the absence of adversarial manipulation. Addressing distribution shift challenges requires systematic approaches to detecting and adapting to changing conditions.

Responsible machine learning design treats robustness as a systemic requirement. Addressing it requires more than improving individual model performance. It involves designing systems that

12 **Adversarial Inputs**: First demonstrated by Szegedy et al. (2013), these are imperceptible input perturbations that cause confident misclassification. A perturbation of magnitude 0.005 (in pixel space) can flip a classifier's output with >99 percent confidence, revealing that neural networks' decision boundaries are far more fragile than their test-set accuracy suggests. For safety-critical ML systems, this means test accuracy provides no guarantee of deployment robustness, requiring adversarial testing as a separate validation stage.

13 **Distribution Shift**: The mismatch between training and deployment data distributions, manifesting as covariate shift (input distribution changes), label shift (class proportions change), or concept drift (the input-output relationship evolves over time). The systems consequence is silent degradation: unlike software bugs that crash, distribution shift erodes accuracy over weeks or months without triggering errors, requiring continuous monitoring infrastructure with automated retraining triggers and model versioning to detect and respond.

anticipate uncertainty, surface their limitations, and support fallback behavior when predictive confidence is low. This includes practices such as setting confidence thresholds, supporting abstention from decision-making, and integrating human oversight into operational workflows. These mechanisms are important for building systems that degrade gracefully rather than failing silently or unpredictably.

These individual-model considerations extend to broader system requirements. Safety and robustness also impose requirements at the architectural and organizational level. Decisions about how models are monitored, how failures are detected, and how updates are governed all influence whether a system can respond effectively to changing conditions. Responsible design demands that robustness be treated not as a property of isolated models but as a constraint that shapes the overall behavior of machine learning systems. This system-level perspective on safety and robustness leads to questions of accountability and governance.

16.2.6 Accountability and governance

Accountability in machine learning refers to the capacity to identify, attribute, and address the consequences of automated decisions. It extends beyond diagnosing failures to ensuring that responsibility for system behavior is explicitly assigned, that harms can be remedied, and that ethical standards are maintained through oversight and institutional processes. Without such mechanisms, even well intentioned systems can generate significant harm without recourse, undermining public trust and eroding legitimacy.

Unlike traditional software systems, where responsibility often lies with an identifiable developer or operator, accountability in machine learning is distributed. Model outputs are shaped by upstream data collection, training objectives, pipeline design, interface behavior, and postdeployment feedback. These interconnected components often involve multiple actors across technical, legal, and organizational domains. For example, if a hiring platform produces biased outcomes, accountability may rest not only with the model developer but also with data providers, interface designers, and deploying institutions. Responsible system design requires that these relationships be explicitly mapped and governed.

Inadequate governance can prevent institutions from recognizing or correcting harmful model behavior. The failure of Google Flu Trends to anticipate distribution shift and feedback loops illustrates the consequences of unmodeled changes in user search behavior, media-driven query spikes, and insufficient recalibration against CDC surveillance ground truth (Lazer et al. 2014). Persistent overestimation went uncorrected for years, contributing to the model's eventual discontinuation.

Legal frameworks also reflect the necessity of accountable design. Regulations such as the Illinois Artificial Intelligence Video Interview Act (Illinois General Assembly 2020) and the EU AI Act impose requirements for transparency, consent, documentation, and oversight in high risk applications (European Parliament and Council of the European Union 2024). These policies embed accountability not only in the outcomes a system produces, but in the operational procedures and documentation that support its use. Internal organizational changes, including the introduction of fairness audits and the imposition of usage restrictions in targeted advertising systems, demonstrate how regulatory pressure can catalyze structural reforms in governance.

Designing for accountability entails supporting traceability at every stage of the system lifecycle. This includes documenting data provenance, recording model versioning, enabling human overrides, and retaining sufficient logs for retrospective analysis. Tools such as **model cards**[14] (Mitchell et al. 2019) and **datasheets for datasets**[15] (Gebru et al. 2021) exemplify practices that make system behavior interpretable and reviewable. Mitchell and colleagues proposed model cards as short documents accompanying trained ML models that provide benchmarked evaluation across cultural, demographic, and phenotypic groups relevant to intended applications. Similarly, Gebru and colleagues proposed that every dataset be accompanied by a datasheet documenting its motivation, composition, collection process, and recommended uses, analogous to how electronic components include specification sheets. However, accountability is not reducible to documentation alone; it also requires mechanisms for feedback, contestation, and redress.

Within organizations, governance structures help formalize this responsibility. Ethics review processes, cross-functional audits, and model risk committees provide forums for anticipating downstream impact and responding to new concerns. These structures must be supported by

14 **Model Cards**: Proposed by Mitchell et al. (2019) as standardized documentation accompanying trained ML models. Each card benchmarks performance across demographic groups, documents intended use cases, and discloses known limitations. For production ML systems, model cards serve as the traceability layer linking a deployed binary to its training provenance, evaluation results, and known failure modes – the minimum metadata required for postdeployment auditing and regulatory compliance.

15 **Datasheets for Datasets**: Proposed by Gebru et al. (2021), modeled after electronics component datasheets that specify operating conditions and tolerances. Each datasheet documents a dataset's motivation, composition, collection process, and recommended uses. For ML pipelines, datasheets function as the data equivalent of hardware specs: they define the valid operating envelope of a model's training distribution, enabling engineers to predict where deployment-time distribution shift will cause failures.

infrastructure that allows users to contest decisions and developers to respond with corrections. For instance, systems that allow explanations or user-initiated reviews help bridge the gap between model logic and user experience, especially in domains where the impact of error is significant.

Architectural decisions also play a role. Interfaces can be designed to surface uncertainty, allow escalation, or suspend automated actions when appropriate. Logging and monitoring pipelines must be configured to detect signs of ethical drift, such as performance degradation across subpopulations or unanticipated feedback loops. In distributed systems, where uniform observability is difficult to maintain, accountability must be embedded through architectural safeguards such as secure protocols, update constraints, or trusted components.

Governance does not imply centralized control. Instead, it involves distributing responsibility in ways that are transparent, actionable, and sustainable. Technical teams, legal experts, end users, and institutional leaders must all have access to the tools and information necessary to evaluate system behavior and intervene when necessary. As machine learning systems become more complex and embedded in important infrastructure, accountability must scale accordingly by becoming a foundational consideration in both architecture and process, not a reactive layer added after deployment.

Despite these governance mechanisms, meaningful accountability faces a challenge: distinguishing between decisions based on legitimate factors vs. spurious correlations that may perpetuate historical biases. This challenge requires careful attention to data quality, feature selection, and ongoing monitoring to ensure that automated decisions reflect fair and justified reasoning rather than problematic patterns from biased historical data.

The principles and techniques examined above provide the conceptual and technical foundation for responsible AI, but their practical implementation depends critically on deployment architecture. Cloud systems can support complex Shapley-value explanations and real-time fairness monitoring, but TinyML devices must rely on static interpretability and compile-time privacy guarantees. Edge deployments enable local privacy preservation but limit global fairness assessment. These architectural constraints are not mere implementation details; they fundamentally shape which responsible AI protections are accessible to different users and applications.

Example 16.4: Choosing a fairness metric

Scenario: A hiring recommendation model must choose the critical fairness metric before launch.

Question: Which metric best matches a setting where qualified candidates should have comparable opportunity across groups?

Answer: Equalized odds is usually most appropriate for hiring because it requires equal true positive rates and false positive rates. Demographic parity can force rejection of qualified candidates when base rates differ, while calibration only ensures that a score such as 0.8 has the same meaning across groups.

The example illustrates why metric selection is a governance decision before it is a computation: the chosen fairness definition determines which error trade-off the system is allowed to make. Selecting the mathematically appropriate fairness metric is the first step, but calculating these metrics requires access to demographic data and significant computational overhead. Enforcing these mathematical guarantees grows far more complex when moving from a centralized cloud environment to constrained, distributed edge deployments where privacy and bandwidth dictate the architecture.

Checkpoint 16.1: Principles as engineering constraints

This section reframed fairness, explainability, transparency, accountability, and value alignment as control-plane invariants rather than post-hoc review items, mapped each across the ML lifecycle, and quantified the fairness tax and the responsible AI overhead.

From principle to mechanism

- ☐ **Transparency vs. fairness**: Explain why fully transparent decision logic can still violate fairness when it disadvantages one demographic group, and identify the principle this system violates.
- ☐ **Lifecycle coverage**: Apply the responsible AI lifecycle to a fairness requirement by identifying which phases must act and explaining why post-hoc threshold adjustment alone is insufficient.

Judging the trade-off

- ☐ **Safeguard budget**: Reason from a limited compute budget and the provided overhead figures to select an affordable safeguard among differential privacy, real-time fairness monitoring, and on-demand explanation.
- ☐ **Distributed accountability**: Argue why accountability for a biased hiring model is distributed across data providers, interface designers, and the deploying institution rather than resting solely with the model developer.

16.3 Responsible AI Across Deployment Environments

Auditing a model for bias is straightforward with a massive centralized database and unlimited cloud GPUs. Auditing a federated learning model running on a million individual smartphones, where strict privacy laws prevent access to demographic data, is an entirely different engineering problem. The deployment environment fundamentally dictates which responsible AI techniques are mathematically and legally possible.

These architectural differences redistribute both feasible safeguards and accountable parties. Resource availability, latency constraints, user interface design, and the presence or absence of connectivity determine whether responsible AI principles can be enforced consistently across deployment contexts. The same model may support rich explanation, continuous monitoring, and centralized auditability in the cloud, yet require static validation and simplified safeguards when embedded in a device with intermittent connectivity.

The geographic and economic distribution of computational resources adds another equity layer. High-performance AI systems typically require proximity to major data centers or high-bandwidth internet connections, creating service quality disparities that map closely to existing socioeconomic inequalities. Rural communities, developing regions, and economically disadvantaged areas often experience degraded AI service quality due to network latency, limited bandwidth, and distance from computational infrastructure. US Federal Communications Commission reports have documented persistent broadband disparities, with rural areas substantially less likely than urban areas to have fixed broadband meeting the benchmark used in those reports. For responsible AI, that infrastructure gap determines whether real-time explainability, continuous fairness monitoring, and privacy-preserving computation are practical for the users and communities most affected by automated decisions.

16.3.1 System explainability

The binding constraint on explainability is the deployment environment, not the choice of technique: computational capacity, latency budget, interface design, and data access decide which explanation method, if any, can run. Cloud systems afford heavyweight posthoc methods like Shapley-value attribution (SHAP) and local surrogate explanations (LIME)[16]; mobile and edge devices can usually afford only single-pass saliency maps[17]; TinyML devices often support no runtime explanation at all, leaving development-time inspection as the only opportunity. Section 16.6 develops the technique-level mechanics and costs that this matrix presupposes; table 16.3 records the per-environment feasibility.

Two dimensions the table cannot capture make explainability distinct from the other principles. The first is audience: an end user needs a concise summary ("elevated heart rate during sleep"), while a developer, auditor, or regulator needs attribution maps, concept activations, or decision traces, so the same deployment must often surface two explanation surfaces at once. The second is timing:

16 **LIME (Local Interpretable Model-Agnostic Explanations)**: Introduced by Ribeiro et al. (2016), LIME explains individual predictions by perturbing the input, querying the black-box model many times, and fitting a weighted linear surrogate to approximate local behavior. The systems trade-off is that explanation latency scales with the number of perturbation samples and model-call cost, so production deployments often use sampling budgets, caching, asynchronous explanation workers, or model-specific methods when the architecture permits.

17 **Saliency Maps**: Gradient-based explanation that highlights which input regions most influenced a prediction by computing a backward pass – the same infrastructure used for training. Saliency methods are often cheaper than perturbation-heavy or subset-enumeration methods, making them attractive for edge and mobile deployments when the model supports gradients. The trade-off is that raw gradients can be noisy and may highlight input artifacts rather than meaningful features, so smoothing or repeated-gradient variants trade additional compute for more stable visualizations.

a drone or industrial control loop has no slack to compute an explanation during operation and can only log internal signals for later analysis, whereas asynchronous systems such as financial risk scoring or medical diagnosis can render a deeper explanation after the decision. Planning for both, within the latency, energy, and interface limits of the deployment, is what makes interpretability a design constraint rather than a postdeployment wish.

16.3.2 Fairness constraints

Fairness presents a parallel deployment problem, and its binding constraint is data visibility. A cloud platform with demographically annotated datasets can compute group-level metrics, run fairness-aware training, and audit posthoc. A federated[18] or on-device deployment cannot: no single entity observes the global demographic distribution, so a group-level fairness audit is not merely expensive but mathematically impossible without privacy-preserving aggregation, and fairness must instead be built into training or dataset curation up front. Table 16.3 records the per-environment differences; two consequences of the data-visibility limit are worth drawing out because they recur across deployments.

The first is that decision-threshold policy, not just model quality, determines realized fairness. Even a model with equal accuracy across groups can produce disparate impact under a uniform threshold when score distributions differ, so a mobile loan-approval system may systematically under-approve one group unless group-specific thresholds are reasoned about and embedded in policy in advance. The second is that personalization without global context can quietly compound disparity: locally adapted models aim for individual fairness, where similar individuals receive similar decisions under a task-relevant similarity metric (Dwork et al. 2012), but retraining on the behavior of marginalized users whose data is sparse or noisy can drift toward reinforcing existing gaps. That drift is a feedback loop. A hiring platform that favors candidates from specific institutions amplifies the inequality when retrained on its own biased outcomes, which is why mitigation must span deployment monitoring, data logging, and impact evaluation rather than a single audit. Figure 16.4 illustrates the bias amplification feedback loop, where each stage amplifies existing bias unless interrupted by the green intervention points.

The cycle in figure 16.4 reveals why post-hoc fairness audits are insufficient: without intervention at each of the four stages (data, model, predictions, and retraining), each iteration compounds the bias introduced by the previous one, turning a small initial skew into a systematic disparity. Fairness therefore moves from a metric choice to a lifecycle constraint: data visibility determines what can be measured, threshold policy determines where errors land, and feedback infrastructure determines whether disparities decay or compound.

That same deployment split creates the next responsible AI tension. Keeping sensitive data local can protect users, but the locality that supports privacy also removes the global observability that many fairness audits require.

16.3.3 Privacy architectures

Privacy inherits the same centralization trade-off, sharpened. Aggregating data in the cloud enables high-capacity modeling and monitoring but concentrates exposure to breaches and surveillance, so it must be paired with strong encryption, access control, and auditability. Keeping data local on mobile, federated, or TinyML clients reduces that central risk but removes the global observability needed to monitor or enforce compliance, forcing privacy safeguards to be compiled into the model and firmware before deployment, with no runtime channel to adjust them. Table 16.3 records where each environment lands.

The constraint the table cannot show is that privacy is judged at the serving layer, not just at training time. A model that satisfies every formal privacy definition can still leak: membership inference attacks recover whether a user's record was in the training set by reading model outputs, so defenses must extend into interface design, rate limiting, and access control. Furthermore, a technically compliant system can still violate user expectations if data collection is opaque, which makes consent surfaces and clear disclosure part of the privacy architecture rather than a UI afterthought.

Privacy protects data flows, but it does not guarantee that the system's outputs remain reliable when inputs shift, sensors degrade, or adversaries probe the interface. That gap leads from privacy

[18] **Federated Learning and Fairness**: Google introduced federated learning in 2016 for Gboard, training across many devices without centralizing raw data. The fairness complication is that no single entity observes the complete demographic distribution across participants, making group-level fairness metrics impossible to compute directly without additional protocols. Federated fairness assessment may require privacy-preserving aggregation, secure aggregation, or differential privacy, each of which changes communication, accuracy, and governance trade-offs compared to centralized training.

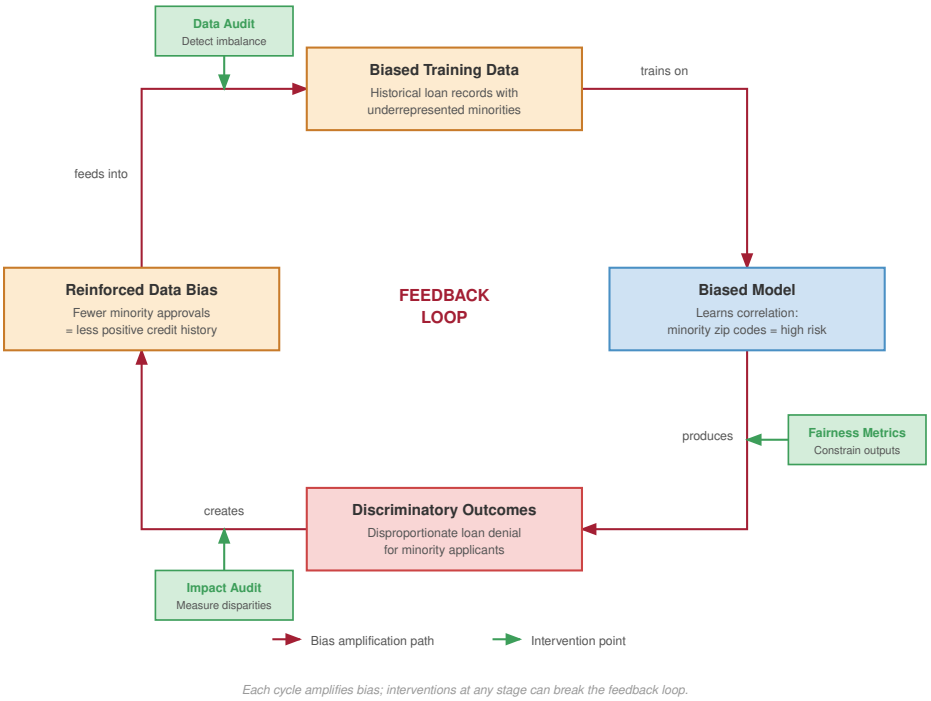


Figure 16.4: Bias Amplification Feedback Loop: When ML systems retrain on their own outputs, historical bias in training data propagates through model predictions, downstream outcomes, and reinforced data before feeding back as even more biased retraining data. Each stage edge ('trains on', 'produces', 'creates', 'feeds into') amplifies the distortion. Solid green intervention arrows show three external mitigation points (Data Audit, Fairness Metrics, and Impact Audit) that can break the cycle.

architecture to safety and robustness architecture: the system must also define how it behaves under stress.

16.3.4 Safety and robustness across deployments

The binding constraint on safety and robustness is the latency budget, because it decides which defense can run at all. A cloud service can afford uncertainty estimators, distributional-change detectors, adversarial input filtering, and API[19] rate limiting; an autonomous-navigation or control loop with a millisecond budget cannot, and must instead precompute its fallback actions in advance. Table 16.3 records the per-environment split; at the constrained extreme, TinyML systems have no runtime monitoring or update channel at all, so robustness has to be engineered statically through conservative design and predeployment testing.

The consequence the table cannot show is that fallback is itself a latency-budget problem. A delivery robot that abstains[20] when pedestrian-detection confidence is low has only solved half the problem; the human reviewer, rule-based default, or escalation queue that catches the abstained case must execute within the same temporal budget as the model it is protecting. A robust system is therefore not one that avoids all errors but one that fails visibly, controllably, and safely, which depends on concrete choices about sensing, confidence estimation, fallback routing, and recovery. Those choices also define the governance surface: the system must specify who can observe harm, who can intervene, and who remains accountable for the outcome.

16.3.5 Governance structures

Accountability is realized through concrete traceability infrastructure, and its binding constraint is how much of that infrastructure the environment can carry. A cloud platform supports model

19 **API Security for ML**: ML serving endpoints face attacks absent from traditional APIs, including model extraction through repeated targeted queries and adversarial input injection. Rate limiting, anomaly detection, access controls, and input validation can reduce exposure, but the fundamental trade-off is that making models more accessible for legitimate explainability simultaneously increases the attack surface for model theft and adversarial probing.

registries, telemetry[21] dashboards, and structured event pipelines that trace a prediction back to a specific model, input, or configuration. A TinyML device carries none of that: with no connectivity, persistent storage, or runtime configurability, accountability has to be embedded statically through cryptographic firmware signatures, fixed audit trails, and predeployment documentation, sometimes enforced at manufacturing because no postdeployment correction is possible. Table 16.3 records where each environment lands on this spectrum.

The deployment that most tests this is mobile, because responsibility is split across a local model, a remote service, and the interface design, so when something goes wrong it is often unclear which layer owns the failure. Governance there depends on accessible recourse pathways and mechanisms for surfacing, explaining, and contesting decisions at the user level, embedded into both the interface and the surrounding service architecture rather than bolted on as a policy overlay. Across every environment, sustaining accountability means planning for failure as deliberately as for success: defining how anomalies are detected, how roles are assigned, how records are maintained, and how remediation occurs, all traceable in logs and enforceable through interfaces. Governance also has to account for the environmental and distributional impacts of infrastructure choices, because organizations deploying AI systems bear responsibility not only for algorithmic outcomes but for the broader effects of resource usage on environmental justice and equitable access.

16.3.6 Design trade-offs

The governance challenges across deployment contexts reveal a general pattern: responsible AI trade-offs are architectural trade-offs. Machine learning systems do not operate in idealized silos; they must navigate competing objectives under finite resources, strict latency requirements, evolving user behavior, and regulatory complexity.

Cloud-based systems often support extensive monitoring, fairness audits, interpretability services, and privacy-preserving tools due to ample computational and storage resources. However, these benefits typically come with centralized data handling, which introduces risks related to surveillance, data breaches, and complex governance. In contrast, on-device systems such as mobile applications, edge platforms, or TinyML deployments provide stronger data locality and user control, but limit postdeployment visibility, fairness instrumentation, and model adaptation.

Tensions between goals often become apparent at the architectural level. For example, systems with real-time response requirements, such as wearable gesture recognition or autonomous braking, cannot afford to compute detailed interpretability explanations during inference. Designers must choose whether to precompute simplified outputs, defer explanation to asynchronous analysis, or omit interpretability altogether in runtime settings.

Conflicts also emerge between personalization and fairness. Systems that adapt to individuals based on local usage data often lack the global context necessary to assess disparities across population subgroups. Ensuring that personalized predictions do not result in systematic exclusion requires careful architectural design, balancing user-level adaptation with mechanisms for group-level equity and auditability.

Privacy and robustness objectives can also conflict. Robust systems often benefit from logging rare events or user outliers to improve reliability. However, recording such data may conflict with privacy goals or violate legal constraints on data minimization. In settings where sensitive behavior must remain local or encrypted, robustness must be designed into the model architecture and training procedure in advance, since post hoc refinement may not be feasible.

The computational demands of responsible AI create tensions that extend beyond technical optimization to questions of environmental justice and equitable access. Energy-efficient deployment often requires simplified models with reduced fairness monitoring capabilities, creating a trade-off between environmental sustainability and ethical safeguards. For example, implementing differential privacy in federated learning can increase per-device energy consumption by 25–40 percent, potentially making such privacy protections prohibitive for battery-constrained devices[22].

Together, these deployment examples reveal a broader systems-level challenge. Responsible AI principles cannot be considered in isolation. They interact, and optimizing for one may constrain another. The appropriate balance depends on deployment architecture, stakeholder priorities, domain-specific risks, the consequences of error, and the environmental and distributional impacts of computational resource requirements. Responsible machine learning design therefore depends

20 **Abstention**: The practice of refusing predictions when confidence falls below a threshold. Abstention can reduce high-risk errors, but only by shifting some cases to fallback infrastructure such as human reviewers, rule-based defaults, or escalation queues. Autonomous vehicles hand control to human drivers; medical AI routes ambiguous cases to specialist review – both requiring the routing logic to execute inside the system's latency and safety budget.

21 **Telemetry in ML Systems**: Real-time capture of prediction latencies, accuracy, and resource utilization across deployed models. Alerts typically trigger when accuracy drops more than 5 percent or latency exceeds 200 ms. The accountability challenge emerges at fleet scale: a system serving hundreds of models to diverse users generates millions of telemetry events daily, and tracing a specific harmful prediction back to a root cause (data drift, model regression, or threshold misconfiguration) requires end-to-end lineage infrastructure that most organizations lack.

22 **Energy-Privacy Trade-off**: Privacy-preserving techniques like differential privacy and secure multi-party computation can increase computation, communication, and battery use. In federated learning on mobile devices, the equity implication is direct: users with older devices, limited data plans, or limited battery life may be excluded from privacy-protected AI services, creating a system where privacy protection becomes contingent on hardware resources.

on making those constraints explicit before choosing methods, so that the resulting system can be evaluated against the deployment context it must actually inhabit.

Table 16.3 synthesizes deployment trade-offs by comparing how responsible AI principles manifest across Cloud ML, Edge ML, Mobile ML, and TinyML systems. Each setting imposes different constraints on explainability, fairness, privacy, safety, and accountability, based on factors such as compute capacity, connectivity, data access, and governance feasibility.

No deployment context dominates across all principles; each makes different compromises. As table 16.3 reveals, cloud systems support complex explainability methods (SHAP, LIME) and centralized fairness monitoring but introduce privacy risks through data aggregation. Edge and mobile deployments offer stronger data locality but limit postdeployment observability and global fairness assessment. TinyML systems face the most severe constraints, requiring static validation and compile-time privacy guarantees with no opportunity for runtime adjustment. These constraints are not merely technical limitations but shape which responsible AI features are accessible to different users and applications, creating equity implications where only well-resourced deployments can afford comprehensive safeguards. Understanding these deployment constraints provides necessary context for the technical methods that operationalize responsible AI principles in practice.

Table 16.3: Deployment Trade-Offs: Responsible AI principles manifest differently across deployment contexts due to varying constraints on compute, connectivity, and governance; cloud deployments support complex explainability methods, while TinyML severely limits them. Prioritizing certain principles like explainability, fairness, privacy, safety, and accountability requires careful consideration of these constraints when designing machine learning systems for cloud, edge, mobile, and TinyML environments.

Principle	Cloud ML	Edge ML	Mobile ML	TinyML
Explainability	Supports complex models and methods like SHAP and sampling approaches	Needs lightweight, low-latency methods like saliency maps	Requires interpretable outputs for users, often defers deeper analysis to the cloud	Severely limited due to constrained hardware; mostly static or compile-time only
Fairness	Large datasets allow bias detection and mitigation	Localized biases harder to detect but allows on-device adjustments	High personalization complicates group-level fairness tracking	Minimal data limits bias analysis and mitigation
Privacy	Centralized data at risk of breaches but can use strong encryption and differential privacy methods	Sensitive personal data on-device requires on-device protections	Tight coupling to user identity requires consent-aware design and local processing	Distributed data reduces centralized risks but poses challenges for anonymization
Safety	Vulnerable to hacking and large-scale attacks	Real-world interactions make reliability important	Operates under user supervision, but still requires graceful failure	Needs static safety envelopes, watchdogs, and fail-safe behavior
Accountability	Corporate policies and audits allow traceability and oversight	Fragmented supply chains complicate accountability	Requires clear user-facing disclosures and feedback paths	Traceability required across long, complex hardware chains
Governance	External oversight and regulations like GDPR or CCPA are feasible	Requires self-governance by developers and integrators	Balances platform policy with app developer choices	Relies on built-in protocols and cryptographic assurances

The deployment analysis establishes a critical insight: the feasibility of a responsible AI technique depends on architectural constraints. A TinyML device cannot run SHAP explanations; an edge system cannot implement real-time fairness monitoring; a mobile application cannot store the audit logs required for comprehensive accountability. Responsible machine learning therefore requires technical methods selected for the constraints just mapped: detecting bias, preserving privacy, ensuring robustness, and providing interpretability only matter operationally when they fit the data quality, compute budget, deployment requirements, and serving stack that determine whether a technique can run.

The need for these methods begins with how models learn. Training data contains historical biases and unfair associations, and a hiring algorithm trained on biased historical data can reproduce discriminatory patterns by associating demographic characteristics with success. The model learns correlations rather than causation, so statistical patterns that reflect unfair social structures can be optimized as if they were meaningful relationships. Traditional machine learning that optimizes

only for accuracy therefore creates tension with fairness goals. Effective solutions must integrate the relevant constraint into data collection, training, serving, or monitoring rather than attach it as a secondary correction after training.

The operational techniques that follow fall into three complementary categories, each with distinct trade-offs in accuracy, computational cost, and implementation complexity. Detection methods identify problematic behavior early enough to intervene, providing warning systems for bias, drift, and performance issues. Mitigation techniques prevent or reduce harmful outcomes through algorithmic interventions and robustness enhancements. Validation approaches make system behavior legible to the developers, auditors, regulators, and affected users who evaluate automated decisions.

16.3.6.1 Computational overhead of responsible AI techniques

The next sections unpack the technical methods in detail, but their systems cost is visible before the implementation details. Table 16.4 shows the performance impact of responsible AI techniques: detection, mitigation, validation, privacy, and explanation techniques each impose different costs on training, inference, memory, and accuracy. Reading the table first lets engineers evaluate responsible AI methods as deployable system choices rather than as abstract ethical labels.

Table 16.4: Performance Impact of Responsible AI Techniques: The table gives planning ranges for capacity exercises rather than universal benchmark results. Differential privacy and fairness constraints mainly affect training and validation budgets; explainability cost depends sharply on the variant, with approximate TreeSHAP and kernel-based methods requiring repeated model work and exact SHAP growing combinatorially with feature count. These sizing ranges help engineers reason about production constraints before choosing a responsible-AI implementation.

Technique	Accuracy Loss	Training Overhead	Inference Cost	Memory Overhead
Differential Privacy (DP-SGD)	2–5%	15–30%	Minimal	10–20%
Fairness-Aware Training (Reweighting/Constraints)	1–3%	5–15%	Minimal	5–10%
Approximate SHAP (Tree/Kernel)	N/A	N/A	50–200%	20–100%
Exact SHAP	N/A	N/A	50–1,000×	50–200%
Adversarial Training	2–5%	100–300%	Minimal	50–100%
Federated Learning	5–15%	200–500%	Minimal	100–300%

These overhead ranges should be read as scenario assumptions, not as claims that any cited paper establishes a single universal tax.[23] Actual overhead varies significantly based on model architecture, dataset size, attack strength, explanation method, hardware, communication pattern, and implementation quality. The point is architectural: responsible-AI controls consume real training, serving, memory, or review capacity, so they must be budgeted before launch. These overhead figures are also where the equity argument of section 16.2.1.1 becomes concrete: an organization that cannot afford the training, inference, and memory budgets in the table cannot afford the protections either, so the right to a fair or private model tracks the compute budget behind it.

Because these computational costs and architectural constraints heavily influence system design, engineering teams must carefully select the right tools for their specific deployment reality. Regardless of the environment, the first step in taking corrective action is knowing that a problem exists, making detection methods the foundation for all other responsible AI interventions.

[23] **Measurement Context**: The sizing exercise assumes common production reference points such as GPU training, GPU or CPU inference, image/text/tabular models, and benchmark datasets. These figures are not canonical constants; they are order-of-magnitude planning anchors for deciding whether a safeguard belongs inline, asynchronous, offline, or in the training pipeline.

16.4 Bias Detection and Fairness Monitoring

A credit scoring model deployed nationally begins rejecting qualified applicants from a specific ZIP code at twice the normal rate. Without bias detection infrastructure, the disparity persists for weeks

or months before anyone notices. Bias detection transforms theoretical fairness definitions into live, operational telemetry. Just as a Site Reliability Engineer monitors latency dashboards, a Responsible AI engineer monitors demographic parity dashboards, using slice-based analysis to identify when the model begins failing specific subpopulations.

16.4.1 Bias detection and mitigation

The fairness definitions examined in section 16.2.3 provide mathematical precision for what fairness means: demographic parity, equalized odds, and equality of opportunity are mathematically defined. Practitioners, however, face a practical challenge: computing these metrics on production systems processing thousands of predictions per second. Manual calculation using the formulas above is infeasible at scale. This gap between definition and deployment motivates specialized tooling.

Operationalizing fairness in deployed systems requires more than principled objectives or theoretical metrics; it demands system-aware methods that detect, measure, and mitigate bias across the machine learning lifecycle. Practical bias detection can be implemented using tools like Fairlearn[24] (Bird et al. 2020).

Listing 16.1 enables offline or CI-stage fairness audits across demographic groups, revealing concerning disparities where loan approval rates vary dramatically by ethnicity: from 94 percent for Asian applicants to 68 percent for Black applicants. Building on the system-level constraints discussed earlier, fairness must be treated as an architectural consideration that intersects with data engineering, model training, inference design, monitoring infrastructure, and policy governance. While fairness metrics such as demographic parity, equalized odds, and equality of opportunity formalize different normative goals, their realization depends on the architecture's ability to measure subgroup performance, support adaptive decision boundaries, and store or surface group-specific metadata during runtime; deployment-time monitoring is handled later in this section.

24 **Fairlearn**: Microsoft's open-source toolkit (2020) for computing fairness metrics and applying mitigation algorithms to scikit-learn compatible models (Bird et al. 2020). The systems integration pattern is that fairness measurement and mitigation wrap the model pipeline instead of living in a separate spreadsheet audit. The practical significance is that fairness monitoring becomes a CI/CD pipeline stage rather than an ad hoc review, enabling automated regression detection when model updates degrade subgroup performance.

Listing 16.1: Bias Detection with Fairlearn: Evaluating a loan-approval model across demographic groups to surface approval-rate and false-positive disparities indicative of discriminatory patterns.

```
from fairlearn.metrics import MetricFrame, selection_rate
from sklearn.metrics import accuracy_score, precision_score

# Loan approval model evaluation across demographic groups
mf = MetricFrame(
    metrics={
        "approval_rate": selection_rate,
        "accuracy": accuracy_score,
        "precision": precision_score,
        "false_positive_rate": lambda y_true, y_pred: (
            (y_pred == 1) & (y_true == 0)
        ).sum()
        / (y_true == 0).sum(),
    },
    y_true=loan_approvals_actual,
    y_pred=loan_approvals_predicted,
    sensitive_features=applicant_demographics["ethnicity"],
)

# Display performance disparities across ethnic groups
print("Loan Approval Performance by Ethnic Group:")
print(mf.by_group)
# Output shows: Asian: 94% approval, White: 91% approval,
# Hispanic: 73% approval, Black: 68% approval
```

Practical implementation is often shaped by limitations in data access and system instrumentation. In many real-world environments, especially in mobile, federated, or embedded systems, sensitive attributes such as gender, age, or race may not be available at inference time, making it difficult to track or audit model performance across demographic groups. Data collection and labeling strategies are essential for fairness assessment throughout the model lifecycle. In such contexts, fairness interventions must occur upstream during data curation or training, as postdeployment

recalibration may not be feasible. Even when data is available, continuous retraining pipelines that incorporate user feedback can reinforce existing disparities unless explicitly monitored for fairness degradation. For example, an on-device recommendation model that adapts to user behavior may amplify prior biases if it lacks the infrastructure to detect demographic imbalances in user interactions or outputs.

Figure 16.5 illustrates how fairness constraints can introduce tension with deployment choices. In a binary loan approval system, two subgroups, Subgroup A (represented in blue) and Subgroup B (represented in red), require different decision thresholds to achieve equal true positive rates. Using a single threshold across groups leads to disparate outcomes, potentially disadvantaging Subgroup B. Addressing this imbalance by adjusting thresholds per group may improve fairness, but doing so requires support for conditional logic in the model serving stack, access to sensitive attributes at inference time, and a governance framework for explaining and justifying differential treatment across groups.

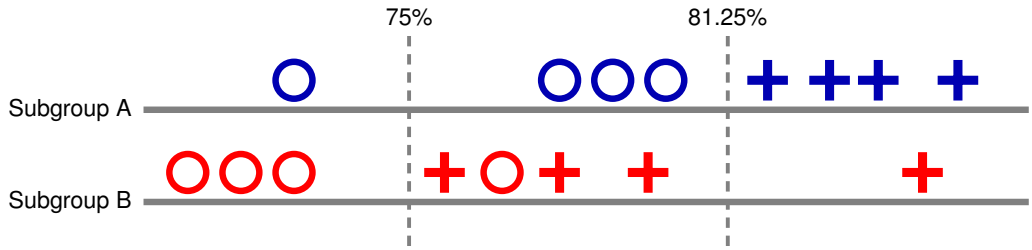


Figure 16.5: Threshold-Dependent Fairness: Varying classification thresholds across subgroups allows equal true positive rates but introduces complexity in model serving and necessitates access to sensitive attributes at inference time. Achieving fairness requires careful consideration of subgroup-specific performance, as a single threshold may disproportionately impact certain groups, highlighting the tension between accuracy and equitable outcomes in machine learning systems.

Fairness interventions may be applied at different points in the pipeline, but each comes with system-level implications. **Preprocessing methods**, which rebalance training data through sampling, reweighting, or augmentation, require access to raw features and group labels, often through a feature store or data lake that preserves lineage. These methods are well-suited to systems with centralized training pipelines and high-quality labeled data. In contrast, **in-processing approaches** embed fairness constraints directly into the optimization objective. These require training infrastructure that can support custom loss functions or constrained solvers and may demand longer training cycles or additional regularization validation. Training techniques and optimization methods, including custom loss functions and constrained optimization, provide the foundation for implementing these fairness-aware training approaches.

At the serving layer, **Postprocessing methods**, including the application of group-specific thresholds or the adjustment of scores to equalize outcomes, require inference systems that can condition on sensitive attributes or reference external policy rules. This demands coordination between model serving infrastructure, access control policies, and logging pipelines to ensure that differential treatment is both auditable and legally defensible. Model serving architectures, including request routing, feature lookup, and conditional inference paths, detail the infrastructure requirements for implementing such conditional logic in production systems. Any postprocessing strategy must be carefully validated to ensure that it does not compromise user experience, model stability, or compliance with jurisdictional regulations on attribute use.

Scalable fairness enforcement often requires more advanced strategies, such as multicalibration[25], which ensures that model predictions remain calibrated across a wide range of intersecting subgroups (Hébert-Johnson et al. 2018). Implementing multicalibration at scale requires infrastructure for dynamically generating subgroup partitions, computing per-group calibration error, and integrating fairness audits into automated monitoring systems. These capabilities are typically only available in large-scale, cloud-based deployments with mature observability and metrics pipelines. In constrained environments such as embedded or TinyML systems, where telemetry is limited and model logic is fixed, such techniques are not feasible and fairness must be validated entirely at design time.

25 **Multicalibration**: Developed by Hébert-Johnson et al. (2018), this technique seeks calibrated predictions across many computationally identifiable, intersecting subgroups, addressing the failure mode where global calibration masks severe miscalibration for minority intersections. The systems cost comes from repeated subgroup auditing and model updates rather than from a fixed multiplier. Multicalibration is one important scalable approach for diverse platforms where single-axis fairness audits miss compounded disparities, but it is not the only possible fairness strategy.

Across deployment environments, maintaining fairness requires lifecycle-aware mechanisms. Model updates, feedback loops, and interface designs all affect how fairness evolves over time. A fairness-aware model may degrade if retraining pipelines do not include fairness checks, if logging systems cannot track subgroup outcomes, or if user feedback introduces subtle biases not captured by training distributions. Monitoring systems must be equipped to surface fairness regressions, and retraining protocols must have access to subgroup-labeled validation data, which may require data governance policies and ethical review. Implementation of these monitoring systems requires production infrastructure for MLOps practices, while privacy-preserving techniques are essential for federated fairness assessment.

Fairness is not a one-time optimization, nor is it a property of the model in isolation. It emerges from coordinated decisions across data acquisition, feature engineering, model design, thresholding, feedback handling, and system monitoring. Embedding fairness into machine learning systems requires architectural foresight, operational discipline, and tooling that spans the full deployment stack, from training workflows to serving infrastructure to user-facing interfaces.

The sociotechnical implications of bias detection extend far beyond technical measurement. When fairness metrics identify disparities, organizations must navigate complex stakeholder deliberation processes as examined in section 16.7.3. These decisions involve competing stakeholder interests, legal compliance requirements, and value trade-offs that cannot be resolved through technical means alone.

16.4.1.1 Real-time fairness monitoring architecture

Implementing responsible AI principles in production systems requires architectural patterns that integrate fairness monitoring, explainability, and privacy controls directly into the model serving infrastructure. Figure 16.6 demonstrates how these responsible AI components integrate with existing ML systems infrastructure, showing the data flow from user requests through anonymization, model inference, fairness monitoring, and explanation generation.

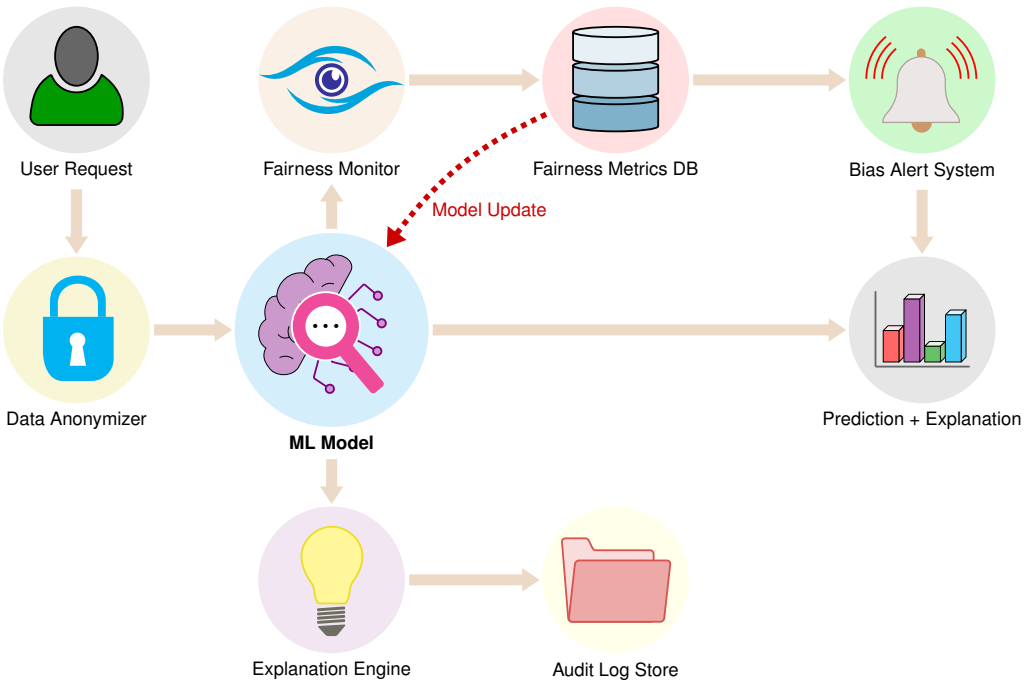


Figure 16.6: Production Responsible AI Architecture: Real-time fairness monitoring requires integrated components that process each inference request through data anonymization, bias detection, and explanation generation while maintaining audit trails and triggering alerts when fairness thresholds are violated. The dashed line shows the feedback loop for model updates based on detected bias patterns.

The architecture works because three governance paths share the model serving stack rather than sitting beside it. The data anonymization layer implements privacy-preserving transformations before model inference, using techniques like k-anonymity[26] or differential privacy noise injection. That layer is not free: privacy transformations consume latency, CPU, and memory, so they must be budgeted inside the model-serving SLO rather than treated as an external compliance step.

Real-time fairness monitoring updates prediction-only metrics such as demographic parity for each prediction, maintaining rolling statistics across protected groups. Label-dependent metrics such as equalized odds and equality of opportunity update asynchronously when delayed outcomes or audited labels arrive, and they remain inactive for groups whose labeled windows are too small to support a stable estimate. The system flags disparities that exceed configurable policy thresholds, while its storage and compute footprint scales with the number of protected groups, decision segments, and retention windows.

The explanation engine generates SHAP or LIME explanations for model decisions, particularly for negative outcomes requiring user recourse. Exact explanations often exceed inline serving budgets, so production systems rely on approximation, caching, sampling, or asynchronous generation to trade latency against fidelity. The implementation that follows should be read as an architectural pattern rather than an API to memorize: fairness metrics enter the serving path, and alerts trigger when thresholds are exceeded. Listing 16.2 integrates these components into a monitoring system that processes inference requests, computes prediction-time fairness metrics across protected groups, and updates label-dependent metrics such as equalized odds when ground-truth outcomes are available.

[26] **k-Anonymity**: A privacy guarantee (Sweeney 2002) ensuring each record is indistinguishable from at least $k - 1$ others by generalizing quasi-identifiers (for example, replacing exact ages with ranges, locations with regions). The systems trade-off is information loss: higher k values provide stronger indistinguishability but coarsen features and can reduce downstream training signal. For ML preprocessing pipelines, k-anonymity adds a transformation stage that must balance privacy guarantees against the accuracy impact of coarser features.

Listing 16.2: Production Fairness Monitoring: Bias detection that processes inference requests, computes rolling prediction-time fairness metrics, and updates label-dependent metrics when outcomes arrive.

```
import numpy as np


class RealTimeFairnessMonitor:
    def __init__(
        self,
        window_size=1000,
        alert_threshold=0.05,
        min_labeled_per_group=30,
    ):
        self.window_size = window_size
        self.alert_threshold = alert_threshold
        self.min_labeled_per_group = min_labeled_per_group
        self.window = []

    async def process_prediction(
        self, prediction, demographics, actual_label=None
    ):
        # Slide a fixed window over the live stream, then re-check
        # fairness.
        self.window.append((prediction, demographics, actual_label))
        self.window = self.window[-self.window_size :]
        metrics = self._compute_metrics()
        if (
            metrics["demographic_parity"] > self.alert_threshold
            or metrics["equalized_odds"] > self.alert_threshold
        ):
            await self._trigger_bias_alert(metrics)
        return metrics

    def _compute_metrics(self):
        # Bucket predictions and labels by protected group.
        by_group = {}
        for pred, demo, label in self.window:
            bucket = by_group.setdefault(
                demo.get("ethnicity", "unknown"),
                {"preds": [], "labels": []},
            )
            bucket["preds"].append(pred)
            if label is not None:
                bucket["labels"].append((pred, label))

        # Demographic parity: positive-prediction-rate spread.
        rates = [
            np.mean(d["preds"])
            for d in by_group.values()
            if d["preds"]
        ]
        parity = max(rates) - min(rates) if len(rates) > 1 else 0.0
        return {
            "demographic_parity": parity,
            "equalized_odds": self._equalized_odds_gap(by_group),
            "groups": {
                g: len(d["preds"]) for g, d in by_group.items()
            },
        }

    def _equalized_odds_gap(self, by_group):
        # Largest TPR/FPR gap across labeled groups; detail omitted.
        ...

    async def _trigger_bias_alert(
        self, metrics
    ): ...  # page on-call and append the event to the audit log
```

This production implementation demonstrates how responsible AI principles translate into concrete system architecture with quantifiable performance impacts. The overhead ranges summarized in table 16.4 explain why fairness monitoring, privacy transformations, and explanation generation must be provisioned as serving-path capabilities, not added after the model is already deployed. These overheads must be balanced against reliability and compliance requirements when designing production systems.

The monitoring cost is the price of turning fairness from an offline audit into an operational signal. Observation is still not remediation: fairness mitigation continues through preprocessing, in-processing constraints, postprocessing, and ongoing subgroup monitoring. The next layer shifts to a different responsibility boundary, where the system must prevent private data from leaking through logs, outputs, or model weights and must support deletion or revocation when users withdraw data rights.

Checkpoint 16.2: Operationalizing bias detection

This section turned static fairness definitions into live telemetry: slice-based detection with Fairlearn, CI-stage audits, production monitoring of demographic parity and equalized odds, and the distinction between observing a disparity and remediating it.

Detecting the disparity

- ☐ **Serving-path detection**: Explain why detection infrastructure must run as a serving-path capability rather than as a periodic offline audit when a deployed credit model begins rejecting one ZIP code at twice the normal rate.
- ☐ **Metric selection**: Select the fairness metric a monitoring dashboard should track for groups with different base rates, given approval rates of 94 percent for one group and 68 percent for another.

From signal to action

- ☐ **Detection vs. remediation**: Explain why a fairness alert with no remediation owner is evidence without action, and name what the system must add to close that loop.
- ☐ **Constrained deployments**: Reason about why slice-based detection may be infeasible in a federated deployment where demographic labels cannot be centralized, and infer the implication for safeguard selection.

16.5 Privacy, Unlearning, and Robustness Mitigations

When a user invokes erasure or deletion rights under privacy laws such as the GDPR, deleting their row from a database is straightforward (European Parliament and Council of the European Union 2016). Deleting their data from the weights of a neural network that has already trained on it is a fundamentally harder problem: the model *cannot* "forget" a specific face or credit card number through row deletion. The mitigation families differ by the boundary they protect:

- **Privacy mechanisms**: These controls reduce leakage.
- **Unlearning mechanisms**: These controls remove or bound retained influence.
- **Robustness mechanisms**: These controls preserve behavior under stress.
- **Validation mechanisms**: These controls produce evidence that the safeguards still work.

Machine unlearning[27] and privacy preservation address the deletion-and-leakage part of that challenge, providing mechanisms to excise specific training data influence from compiled models.

16.5.1 Privacy preservation

Privacy constraints extend across data collection, model behavior, and user interaction. They are shaped not only by ethical and legal obligations, but also by the architectural properties of the system and the context in which it is deployed. Technical methods for privacy preservation aim to prevent data leakage, limit memorization, and uphold user rights such as consent, opt-out, and data deletion, particularly in systems that learn from personalized or sensitive information.

27 **Machine Unlearning**: First formalized by Cao and Yang in 2015, this is the ability to remove specific training data influence from a model without full retraining. The naive approach (retrain from scratch) costs the same as the original training run. SISA (Sharded, Isolated, Sliced, and Aggregated) training partitions training into independent shards, reducing unlearning to retraining only the affected shard; in the GPT-scale sizing example below, that reduces the retraining surface dramatically at the cost of model-quality and aggregation trade-offs. For GDPR-compliant ML systems, unlearning latency becomes a service-level concern: deletion requests must be handled within the organization's applicable legal and policy timelines.

Large language models have been shown to memorize and expose individual training strings, including names, locations, or excerpts of private communication (Carlini et al. 2021). This memorization presents risks for privacy-sensitive text-generating systems, such as assistants trained or adapted on user logs, where training data may encode protected or regulated content. For example, a voice or chat assistant that adapts to user speech or messages may inadvertently retain specific phrases, which could later be extracted through carefully designed prompts or queries.

The memorization risk extends beyond language models. Figure 16.7 demonstrates that diffusion models trained on image datasets can regenerate visual instances from the training set (Carlini et al. 2023). The left panel is an original training image and the right panel is the diffusion model's reconstruction of that same instance; their near-identity is the evidence of memorization, since a model that had only learned a general distribution of portraits could not reproduce a specific individual. Such behavior highlights a more general vulnerability: contemporary generative model architectures can internalize and reproduce training data, often without explicit signals or intent, and without easy detection or control.

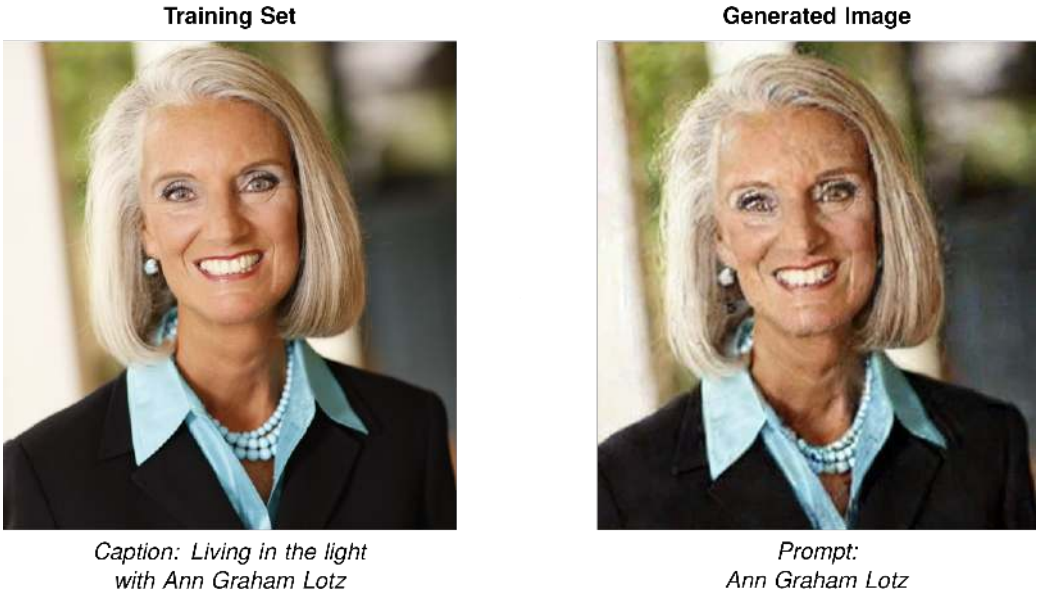


Figure 16.7: Diffusion Model Memorization: Image diffusion models can reproduce training samples (Carlini et al. 2023), revealing a risk of unintended memorization beyond language models and highlighting a general vulnerability in contemporary neural architectures. This memorization poses privacy concerns when training on sensitive datasets.

Models are also susceptible to membership inference attacks, in which adversaries attempt to determine whether a specific datapoint was part of the training set (Shokri et al. 2017). These attacks exploit subtle differences in model behavior between seen and unseen inputs. In high stakes applications such as healthcare or legal prediction, the mere knowledge that an individual's record was used in training may violate privacy expectations or regulatory requirements.

To mitigate such vulnerabilities, a range of privacy-preserving techniques have been developed. A canonical formal method is differential privacy[28], which provides guarantees that the inclusion or exclusion of a single datapoint has a statistically bounded effect on the model's output. Algorithms such as differentially private stochastic gradient descent (DP-SGD) operationalize this idea for deep learning by clipping gradients and injecting noise during training (Abadi et al. 2016). When the privacy accounting, clipping, and noise calibration are appropriate for the threat model, these methods can limit memorization and reduce the risk of inference attacks.

However, differential privacy introduces significant system-level trade-offs. The noise added during training can degrade model accuracy, increase the number of training iterations, and require access to larger datasets to maintain performance. These constraints are especially pronounced in resource-limited deployments such as mobile, edge, or embedded systems, where memory,

28 **Differential Privacy**: A mathematical framework introduced by Dwork et al. (2006) that bounds how much a single individual's inclusion or exclusion can affect a released computation, parameterized by a privacy budget such as ϵ. DP-SGD adapts that framework to deep learning by clipping per-example gradients and injecting calibrated noise (Abadi et al. 2016). The privacy budget quantifies a trade-off among privacy, utility, and training cost; Apple has publicly described local differential privacy deployments for selected telemetry tasks, but those deployments should not be treated as a universal DP-SGD cost benchmark (Apple Differential Privacy Team 2017).

compute, and power budgets are tightly constrained. In such settings, it may be necessary to combine lightweight privacy techniques (for example, feature obfuscation, local differential privacy) with architectural strategies that limit data collection, shorten retention, or enforce strict access control at the edge.

Systems Perspective 16.2: The price of privacy

Training a next-word predictor on sensitive messages with **DP-SGD** (Differentially Private Stochastic Gradient Descent) reduces training-data extraction risk, but the protection is priced in compute and accuracy.

The privacy parameter ϵ is the privacy budget. Lower ϵ means more privacy but more noise.

Strong privacy ($\epsilon = 1$): Gradients are heavily clipped ($C = 1.0$, clipping 40 percent of updates) and noisy ($\sigma = 1.0$). The model requires $3\times$ more epochs to converge. Training cost jumps from \$4.6M to approximately \$13.8M. Accuracy drops 6 percent.

Moderate privacy ($\epsilon = 8$, an illustrative setting): Less noise ($\sigma = 0.5$). Training overhead is 30 percent. Accuracy drops 1 percent.

Weak privacy ($\epsilon = 10$): Minimal noise. Less than 1 percent accuracy loss. Limited formal guarantees.

Privacy is not binary. It is a continuous curve where organizations buy user trust with compute dollars and model accuracy. The key engineering decision is allocating the privacy budget across the system lifecycle: how much ϵ to spend during training, how much to reserve for postdeployment queries, and how to communicate these trade-offs to users and regulators.

Privacy enforcement also depends on infrastructure beyond the model itself. Data collection interfaces must support informed consent and transparency. Logging systems must avoid retaining sensitive inputs unless strictly necessary, and must support access controls, expiration policies, and auditability. Model serving infrastructure must be designed to prevent overexposure of outputs that could leak internal model behavior or allow reconstruction of private data. These system-level mechanisms require close coordination between ML engineering, platform security, and organizational governance.

Privacy must be enforced not only during training but throughout the machine learning lifecycle. Retraining pipelines must account for deleted or revoked data, especially in jurisdictions with data deletion mandates. Monitoring infrastructure must avoid recording personally identifiable information in logs or dashboards. Privacy-aware telemetry collection, secure enclave deployment, and per-user audit trails can support these goals, particularly in applications with strict legal oversight.

Architectural decisions also vary by deployment context. Cloud-based systems may rely on centralized enforcement of differential privacy, encryption, and access control, supported by telemetry and retraining infrastructure. In contrast, edge and TinyML systems must build privacy constraints into the deployed model itself, often with no runtime configurability or feedback channel. In such cases, static analysis, conservative design, and embedded privacy guarantees must be implemented at compile time, with validation performed prior to deployment.

The privacy budget therefore stretches across the whole pipeline: technical safeguards, interface controls, logging and retention policies, and regulatory compliance mechanisms must work together to minimize risk throughout the lifecycle of a deployed system. No single mechanism, differential privacy included, suffices on its own.

Privacy preservation techniques create complex sociotechnical tensions that extend well beyond technical implementation. Differential privacy mechanisms may reduce model accuracy in ways that disproportionately affect underrepresented groups, creating conflicts between privacy and fairness objectives. These challenges require ongoing stakeholder engagement as detailed in section 16.7.3, where organizations must navigate competing values around data control, personalization, and regulatory compliance. These privacy challenges become even more complex when considering the dynamic nature of user rights and data governance.

16.5.1.1 Machine unlearning

The privacy mechanisms examined above protect data during collection and training, but they do not address a temporal problem: when users invoke their legal right to have their data forgotten, models trained on that data retain its influence in their learned parameters. The privacy violation persists even after the raw data is deleted from storage systems. Machine unlearning addresses this temporal dimension of privacy, ensuring that data deletion rights extend beyond databases to the models themselves.

Privacy preservation does not end at training time. In many real-world systems, users may have rights to revoke consent or request deletion of their data, even after a model has been trained and deployed (European Parliament and Council of the European Union 2016; California Legislature 2023). Supporting this requirement introduces a core technical challenge: removing the influence of specific datapoints from a trained model without full retraining, a task that is often infeasible in edge, mobile, or embedded deployments with constrained compute, storage, and connectivity.

Traditional approaches to data deletion assume that the full training dataset remains accessible and that models can be retrained from scratch after removing the targeted records. Figure 16.8 contrasts traditional model retraining with machine unlearning approaches: while retraining involves reconstructing the model from scratch using a modified dataset, unlearning aims to remove a specific datapoint's influence without repeating the entire learning process.

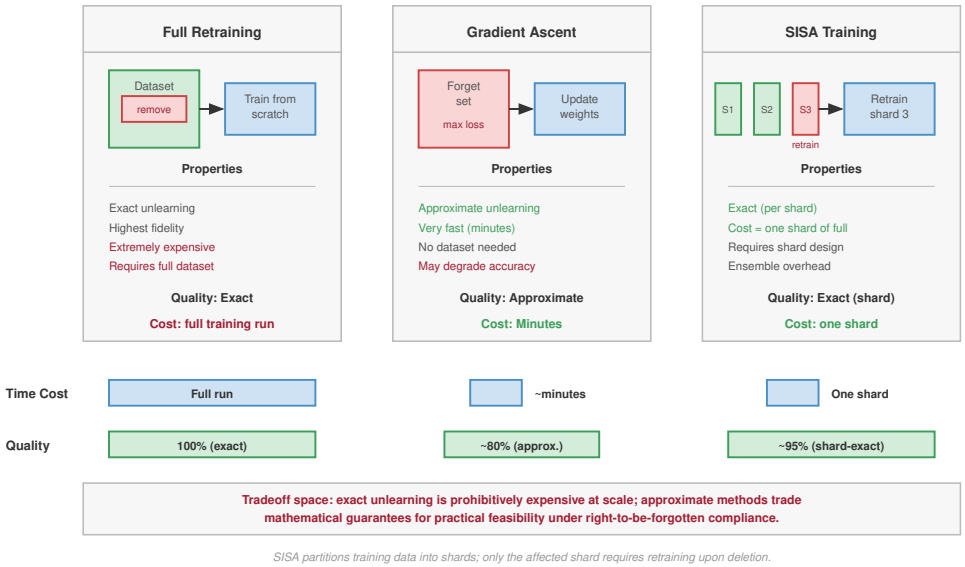


Figure 16.8: Model Update Strategies: Three approaches to removing a data point's influence. Full retraining gives exact unlearning at the highest cost; gradient ascent provides an approximate update by maximizing loss on the forget set; and SISA training partitions data into shards so only the affected shard is retrained. The cost model below quantifies the full-retraining and SISA cases.

The distinction between retraining and unlearning becomes critical in systems with tight latency, compute, or privacy constraints, because the assumptions underlying full retraining rarely hold in practice. Many deployed machine learning systems do not retain raw training data due to security, compliance, or cost constraints. In such environments, full retraining is often impractical and operationally disruptive, especially when data deletion must be verifiable, repeatable, and audit-ready.

Machine unlearning aims to address this limitation by removing the influence of individual datapoints from an already trained model without retraining it entirely (Cao and Yang 2015). Cao and Yang first formalized this problem, proposing a general approach that transforms learning algorithms into summation forms, enabling efficient removal of data influence by retraining only the

constituent models containing the targeted information rather than the entire model. Approaches represented by Cao and Yang's formulation and SISA-style training approximate this behavior by adjusting internal parameters, modifying gradient paths, or isolating and pruning components of the model so that the resulting predictions reflect what would have been learned without the deleted data (Bourtoule et al. 2021). These techniques may require simplified model architectures, additional tracking metadata, or compromise on model accuracy and stability. They also introduce new burdens around verification: how to prove that deletion has occurred in a meaningful way, especially when internal model state is not fully interpretable.

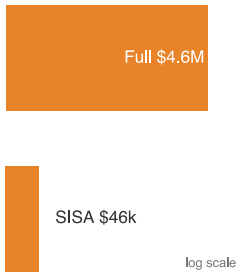
Napkin Math 16.3: The cost of forgetting

A user invokes GDPR Article 17 ("Right to Erasure") on a model trained on 1 TB of data.

Baseline (full retraining): In this sizing scenario, a 175B-parameter model at GPT-3 scale retrains on 1,024 A100 GPUs for approximately 34 days at a cost of roughly $4.6M.

Engineering fix (SISA): Sharded, Isolated, Sliced, and Aggregated training partitions data into $K = 100$ independent shards, training 100 sub-models. To delete one datum, retrain only the specific shard containing it (1 percent of data). New cost: $46,000. Time: approximately 8.2 hours.

Trade-off: Accuracy drops 3–7 percent because each sub-model sees less data. Inference slows because predictions must be aggregated across 100 sub-models. For a fleet receiving 1,000 requests/day in erasure traffic, SISA transforms unlearning from "economically impossible" to "manageable operational cost"—at the price of model quality.



SISA unlearning beats full retraining by about 100 times.

The motivation for machine unlearning is reinforced by regulatory frameworks. Deletion or erasure rights under laws such as the General Data Protection Regulation (GDPR), the California Consumer Privacy Act (CCPA), and similar statutes in other jurisdictions can create pressure to account for personal data used in training (European Parliament and Council of the European Union 2016; California Legislature 2023). Machine unlearning is a technical strategy for reducing or removing a deleted record's influence where model state itself may retain personal data, but the legal requirements are context-dependent and not a universal statutory command to retrain every model. High-profile incidents in which generative models have reproduced personal content or copyrighted data highlight the practical urgency of integrating deletion-aware mechanisms into responsible system design.

From a systems perspective, machine unlearning introduces nontrivial architectural and operational requirements. Systems must be able to track data lineage, including which datapoints contributed to a given model version. This often requires structured metadata capture and training pipeline instrumentation. Additionally, systems must support user-facing deletion workflows, including authentication, submission, and feedback on deletion status. Verification may require maintaining versioned model registries, along with mechanisms for confirming that the updated model exhibits no residual influence from the deleted data. These operations must span data storage, training orchestration, model deployment, and auditing infrastructure, and they must be robust to failure or rollback.

Resource-constrained deployments amplify these challenges further. TinyML systems typically run on devices with no persistent storage, no connectivity, and highly compressed models. Once deployed, they cannot be updated or retrained in response to deletion requests. In such settings, machine unlearning is effectively infeasible postdeployment and must be enforced during initial model development through static data minimization and conservative generalization strategies. Even in cloud-based systems, where retraining is more tractable, unlearning must contend with distributed training pipelines, replication across services, and the difficulty of synchronizing deletion across model snapshots and logs.

Machine unlearning is becoming important for responsible system design despite these challenges. As machine learning systems become more embedded, personalized, and adaptive, the ability to revoke training influence becomes central to maintaining user trust and meeting legal requirements. Critically, unlearning cannot be retrofitted after deployment. It must be considered during the

architecture and policy design phases, with support for lineage tracking, re-training orchestration, and deployment roll-forward built into the system from the beginning.

Machine unlearning represents a shift in privacy thinking, from protecting what data is collected to controlling how long that data continues to affect system behavior. This lifecycle-oriented perspective introduces new challenges for model design, infrastructure planning, and regulatory compliance, while also providing a foundation for more user-controllable, transparent, and adaptable machine learning systems. Responsible AI systems must also maintain reliable behavior under challenging conditions, including deliberate attacks.

16.5.2 Adversarial robustness

Adversarial robustness, examined in Chapter 14 and Chapter 13 as a defense against deliberate attacks, also serves as a foundation for responsible AI deployment. Beyond protecting against malicious adversaries, adversarial robustness ensures models behave reliably when encountering naturally occurring variations, edge cases, and inputs that deviate from training distributions. A model vulnerable to adversarial perturbations reveals fundamental brittleness in its learned representations, a brittleness that compromises trustworthiness even in nonadversarial contexts.

Machine learning models, particularly deep neural networks, are known to be vulnerable to small, carefully crafted perturbations that significantly alter their predictions. These vulnerabilities, first formalized through the concept of adversarial examples (Szegedy et al. 2013), highlight a gap between model performance on curated training data and behavior under real-world variability. A model that performs reliably on clean inputs may fail when exposed to inputs that differ only slightly from its training distribution, differences imperceptible to humans, but sufficient to change the model's output. NIST's adversarial machine learning taxonomy treats these failures as part of a broader attack and mitigation landscape that also includes poisoning, privacy attacks, attacker capabilities, and lifecycle stage (Vassilev et al. 2025).

The threat extends beyond theory. Adversarial examples have been used to manipulate real systems, including content moderation pipelines (Bhagoji et al. 2018), ad-blocking detection (Tramèr et al. 2019), and voice recognition models (Carlini et al. 2016). In safety-important domains such as autonomous driving or medical diagnostics, even rare failures can have high-consequence outcomes, compromising user trust or opening attack surfaces for malicious exploitation.

Figure 16.9 demonstrates how a visually negligible perturbation can cause confident misclassification, underscoring how subtle changes produce disproportionately harmful effects in safety-critical applications.

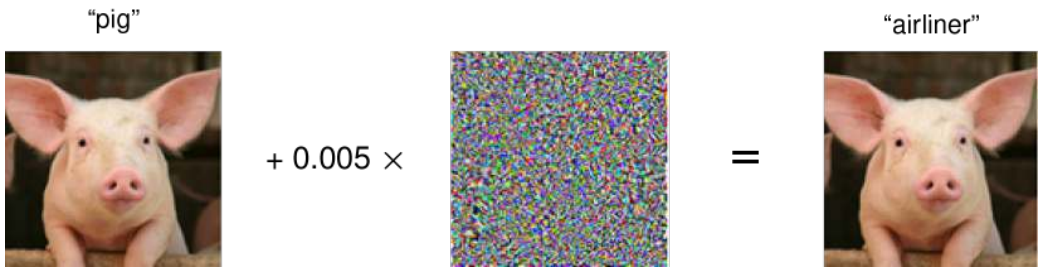


Figure 16.9: Adversarial Perturbation: An intentionally crafted noise pattern, when added to the original image of a pig, creates a new image that is visually imperceptible to humans but can cause a machine learning model to misclassify it with high confidence. Source: Microsoft.

At its core, adversarial vulnerability stems from an architectural mismatch between model assumptions and deployment conditions. Many training pipelines assume data is clean, independent, and identically distributed. In contrast, deployed systems must operate under uncertainty, noise, domain shift, and possible adversarial tampering. Robustness, in this context, encompasses not only the ability to resist attack but also the ability to maintain consistent behavior under degraded or unpredictable conditions.

Improving robustness begins at training. Adversarial training, a widely studied technique, augments training data with perturbed examples (Madry et al. 2018). Madry and colleagues formulated

adversarial training as a min-max optimization problem, training models against adversarial samples generated with Projected Gradient Descent (PGD)[29].

Adversarial training provides a principled framework for robust optimization that has become foundational in the field. It helps the model learn more stable decision boundaries but typically increases training time and reduces clean-data accuracy. Implementing adversarial training at scale also places demands on data preprocessing pipelines, model checkpointing infrastructure, and validation protocols that can accommodate perturbed inputs.

Architectural modifications can also promote robustness. Techniques that constrain a model's Lipschitz constant[30], regularize gradient sensitivity, or enforce representation smoothness can make predictions more stable.

These design changes must be compatible with the model's expressive needs and the underlying training framework. For example, smooth models may be preferred for embedded systems with limited input precision or where safety-important thresholds must be respected.

At inference time, systems may implement uncertainty-aware decision-making. Models can abstain from making predictions when confidence is low, or route uncertain inputs to fallback mechanisms, such as rule-based components or human-in-the-loop systems. These strategies require deployment infrastructure that supports fallback logic, user escalation workflows, or configurable abstention policies. For instance, a mobile diagnostic app might return "inconclusive" if model confidence falls below a specified threshold, rather than issuing a potentially harmful prediction.

Monitoring infrastructure plays a critical role in maintaining robustness postdeployment. Distribution shift detection, anomaly tracking, and behavior drift analytics allow systems to identify when robustness is degrading over time. Implementing these capabilities requires persistent logging of model inputs, predictions, and contextual metadata, as well as secure channels for triggering retraining or escalation. These tools introduce their own systems overhead and must be integrated with telemetry services, alerting frameworks, and model versioning workflows.

Beyond empirical defenses, formal approaches offer stronger guarantees. Certified defenses[31], such as randomized smoothing (Cohen et al. 2019), provide probabilistic assurances that a model's output will remain stable within a bounded input region.

Simpler defenses, such as input preprocessing, filter inputs through denoising, compression, or normalization steps to remove adversarial noise. These transformations must be lightweight enough for real-time execution, especially in edge deployments, and robust enough to preserve task-relevant features. Another approach is ensemble modeling, in which predictions are aggregated across multiple diverse models. This increases robustness but adds complexity to inference pipelines, increases memory footprint, and complicates deployment and maintenance workflows.

System constraints such as latency, memory, power budget, and model update cadence strongly shape which robustness strategies are feasible. Adversarial training increases model size and training duration, which may challenge CI/CD pipelines and increase retraining costs. Certified defenses demand computational headroom and inference time tolerance. Monitoring requires logging infrastructure, data retention policies, and access control. On-device and TinyML deployments, in particular, often cannot accommodate runtime checks or dynamic updates. In such cases, robustness must be validated statically and embedded at compile time.

Robustness emerges from coordination across training, model architecture, inference logic, logging, and fallback pathways. A model that appears robust in isolation may still fail if deployed in a system that lacks monitoring or interface safeguards. Conversely, even a partially robust model can contribute to overall system reliability if embedded within an architecture that detects uncertainty, limits exposure to untrusted inputs, and supports recovery when things go wrong. Responsible design therefore anticipates the ways in which models fail under real-world stress and builds the infrastructure that makes those failures detectable, recoverable, and safe.

16.5.3 Validation approaches

If detection identifies a problem and mitigation attempts to fix it, **validation** provides the evidence that stakeholders need to understand and audit whether the system is safe to deploy. This constitutes the third pillar of the responsible AI lifecycle. Unlike standard accuracy evaluation, which compresses performance into a single scalar metric, responsible validation is a multi-stakeholder process that interrogates the system's behavior under constraint. Different stakeholders require different proofs:

29 **PGD (Projected Gradient Descent)**: The standard first-order adversarial attack, as formalized in Madry et al. (2018), iteratively maximizes loss within an L_∞ perturbation ball, then projects back to the constraint boundary. Training against PGD examples can add substantial compute because each training step includes an inner adversarial-example search whose cost grows with the number of attack iterations. The result is a model trained for a specific threat model; transfer to unseen attack methods must still be validated rather than assumed.

30 **Lipschitz Constant**: A bound on how much a function's output changes relative to its input: $\|f(x_1) - f(x_2)\| \leq L_{\text{Lip}} \cdot \|x_1 - x_2\|$. For neural networks, a lower Lipschitz constant L_{Lip} limits sensitivity to small perturbations, directly constraining adversarial vulnerability. Techniques such as spectral normalization bound weight matrices to encourage smoother behavior, but the training cost and robustness gain depend on architecture and threat model.

31 **Certified Defenses**: Robustness guarantees backed by mathematical proof rather than empirical testing. Randomized smoothing (Cohen et al. 2019) averages predictions over many noise-perturbed inputs, yielding a provable robustness radius within which no adversarial perturbation can change the output. The trade-off is that certification can require many additional model evaluations and may reduce clean accuracy, making certified defenses more natural as offline validation gates or selective serving paths than as universal inline checks.

developers need granular debugging tools to isolate failure modes, auditors require statistical evidence of nondiscrimination for compliance, regulators mandate formal conformity assessments, and end users demand actionable explanations for specific decisions. The evidence bundle can include fairness audits, privacy-budget checks, adversarial and distribution-shift tests, explanation-fidelity checks, and revalidation triggers tied to deployment changes.

The engineering cost of this rigorous validation is substantial. A comprehensive regime that incorporates fairness audits, adversarial robustness testing, and explainability verification extends the model-evaluation phase materially. The investment is justified by where it moves the cost: issues caught in predeployment validation are remediated before release, whereas the same issues surfacing in production carry remediation costs orders of magnitude higher, because a deployed fairness or safety defect must be detected, rolled back across a fleet, and explained to users and regulators rather than fixed in a notebook. Validation is therefore not a *one-time gate* but a *continuous process*. A model that passes initial validation can drift into noncompliance as data distributions shift, requiring automated re-validation triggers in the deployment pipeline (Chapter 12).

Table 16.5 summarizes the evidence bundle by risk class. The table is not a separate checklist; it is a way to make residual risk explicit before deployment.

Table 16.5: Validation Evidence by Risk Class: Responsible validation combines several kinds of evidence because no single metric proves that a system is safe to deploy. Each risk class needs a measurable artifact and an operational owner.

Risk class	Evidence artifact	Operational owner
Fairness	Disaggregated metrics, thresholds, confidence intervals	Product owner and model-risk reviewer
Privacy	Privacy budget, deletion evidence, retention and access logs	Data-governance and privacy engineering teams
Robustness	Drift tests, corruption tests, canaries, red-team probes	ML platform and incident-response teams
Explanation	Fidelity tests, stability checks, recourse documentation	Model team and user-facing operations
Governance	Approval record, residual-risk owner, revalidation trigger	Review board or accountable launch authority

The most visible and computationally demanding form of validation is explainability. While fairness metrics provide aggregate statistical guarantees, explainability offers instance-level validation, allowing users and operators to verify why a specific decision was made. This bridges the gap between statistical correctness and individual trust.

16.6 Explainability and Interpretability

A loan officer using a traditional rules-based system can tell an applicant exactly why they were rejected: "Your debt-to-income ratio exceeds 40 percent." A neural network, however, outputs a rejection based on millions of dense matrix multiplications. Explainability and interpretability are the engineering techniques used to crack open this black box, allowing us to generate mathematically grounded, human-readable justifications for every high-stakes automated decision.

Explainability plays a central role in system validation, error analysis, user trust, regulatory compliance, and incident investigation. In high stakes domains such as healthcare, financial services, and autonomous decision systems, explanations help determine whether a model is making decisions for legitimate reasons or relying on spurious correlations. For instance, an explainability tool might reveal that a diagnostic model is overly sensitive to image artifacts rather than medical features, which is a failure mode that could otherwise go undetected. For qualifying automated decisions under GDPR-style access and notice provisions, systems may need to provide meaningful information about the logic involved, reinforcing the need for systematic support for explanation.

War Story 16.1: Apple Card: The cost of missing explanations

Context: In 2019, Apple and Goldman Sachs faced intense public scrutiny when prominent tech leaders, including Steve Wozniak, reported receiving credit limits 10× higher than their spouses despite shared finances and comparable credit profiles (New York State Department of Financial Services 2021).

Failure mode: The controversy centered on the engineering failure that compounded the disparity: when customers called to ask *why* they were denied, support staff could not answer. The algorithm offered no recourse, no explanation, and no mechanism for appeal.

Consequence: The New York Department of Financial Services investigated and found no fair-lending violations by Apple Card or Goldman Sachs, but the episode still exposed how perceived opacity, poor customer support, and limited recourse can undermine trust in high-stakes automated credit decisions.

Systems lesson: Explainability serves as a customer service interface, not merely a debugging tool. A system that cannot explain its high-stakes decisions is operationally fragile, regardless of its aggregate accuracy. When customers cannot understand or challenge outcomes, the absence of an explanation layer can turn an aggregate model decision into a reputational and governance crisis.

Explainability methods can be broadly categorized based on when they operate and how they relate to model structure. **Post-hoc methods** are applied after training and treat the model as a black box. These methods do not require access to internal model weights and instead infer influence patterns or feature contributions from model behavior. Common post-hoc techniques include feature attribution methods such as input gradients, Integrated Gradients[32] (Sundararajan et al. 2017), GradCAM[33] (Selvaraju et al. 2017), LIME (Ribeiro et al. 2016), and SHAP (Lundberg and Lee 2017). Sundararajan and colleagues introduced Integrated Gradients by identifying two fundamental axioms—Sensitivity and Implementation Invariance—that attribution methods should satisfy, demonstrating that most prior methods violated these properties.

Posthoc approaches are widely used in image and tabular domains, where explanations can be rendered as saliency maps or feature rankings. To illustrate how SHAP attribution works in practice, consider a trained random forest model predicting loan approval (approve $= 1$, deny $= 0$) based on three features: `income`, `debt_ratio`, and `credit_score`. For a specific applicant who was denied, with income of $45,000, debt ratio of 0.55 (55 percent of income goes to debt), and credit score of 620, the model predicts denial with probability 0.92. SHAP values, based on Shapley values from cooperative game theory, measure each feature's contribution to moving the prediction from a baseline (average prediction across all training data, $\Pr(\text{approve}) = 0.50$) to this individual prediction.

The SHAP framework[34] computes each feature's contribution by evaluating the model on all possible feature subsets. Starting from the baseline prediction of 0.50, adding income ($45K, slightly below average) decreases approval probability by 0.05.

Adding debt_ratio (0.55, high) strongly decreases approval by an additional 0.25. Adding credit_-score (620, below threshold) moderately decreases approval by 0.12. The final prediction becomes $0.50 - 0.05 - 0.25 - 0.12 = 0.08$, corresponding to $\Pr(\text{deny}) = 0.92$. This reveals that the high debt ratio contributed most strongly to the denial (-0.25), followed by the below-average credit score (-0.12), while income had minimal impact (-0.05). Such explanations are actionable: reducing debt ratio below 40 percent would likely flip the decision.

However, this rigor comes at significant computational cost. This 3-feature example requires evaluating $2^3 = 8$ feature subsets. For a model with 20 features, exact enumeration requires $2^{20} \approx 1$ million subset evaluations. Tree-based SHAP implementations exploit model structure to reduce this to polynomial time, but deep learning models typically require approximation algorithms (KernelSHAP, DeepSHAP) with sampling-based estimation. While SHAP provides theoretically grounded, additive feature attribution that satisfies desirable properties (local accuracy, missingness, consistency), these costs make SHAP impractical for real-time explanation in high-throughput systems without approximation, caching, or asynchronous serving strategies.

32 **Integrated Gradients**: An attribution method that integrates gradients along a path from a baseline input to the actual input. Unlike vanilla gradients, it satisfies two axioms (sensitivity and implementation invariance) that guarantee attributions change when features matter and remain consistent across functionally equivalent models (Sundararajan et al. 2017). The cost is higher than a single raw-gradient explanation because the method evaluates the model at multiple points along the baseline-to-input path, typically 50–300 discrete steps, making it better suited to bounded explanation budgets than to every inline prediction.

33 **GradCAM (Gradient-Weighted Class Activation Mapping)**: Selvaraju et al. (2017) generalized Class Activation Mapping to any convolutional neural network (CNN) architecture by using gradients flowing into the final convolutional layer to produce spatial importance maps. Because it reuses gradient computation, GradCAM can be far cheaper than perturbation-heavy explanation methods, but whether it fits real-time medical imaging or autonomous-vehicle pipelines depends on the model, hardware, batch size, and explanation-frequency budget.

34 **Shapley Values**: From cooperative game theory, Shapley values fairly distribute a payoff among players based on marginal contributions across all possible orderings. In ML explainability, features are "players" and the prediction is the "payoff." The mathematical guarantees (efficiency, symmetry, null player) make SHAP a widely used attribution framework with clear axiomatic appeal, but exact subset enumeration grows as 2^n for n features, which is why production systems often rely on model-specific algorithms, sampling approximations, caching, or asynchronous explanation paths (Lundberg and Lee 2017).

Another posthoc approach involves **counterfactual explanations**[35], which describe how a model's output would change if the input were modified in specific ways. These are especially relevant for decision-facing applications such as credit or hiring systems. For example, a counterfactual explanation might state that an applicant would have received a loan approval if their reported income were higher or their debt lower (Wachter et al. 2017). Counterfactual generation requires access to domain-specific constraints and realistic data manifolds, making integration into real-time systems challenging.

[35] **Counterfactual Explanations**: Formalized for ML by Wachter et al. (2017), counterfactuals answer "what would need to change?" rather than "why did this happen?" For regulatory compliance, they can provide actionable recourse: "if the applicant's income were $5,000 higher, the loan would be approved." Generating counterfactuals requires solving a constrained optimization problem that finds a feasible input change that flips the output, so latency and reliability depend on feature dimensionality, domain constraints, and whether immutable or monotone attributes are enforced.

A third class of techniques relies on concept-based explanations, which attempt to align learned model features with human-interpretable concepts. For example, TCAV represents user-defined concepts as concept activation vectors and tests how sensitive a model's predictions are to those concepts in its internal representation (B. Kim et al. 2018). These methods are especially useful in domains where subject matter experts expect explanations in familiar semantic terms. However, they require training data with concept annotations or auxiliary models for concept detection, which introduces additional infrastructure dependencies.

While posthoc methods are flexible and broadly applicable, they come with limitations. Because they approximate reasoning after the fact, they may produce plausible but misleading rationales. Their effectiveness depends on model smoothness, input structure, and the fidelity of the explanation technique. These methods are often most useful for exploratory analysis, debugging, or user-facing summaries, not as definitive accounts of internal logic.

In contrast, **inherently interpretable models** are transparent by design. Examples include decision trees, rule lists, linear models with monotonicity constraints, and k-nearest neighbor classifiers. These models expose their reasoning structure directly, enabling stakeholders to trace predictions through a set of interpretable rules or comparisons. In regulated or safety-important domains such as recidivism prediction or medical triage, inherently interpretable models may be preferred, even at the cost of some accuracy (Rudin 2019). However, these models generally do not scale well to high-dimensional or unstructured data, and their simplicity can limit performance in complex tasks.

Figure 16.10 visualizes the relative interpretability of different model types along a spectrum: decision trees and linear regression offer transparency by design, whereas more complex architectures like neural networks and convolutional models require external techniques to explain their behavior. This distinction is central to choosing an appropriate model for a given application, particularly in settings where regulatory scrutiny or stakeholder trust is paramount.



Figure 16.10: Model Interpretability Spectrum: Inherently interpretable models, such as linear regression and decision trees, offer transparent reasoning, while complex models like neural networks require posthoc explanation techniques to understand their predictions. This distinction guides model selection based on application needs, prioritizing transparency in regulated domains or when stakeholder trust is important.

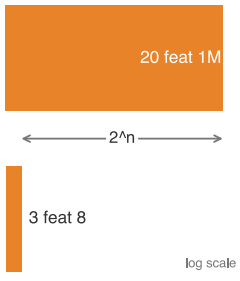
Hybrid approaches aim to combine the representational capacity of deep models with the transparency of interpretable components. Concept bottleneck models (Koh et al. 2020), for example, first predict intermediate, interpretable variables and then use a simple classifier to produce the final prediction. ProtoPNet models (Chen et al. 2019) classify examples by comparing them to learned prototypes, offering visual analogies for users to understand predictions. These hybrid methods are attractive in domains that demand partial transparency, but they introduce new system design considerations, such as the need to store and index learned prototypes and surface them at inference time.

Work such as Olah et al. and Geiger et al. applies **mechanistic interpretability** to reverse-engineer the internal operations of neural networks. This line of work, inspired by program analysis and neuroscience, attempts to map neurons, layers, or activation patterns to specific computational functions (Olah et al. 2020; Geiger et al. 2021). Many examples focus on large foundation mod-

els where traditional interpretability tools are insufficient, but the systems obligation is broader: instrumentation, storage, and causal tests must make internal mechanisms auditable.

From a systems perspective, explainability introduces a number of architectural dependencies. Explanations must be generated, stored, surfaced, and evaluated within system constraints. The required infrastructure may include explanation APIs, memory for storing attribution maps, visualization libraries, and logging mechanisms that capture intermediate model behavior. Models must often be instrumented with hooks or configured to support repeated evaluations, particularly for explanation methods that require sampling, perturbation, or backpropagation.

These requirements interact directly with deployment constraints and impose performance costs that must be factored into system design. SHAP and LIME can require repeated model evaluations, perturbation sampling, surrogate fitting, or subset enumeration, while counterfactual methods require constrained optimization (Lundberg and Lee 2017; Ribeiro et al. 2016; Wachter et al. 2017). In production deployments, these costs translate into concrete architecture choices: explanations may need asynchronous workers, sampled explanation rates, caching, or separate models chosen partly for interpretability.



Exact SHAP cost explodes with feature count.

For resource-constrained environments, gradient-based attribution methods offer more efficient alternatives by reusing backpropagation infrastructure from training. However, these methods are less reliable for complex models and may produce inconsistent explanations across model updates. Edge deployments often implement explainability through precomputed rule approximations or simplified decision boundaries, sacrificing explanation fidelity for feasible latency profiles.

Storage requirements also scale significantly with explanation needs. Storing SHAP values for tabular data requires approximately 4–8 bytes per feature per prediction, so monthly storage is 30 million predictions times the feature count times 4–8 bytes for a system logging 1 million predictions daily. Gradient attribution maps for images can require 1–10 MB per explanation depending on resolution; logging every image explanation at the same 1 million-per-day rate would require roughly 30–300 TB per month unless explanations are sampled, compressed, or retained for a shorter window. These volumes necessitate careful data lifecycle management and retention policies.

Explainability spans the full machine learning lifecycle. During development, interpretability tools are used for dataset auditing, concept validation, and early debugging. At inference time, they support accountability, decision verification, and user communication. Postdeployment, explanations may be logged, surfaced in audits, or queried during error investigations. System design must support each of these phases, ensuring that explanation tools are integrated into training frameworks, model serving infrastructure, and user-facing applications.

Compression and optimization techniques also affect explainability. Pruning, quantization, and architectural simplifications often used in TinyML or mobile settings can distort internal representations or disable gradient flow, degrading the reliability of attribution-based explanations. In such cases, interpretability must be validated postoptimization to ensure that it remains meaningful and trustworthy. If explanation quality is important, these transformations must be treated as part of the design constraint space.

Explainability therefore lands the same way the chapter's other properties do: it is budgeted serving-path infrastructure, not a feature appended after the model works. Designing for interpretability requires careful decisions about who needs explanations, what kind of explanations are meaningful, and how those explanations can be delivered given the system's latency, compute, and interface budget. As machine learning becomes embedded in important workflows, the ability to explain becomes a core requirement for safe, trustworthy, and accountable systems.

The sociotechnical challenges of explainability center on the gap between technical explanations and human understanding. While algorithms can generate feature attributions and gradient maps, stakeholders often need explanations that align with their mental models, domain expertise, and decision-making processes. A radiologist reviewing an AI-generated diagnosis needs explanations that reference medical concepts and visual patterns, not abstract neural network activations. This translation challenge requires ongoing collaboration between technical teams and domain experts to develop explanation formats that are both technically accurate and practically meaningful. Explanations can shape human decision-making in unexpected ways, creating new responsibilities for how explanatory information is presented and interpreted. Because explanations are evidence rather

than control actions, they must feed a monitoring system that detects when responsible behavior changes after deployment.

16.6.1 Model performance monitoring

Training-time evaluations, no matter how rigorous, do not guarantee reliable model performance once a system is deployed. Real-world environments are dynamic: input distributions shift due to seasonality, user behavior evolves in response to system outputs, and contextual expectations change with policy or regulation. These factors can cause predictive performance and system trustworthiness to degrade over time. A model that performs well under training or validation conditions may still make unreliable or harmful decisions in production.

The implications of such drift extend beyond raw accuracy. Fairness guarantees may break down if subgroup distributions shift relative to the training set, or if features that previously correlated with outcomes become unreliable in new contexts. Interpretability demands may also evolve, for instance as new stakeholder groups seek explanations, or as regulators introduce new transparency requirements. Trustworthiness, therefore, is not a static property conferred at training time, but a dynamic system attribute shaped by deployment context and operational feedback.

To ensure responsible behavior over time, machine learning systems must incorporate mechanisms for continual monitoring, evaluation, and corrective action. Monitoring involves more than tracking aggregate accuracy; it requires surfacing performance metrics across relevant subgroups, detecting shifts in input distributions, identifying anomalous outputs, and capturing meaningful user feedback. These signals must then be compared to predefined expectations around fairness, robustness, and transparency, and linked to actionable system responses such as model retraining, recalibration, or rollback.

Implementing effective monitoring depends on robust infrastructure. Systems must log inputs, outputs, and contextual metadata in a structured and secure manner, feeding a continuous observability pipeline (figure 16.11).

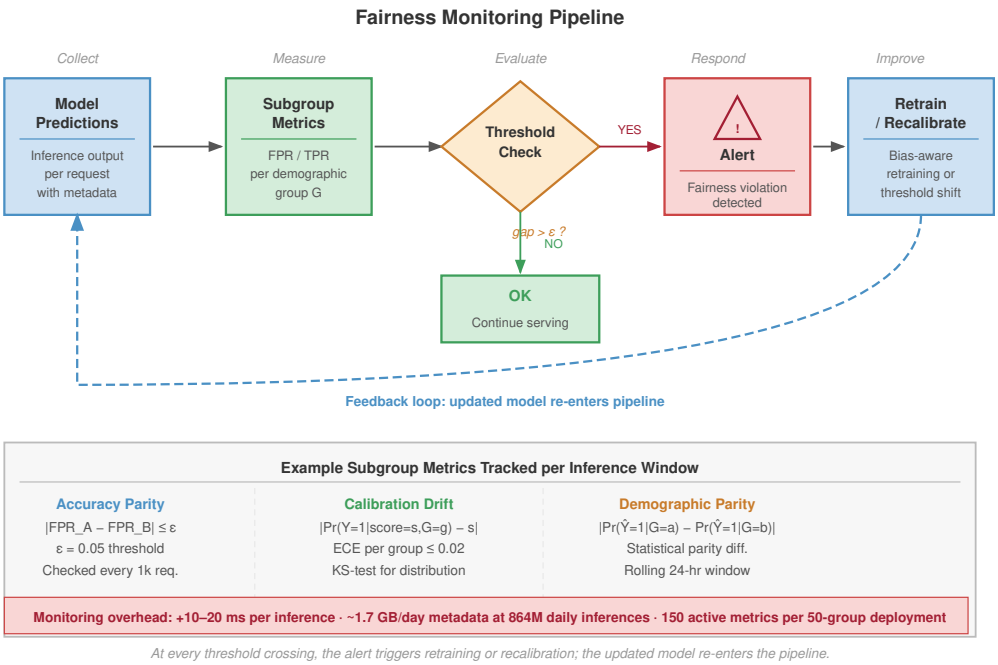
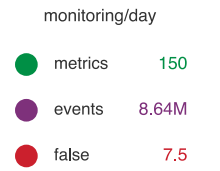


Figure 16.11: Fairness Monitoring Pipeline: End-to-end observability for deployed models. Model predictions feed subgroup metric computation across demographic segments; a threshold check identifies performance or fairness regressions; and alerts trigger automated retraining or manual review. This continuous feedback loop ensures that responsible AI properties are maintained postdeployment.

This requires telemetry pipelines that capture model versioning, input characteristics, prediction confidence, and postinference feedback. These logs support drift detection and provide evidence for retrospective audits of fairness and robustness. Monitoring systems must also be integrated with alerting, update scheduling, and policy review processes to support timely and traceable intervention.

Monitoring also supports feedback-driven improvement. For example, repeated user disagreement, correction requests, or operator overrides can signal problematic behavior. This feedback must be aggregated, validated, and translated into updates to training datasets, data labeling processes, or model architecture. However, such feedback loops carry risks: biased user responses can introduce new inequities, and excessive logging can compromise privacy. Designing these loops requires careful coordination between user experience design, system security, and ethical governance.



Responsible AI monitoring becomes its own production data system.

At the scale of a global production fleet, responsible AI monitoring becomes a massive data engineering challenge. A platform serving roughly 864 million inferences per day at 10,000 QPS across 50 distinct demographic subgroups must track at least 150 metrics continuously (for example, false positive rate, true positive rate, and calibration error for each of the 50 groups). Even with a 1 percent sampling rate, this generates 8.64 million monitoring events daily. Storing the necessary metadata—prediction inputs, confidence scores, ground truth labels, and sensitive attributes—at a modest 200 bytes per record requires approximately 1.7 GB per day of storage, while full audit logging can consume substantially more. This scale introduces a meta-monitoring problem: the monitoring infrastructure itself becomes a complex distributed system that must be reliable, secure, and cost-effective. With 150 active metrics, a standard false alarm rate of just 5 percent would trigger roughly 7.5 spurious alerts every day, leading to severe alert fatigue. Effective monitoring therefore requires intelligent aggregation, hierarchical alerting logic, and automated root cause analysis to distinguish genuine fairness drift from statistical noise.

Monitoring mechanisms vary by deployment architecture. In cloud-based systems, rich logging and compute capacity allow for real-time telemetry, scheduled fairness audits, and continuous integration of new data into retraining pipelines. These environments support dynamic reconfiguration and centralized policy enforcement. However, the volume of telemetry may introduce its own challenges in terms of cost, privacy risk, and regulatory compliance.

In mobile systems, connectivity is intermittent and data storage is limited. Monitoring must be lightweight and resilient to synchronization delays. Local inference systems may collect performance data asynchronously and transmit it in aggregate to backend systems. Privacy constraints are often stricter, particularly when personal data must remain on-device. These systems require careful data minimization and local aggregation techniques to preserve privacy while maintaining observability.

Edge deployments, such as those in autonomous vehicles, smart factories, or real-time control systems, demand low-latency responses and operate with minimal external supervision. Monitoring in these systems must be embedded within the runtime, with internal checks on sensor integrity, prediction confidence, and behavior deviation. These checks often require low-overhead implementations of uncertainty estimation, anomaly detection, or consistency validation. System designers must anticipate failure conditions and ensure that anomalous behavior triggers safe fallback procedures or human intervention.

TinyML systems, which operate on deeply embedded hardware with no connectivity, persistent storage, or dynamic update path, present the most constrained monitoring scenario. In these environments, monitoring must be designed and compiled into the system prior to deployment. Common strategies include input range checking, built-in redundancy, static failover logic, or conservative validation thresholds. Once deployed, these models operate independently, and any postdeployment failure may require physical device replacement or firmware-level reset.

The core challenge is universal: deployed ML systems must not only perform well initially, but continue to behave responsibly as the environment changes. Monitoring provides the observability layer that links system performance to ethical goals and accountability structures. Without monitoring, fairness and robustness become invisible. Without feedback, misalignment cannot be corrected. Monitoring, therefore, is the operational foundation that allows machine learning systems to remain adaptive, auditable, and aligned with their intended purpose over time.

The monitoring section closes the technical loop: bias detection, differential privacy, adversarial training, and explainability provide essential capabilities for responsible AI implementation, but

they also reveal a fundamental limitation. Technical correctness alone cannot guarantee beneficial outcomes. Consider three concrete examples that illustrate this challenge:

A fairness auditing system detects racial bias in a loan approval model, but the organization lacks processes for interpreting results or implementing corrections. The technical capability exists, but organizational inertia prevents remediation. Differential privacy preserves formal mathematical guarantees about data protection, but users do not understand these protections and continue to share sensitive information inappropriately. The privacy method works as designed, but behavioral context undermines its effectiveness. An explainability system generates technically accurate feature importance scores, but affected individuals cannot access or interpret these explanations due to interface design and literacy barriers.

These examples demonstrate that responsible AI implementation depends on alignment between technical capabilities and sociotechnical contexts, organizational incentives, human behavior, stakeholder values, and institutional governance structures. Sustaining that alignment requires monitoring mechanisms that provide operational observability. However, the emergence of generative AI has transformed the nature of the "failures" we must monitor.

16.6.2 Responsibility in the generative era

Generative AI does not replace the fairness, privacy, and explainability concerns above; it changes the control surface. Instead of only auditing labels or feature attributions, operators must govern prompts, retrieved context, reward models, and open-ended outputs.

The transition from discriminative classification to generative large language models (LLMs) fundamentally alters the engineering surface of responsibility. Fairness is no longer merely a statistical parity metric between labeled groups; it evolves into **Generative Alignment**, the complex optimization problem of constraining open-ended stochastic outputs to remain helpful, harmless, and honest across a combinatorial explosion of possible prompts. This requires a transition from *static dataset curation* to *dynamic behavioral shaping*, typically through a multi-stage alignment process (figure 16.12).

A common instruction-tuning pipeline in large language models circa 2022–2024 uses Reinforcement Learning from Human Feedback (RLHF) as a sociotechnical bridge between human values and model weights. By training a reward model on human preferences—in the scenario here, 50,000 to 500,000 pairwise comparisons at a cost of $0.50 to $5.00 per label—engineers effectively compile subjective judgments into a differentiable loss function. Proximal Policy Optimization (PPO) is the policy-optimization stage that updates the model against this reward model while constraining how far the policy moves from the supervised baseline. This alignment process introduces an **alignment tax**, often observed as a 2–8 percent degradation in standard NLP benchmarks as the model trades raw capability for safety constraints. The reliance on human raters introduces a **representativeness gap**: if the labeling investment reflects only a narrow demographic slice, the resulting "aligned" model will inherently overfit to that specific cultural or socioeconomic context. Constitutional AI offers an alternative engineering path, using a set of high-level principles to guide AI feedback on its own outputs, thereby reducing the dependency on massive-scale human annotation while making the values explicit in the prompt rather than implicit in the rater pool.

In Retrieval-Augmented Generation (RAG) architectures (Chapter 10), responsibility becomes decoupled from the core model. An LLM may be perfectly aligned via extensive RLHF, yet still generate toxic or biased responses if the retrieval layer surfaces contaminated context. If a retrieval index disproportionately surfaces biased historical documents, the model—conditioned to be faithful to its context—will propagate that bias regardless of its internal safety training. This necessitates **context filtering** as a distinct infrastructure component, validating retrieved chunks for toxicity and bias before they reach the generation context window.

In many LLM serving stacks, the **system prompt** operates as an early configuration control alongside retrieval filtering, policy classifiers, telemetry, and rollout gates. These hidden instructions (for example, "You are a helpful assistant. Do not provide medical advice.") define operational boundaries of the system. At the scale of 10,000+ distinct deployment configurations, managing these prompts becomes a distributed configuration management problem akin to weight distribution. A single unversioned change to a system prompt can subtly shift the ethical posture of millions of

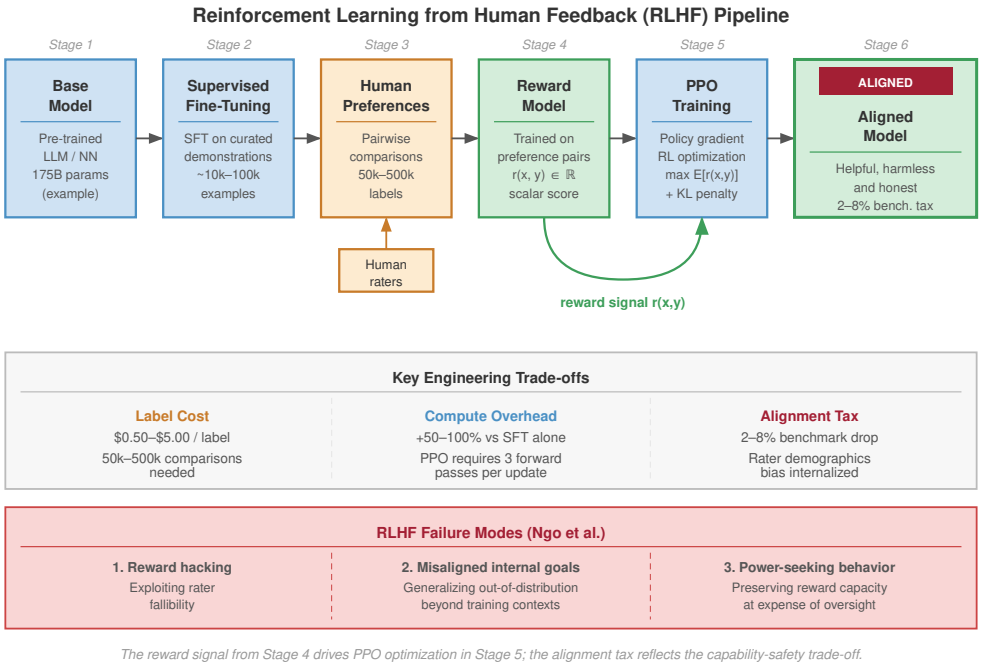


Figure 16.12: Reinforcement Learning from Human Feedback (RLHF) Alignment Pipeline: Six-stage process for aligning generative models: (1) a pretrained Base Model, (2) Supervised Fine-Tuning (SFT) on demonstrations, (3) Human Preference collection, (4) Reward Model training, (5) Proximal Policy Optimization (PPO) Training, and (6) a final Aligned Model. The figure also annotates label cost, compute overhead, and alignment tax, plus three RLHF failure modes (reward hacking, misaligned internal goals, power-seeking behavior).

interactions, making prompt version control, alignment regression testing, and gradual rollouts as critical for safety as the model training process itself (Chapter 12).

Together, system prompts and RLHF alignments act as important, yet fragile, technical guardrails. When these guardrails fail, whether through deliberate jailbreaking or nuanced edge cases that bypass the reward model, the reality becomes clear: AI safety *cannot* be solved entirely through mathematics. The complex sociotechnical dynamics between the algorithm and the human using it demand equal attention.

16.7 Sociotechnical Dynamics

A hospital deployed a highly accurate sepsis prediction model, but mortality rates did not improve. The doctors, overwhelmed by alert fatigue, simply ignored the model's warnings. A mathematically flawless, perfectly fair, highly explainable model still fails spectacularly in production when it misaligns with human psychology, organizational incentives, or the operational reality of the workplace.

The technical tools of the previous sections solved well-defined problems: detecting bias, preserving privacy, generating explanations. The sepsis case marks where that toolbox runs out. Sociotechnical engineering demands a different mode of reasoning: instead of optimizing an objective function, we analyze stakeholder conflicts; instead of tuning hyperparameters, we navigate ethical trade-offs; instead of measuring technical performance, we assess social impact. The system must now satisfy both computational constraints and human values, and no amount of optimization resolves a conflict between the two.

Deployed systems create feedback loops that reshape the environments they model, introduce human-AI collaboration risks that neither humans nor algorithms can address alone, and surface stakeholder value conflicts that no optimization can satisfy. These dynamics determine whether responsible AI implementations succeed or fail in practice.

16.7.1 System feedback loops

The sociotechnical feedback invariant (principle 21) captures this dynamic: deployed models shape the environment they operate in, so that future data $p_{t+1}(X)$ is a function of the model's past decisions $f_t(X)$. Here, $p_t(X)$ denotes the input distribution observed by the system at time t, while $f_t(X)$ denotes the deployed model or decision policy acting on those inputs. Systems require **Closed-Loop Governance**—reliability requires modeling the feedback loop, not just the feed-forward inference.

Machine learning systems do not merely observe and model the world; they also shape it. Once deployed, their predictions and decisions often influence the environments they are intended to analyze. This feedback alters future data distributions, modifies user behavior, and affects institutional practices, creating a recursive loop between model outputs and system inputs (figure 16.4). Over time, such dynamics can amplify biases, entrench disparities, or unintentionally shift the objectives a model was designed to serve.

A well-documented example of this phenomenon is predictive policing. When a model trained on historical arrest data predicts higher crime rates in a particular neighborhood, law enforcement may allocate more patrols to that area. This increased presence leads to more recorded incidents, which are then used as input for future model training, further reinforcing the model's original prediction. Even if the model was not explicitly biased at the outset, its integration into a feedback loop results in a self-fulfilling pattern that disproportionately affects already over-policed communities.

Recommender systems exhibit similar dynamics in digital environments. A content recommendation model that prioritizes engagement may gradually narrow the range of content a user is exposed to, leading to feedback loops that reinforce existing preferences or polarize opinions. These effects can be difficult to detect using conventional performance metrics, as the system continues to optimize its training objective even while diverging from broader social or epistemic goals.

From a systems perspective, feedback loops present a core challenge to responsible AI. They undermine the assumption of independently and identically distributed data and complicate the evaluation of fairness, robustness, and generalization. Standard validation methods, which rely on static test sets, may fail to capture the evolving impact of the model on the data-generating process. Once such loops are established, interventions aimed at improving fairness or accuracy may have limited effect unless the underlying data dynamics are addressed.

Designing for responsibility in the presence of feedback loops requires a lifecycle view of machine learning systems. It entails not only monitoring model performance over time, but also understanding how the system's outputs influence the environment, how these changes are captured in new data, and how retraining practices either mitigate or exacerbate these effects.

In cloud-based systems, these updates may occur frequently and at scale, with extensive telemetry available to detect behavior drift. In contrast, edge and embedded deployments often operate offline or with limited observability. A smart home system that adapts thermostat behavior based on user interactions may reinforce energy consumption patterns or comfort preferences in ways that alter the home environment, and subsequently affect future inputs to the model. Without connectivity or centralized oversight, these loops may go unrecognized, despite their impact on both user behavior and system performance. Operational monitoring practices, including drift detection, performance tracking, and automated alerting, are crucial for detecting and managing these feedback dynamics in production systems.

Systems must be equipped with mechanisms to detect distributional drift, identify behavior shaping effects, and support corrective updates that align with the system's intended goals. Feedback loops are not inherently harmful, but they must be recognized and managed. When left unexamined, they introduce systemic risk; when thoughtfully addressed, they provide an opportunity for learning systems to adapt responsibly in complex, dynamic environments.

War Story 16.2: The algorithmic grading failure

Context: In 2020, following the cancellation of A-level exams due to the COVID-19 pandemic, Ofqual (the qualifications regulator for England) deployed an algorithmic standardization model to assign grades. The model, intended to combat grade inflation, used a school's

historical performance distribution to adjust individual teacher predictions (Ofqual 2020; Department for Education and Ofqual 2020).

Failure mode: While designed to maintain aggregate standards, the engineering constraint of preserving historical grade distributions decoupled many individual outcomes from teacher assessment. Students at historically high-performing schools were more likely to see predictions preserved, while high-achieving students at schools with weaker historical results could be downgraded to fit the school's statistical prior.

Consequence: The algorithm enforced a feedback loop where past institutional performance constrained future individual outcomes. The resulting loss of confidence forced the government and Ofqual to revert to centre assessment grades days later.

Systems lesson: Optimizing for aggregate statistical properties (preventing inflation) without constraints on individual fairness (rank preservation) creates a system that is mathematically "correct" but socially catastrophic.

16.7.2 Human-AI collaboration

Human operators turn feedback-loop risk into a shared-control problem. Machine learning systems are often deployed not as standalone agents, but as components in larger workflows that involve human decision-makers. In many domains, such as healthcare, finance, and transportation, models serve as decision-support tools, offering predictions, risk scores, or recommendations that are reviewed and acted upon by human operators. The collaborative configuration raises questions about how responsibility is shared between humans and machines, how trust is calibrated, and how oversight mechanisms are implemented in practice.

Human-AI collaboration introduces both opportunities and risks. When designed appropriately, systems can augment human judgment, reduce cognitive burden, and enhance consistency in decision-making. However, when poorly designed, they may lead to **automation bias**[36], where users over-rely on model outputs even in the presence of clear errors.

Conversely, excessive distrust can result in **algorithm aversion**, where users disregard useful model predictions due to a lack of transparency or perceived credibility. The effectiveness of collaborative systems depends not only on the model's performance, but on how the system communicates uncertainty, provides explanations, and allows for human override or correction.

Automation bias is often reinforced by institutional structures through asymmetric liability. In high stakes domains like criminal justice or healthcare, human decision-makers face different consequences based on their agreement with algorithms. Consider two scenarios: In Scenario A, a judge overrides a "high risk" algorithmic score and releases a defendant who later re-offends. The judge faces public scrutiny and potential career consequences for "ignoring the science." In Scenario B, a judge follows the "high risk" score and detains the defendant unnecessarily. The blame is diffused to the algorithm ("the system said so").

The asymmetry creates strong pressure for **Institutional Deference**, where human oversight becomes a "rubber stamp" for algorithmic decisions to avoid personal liability. Responsible AI design must explicitly counter this by protecting operators who exercise judgment and requiring justification for *agreement* as well as *disagreement*.

Oversight mechanisms must be tailored to the deployment context. In high stakes domains, such as medical triage or autonomous driving, humans may be expected to supervise automated decisions in real-time. This configuration places cognitive and temporal demands on the human operator and assumes that intervention will occur quickly and reliably when needed. In practice, however, continuous human supervision is often impractical or ineffective, particularly when the operator must monitor multiple systems or lacks clear criteria for intervention.

From a systems design perspective, supporting effective oversight requires more than providing access to raw model outputs. Interfaces must be constructed to surface relevant information at the right time, in the right format, and with appropriate context. Confidence scores, uncertainty estimates, explanations, and change alerts can all play a role in enabling human oversight. Workflows must define when and how intervention is possible, who is authorized to override model outputs, and how such overrides are logged, audited, and incorporated into future system updates.

36 **Automation Bias**: First studied in aviation in the 1990s, this is the paradox where humans defer to automated systems even when clearly wrong – and the effect intensifies as system accuracy increases. At 70–80 percent model accuracy, operators accept erroneous outputs at high rates when presented without uncertainty indicators. For ML serving systems, this means higher model accuracy can paradoxically reduce system-level safety by suppressing human oversight, requiring deliberate interface friction (uncertainty visualization, mandatory justification) that adds latency but preserves the human correction channel.

Consider a hospital triage system that uses a machine learning model to prioritize patients in the emergency department. The model generates a risk score for each incoming patient, which is presented alongside a suggested triage category. In principle, a human nurse is responsible for confirming or overriding the suggestion. However, if the model's outputs are presented without sufficient justification, such as an explanation of the contributing features or the context for uncertainty, the nurse may defer to the model even in borderline cases. Over time, the model's outputs may become the de facto triage decision, especially under time pressure. If a distribution shift occurs (for instance, due to a new illness or change in patient demographics), the nurse may lack both the situational awareness and the interface support needed to detect that the model is underperforming. In such cases, the appearance of human oversight masks a system in which responsibility has effectively shifted to the model without clear accountability or recourse.

In such systems, human oversight is not merely a matter of policy declaration, but a function of infrastructure design: how predictions are surfaced, what information is retained, how intervention is enacted, and how feedback loops connect human decisions to system updates. Without integration across these components, oversight becomes fragmented, and responsibility may shift invisibly from human to machine.



Napkin Math 16.4: The automation bias paradox

Consider a radiology department deploying an AI assistant for tumor detection.

- **Human sensitivity**: $S_{\text{human}} = 92$ percent
- **AI sensitivity**: $S_{\text{AI}} = 95$ percent

One might assume the combined system performance would exceed 95 percent. However, studies in automation bias show that humans accept erroneous AI recommendations at rates of 60–80 percent. If the AI makes an error (probability $1 - S_{\text{AI}} = 0.05$) and the human blindly accepts it ($\alpha = 0.7$), accepted AI errors alone create a failure probability of $0.05 \times 0.7 = 0.035$. In this simplified upper-bound case, where every non-accepted AI error is corrected, sensitivity is 96.5 percent; real workflows can do worse once human override errors, false positives, fatigue, and interface delays are included.

As AI reliability increases, human vigilance decreases—a phenomenon known as the paradox of reliability.

- At 90 percent AI accuracy, human override rate might be $R_{\text{override}} = 15\%$.
- At 99 percent AI accuracy, R_{override} drops to $\approx 2\%$.

The remaining 1 percent of errors are almost never caught because the human has calibrated their trust to the "perfect" machine. This creates a **trust calibration gap**: *the safer the system appears, the more dangerous its rare failures become.* Responsible design requires introducing friction—forcing the human to justify acceptance—to artificially lower α and maintain the human in the loop.

The boundary between decision support and automation is often fluid. Systems initially designed to assist human decision-makers may gradually assume greater autonomy as trust increases or organizational incentives shift. This transition can occur without explicit policy changes, resulting in de facto automation without appropriate accountability structures. Responsible system design must therefore anticipate changes in use over time and ensure that appropriate checks remain in place even as reliance on automation grows.

Human-AI collaboration requires careful integration of model capabilities, interface design, operational policy, and institutional oversight. Collaboration is not simply a matter of inserting a "human-in-the-loop"; it is a systems challenge that spans technical, organizational, and ethical dimensions. Designing for oversight entails embedding mechanisms that allow intervention, support informed trust, and support shared responsibility between human operators and machine learning systems.

16.7.3 Normative pluralism and value conflicts

Human-AI collaboration exposes a deeper systems constraint: different stakeholders often hold conflicting values and priorities. Real-world ML deployment forces a confrontation with value tensions that no algorithm can resolve. Technical excellence is necessary but insufficient for trustworthy AI, because stakeholders hold legitimately different conceptions of fairness, privacy, and accountability that cannot be reconciled through better algorithms.

Responsible machine learning cannot be reduced to the optimization of a single objective. In real-world settings, machine learning systems are deployed into environments shaped by diverse, and often conflicting, human values. A high-stakes deployment makes these tensions concrete.

Example 16.5: Conflicting values in practice

Scenario: A team builds a mental health chatbot for adolescents that uses ML to detect crisis situations and recommend interventions. The system must balance multiple legitimate but incompatible objectives:

Medical efficacy: Optimize for best clinical outcomes based on evidence-based practices. This suggests aggressive intervention, alerting parents, counselors, or emergency services whenever the model detects potential self-harm risk, even with low confidence, because false negatives could be fatal.

Patient autonomy: Respect adolescent privacy and agency. Many teenagers seek mental health support specifically because they cannot talk to parents or authority figures. Aggressive notification policies may deter vulnerable teens from using the system at all, leaving them without any support.

Privacy protection: Minimize data collection and retention to protect sensitive mental health information. This suggests local processing, no conversation logging, and no sharing with third parties, but also prevents the system from improving through learning from interactions or enabling human review when the model is uncertain.

Resource efficiency: Operate within computational and human oversight budgets. Involving human counselors for every flagged interaction provides better care but is prohibitively expensive at scale. Fully automated responses reduce costs but may provide inappropriate guidance in complex situations.

Legal compliance: Meet mandatory reporting requirements and liability standards. In many jurisdictions, systems that detect imminent harm must notify authorities, overriding patient autonomy and privacy regardless of clinical judgment about whether notification helps or harms the patient.

These values are not poorly specified requirements that can be reconciled through better engineering. They reflect fundamentally different conceptions of what the system should achieve and whom it should prioritize. Optimizing for medical efficacy (aggressive intervention) directly conflicts with patient autonomy (minimal intervention). Privacy protection (no data retention) conflicts with resource efficiency (learning from interactions). Legal compliance (mandatory reporting) may conflict with clinical efficacy (therapeutic relationship based on trust).

No algorithm determines which value should dominate. Different stakeholders hold legitimately different positions: clinicians may prioritize efficacy, teenagers may prioritize autonomy, lawyers may prioritize compliance, and budget officers may prioritize efficiency. The technical team must facilitate stakeholder deliberation to determine which trade-offs are acceptable in this specific context, a fundamentally normative decision that precedes and constrains technical optimization.

Systems lesson: Responsible AI trade-offs are system requirements, not after-the-fact policy notes. The deployment architecture must encode the stakeholder choice before optimization begins.

What constitutes a fair outcome for one stakeholder may be perceived as inequitable by another. Similarly, decisions that prioritize accuracy or efficiency may conflict with goals such as transparency, individual autonomy, or harm reduction. These tensions are not incidental; they are structural. They reflect the pluralistic nature of the societies in which machine learning systems are embedded and the institutional settings in which they are deployed.

Fairness is a particularly prominent site of value conflict. Fairness can be formalized in multiple, often incompatible ways. A model that satisfies demographic parity may violate equalized odds; a model that prioritizes individual fairness may undermine group-level parity. Choosing among these definitions is not purely a technical decision but a normative one, informed by domain context, historical patterns of discrimination, and the perspectives of those affected by model outcomes. In practice, multiple stakeholders, including engineers, users, auditors, and regulators, may hold conflicting views on which definitions are most appropriate and why.

Value conflicts extend beyond fairness alone. Conflicts also arise between interpretability and predictive performance, privacy and personalization, or short-term utility and long-term consequences. These trade-offs manifest differently depending on the systems deployment architecture, revealing how deeply value conflicts are tied to the design and operation of ML systems.

Consider a voice-based assistant deployed on a mobile device. To enhance personalization, the system may learn user preferences locally, without sending raw data to the cloud. This design improves privacy and reduces latency, but it may also lead to performance disparities if users with underrepresented usage patterns receive less accurate or responsive predictions. One way to improve fairness would be to centralize updates using group-level statistics, but doing so introduces new privacy risks and may violate user expectations around local data handling. Here, the design must navigate among valid but competing values: privacy, fairness, and personalization.

In cloud-based deployments, such as credit scoring platforms or recommendation engines, tensions often arise between transparency and proprietary protection. End users or regulators may demand clear explanations of why a decision was made, particularly in situations with significant consequences, but the models in use may rely on complex ensembles or proprietary training data. Revealing these internals may be commercially sensitive or technically infeasible. In such cases, the system must reconcile competing pressures for institutional accountability and business confidentiality.

In edge systems, such as home security cameras or autonomous drones, resource constraints often dictate model selection and update frequency. Prioritizing low latency and energy efficiency may require deploying compressed or quantized models that are less robust to distribution shift or adversarial perturbations. More resilient models could improve safety, but they may exceed the system's memory budget or violate power constraints. Here, safety, efficiency, and maintainability must be balanced under hardware-imposed trade-offs. Efficiency techniques and optimization methods are essential for implementing responsible AI in resource-constrained environments.

On TinyML platforms, where models are deployed to microcontrollers with no persistent connectivity, trade-offs are even more pronounced. A system may be optimized for static performance on a fixed dataset, but unable to incorporate new fairness constraints, retrain on updated inputs, or generate explanations once deployed. Hardware constraints fundamentally shape what responsible AI practices are feasible on resource-limited devices. The value conflict extends beyond what the model optimizes to encompass what the system can support postdeployment.

The recurring systems constraint is **Normative pluralism**, not an abstract philosophical challenge. Technical approaches such as multi-objective optimization, constrained training, and fairness-aware evaluation can help surface and formalize trade-offs, but they do not eliminate the need for judgment. Decisions about whose values to represent, which harms to mitigate, and how to balance competing objectives cannot be made algorithmically. They require deliberation, stakeholder input, and governance structures that extend beyond the model itself.

Participatory and value-sensitive design methodologies offer potential paths forward. Rather than treating values as parameters to be optimized after deployment, these approaches seek to engage stakeholders during the requirements phase, define ethical trade-offs explicitly, and trace how they are instantiated in system architecture. While no design process can satisfy all values simultaneously, systems that are transparent about their trade-offs and open to revision are better positioned to sustain trust and accountability over time.

Machine learning systems are not neutral tools. They embed and enact value judgments, whether explicitly specified or implicitly assumed. A commitment to responsible AI requires acknowledging this fact and building systems that reflect and respond to the ethical and social pluralism of their operational contexts.

16.7.4 Transparency and contestability

Value conflicts become governable only when stakeholders can understand and challenge system decisions. Transparency allows users, developers, auditors, and regulators to understand how a system functions, assess its limitations, and identify sources of harm. Yet transparency alone is not sufficient. In high stakes domains, individuals and institutions must not only understand system behavior; they must also be able to challenge, correct, or reverse it when necessary. This capacity for **contestability**, which refers to the ability to interrogate and contest a system's decisions, is an important feature of accountability.

Transparency in machine learning systems typically focuses on disclosure: revealing how models are trained, what data they rely on, what assumptions are embedded in their design, and what known limitations affect their use. Documentation tools such as model cards and datasheets for datasets support this goal by formalizing system metadata in a structured, reproducible format. These resources can improve governance, support compliance, and inform user expectations. However, transparency as disclosure does not guarantee meaningful control. Even when technical details are available, users may lack institutional support, interface tools, or procedural access to contest a decision that adversely affects them.

To move from transparency to contestability, machine learning systems must be designed with mechanisms for explanation, recourse, and feedback:

- **Explanation**: The system provides understandable reasons for its outputs, tailored to the needs and context of the person receiving them.
- **Recourse**: Individuals can alter their circumstances and receive a different outcome.
- **Feedback**: Users can report errors, dispute outcomes, or signal concerns, and those signals can be incorporated into system updates or oversight processes.

These mechanisms are often lacking in practice, particularly in systems deployed at scale or embedded in low-resource devices. For example, in mobile loan application systems, users may receive a rejection without explanation and have no opportunity to provide additional information or appeal the decision. The lack of transparency at the interface level, even if documentation exists elsewhere, makes the system effectively unchallengeable. Similarly, a predictive model deployed in a clinical setting may generate a risk score that guides treatment decisions without surfacing the underlying reasoning to the physician. If the model underperforms for a specific patient subgroup, and this behavior is not observable or contestable, the result may be unintentional harm that cannot be easily diagnosed or corrected.

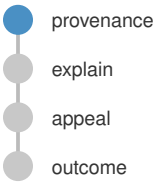
From a systems perspective, enabling contestability requires coordination across technical and institutional components. Models must expose sufficient information to support explanation. Interfaces must surface this information in a usable and timely way. Organizational processes must be in place to review feedback, respond to appeals, and update system behavior. Logging and auditing infrastructure must track not only model outputs, but user interventions and override decisions. In some cases, technical safeguards, including human-in-the-loop overrides and decision abstention thresholds, may also serve contestability by ensuring that ambiguous or high-risk decisions defer to human judgment.

Implementing contestability imposes concrete infrastructure costs that scale with system throughput and complexity. Storing the necessary metadata to reconstruct a decision—input features, model version, and decision thresholds—requires persistent storage whose footprint depends on feature dimensionality, explanation payloads, and retention windows. Generating on-demand explanations using Shapley values or counterfactuals can add enough latency that contested decisions often need asynchronous processing queues to preserve serving SLAs. Maintaining immutable audit trails for high-risk systems under frameworks like the EU AI Act requires storage, provenance, and oversight capacity to be budgeted as part of the inference fleet rather than as a separate policy artifact (European Parliament and Council of the European Union 2024).

Architecturally, contestability requires a specialized **contestability stack**, a design pattern analogous to distributed tracing in microservices. This stack must orchestrate four coupled components:

- **Decision provenance**: The system cryptographically links a specific output to the exact model binary and input vector used.
- **Explanation generation**: A high-latency service triggers resource-intensive interpretation methods only upon user request.
- **Appeal routing**: The workflow directs contested decisions to human reviewers with appropriate domain expertise.
- **Outcome tracking**: The system closes the loop by recording whether the appeal overturned the machine decision.

Without this integrated infrastructure, debugging algorithmic errors becomes impossible, as the system lacks the granular lineage required to trace a specific user complaint back to the offending weights or training data.



Contestability is a production stack, not just a policy word.

The degree of contestability that is feasible varies by deployment context. In centralized cloud platforms, it may be possible to offer full explanation APIs, user dashboards, and appeal workflows. In contrast, in edge and TinyML deployments, contestability may be limited to logging and periodic updates based on batch-synchronized feedback. In all cases, the design of machine learning systems must acknowledge that transparency is not simply a matter of technical disclosure. It is a structural property of systems that determines whether users and institutions can meaningfully question, correct, and govern the behavior of automated decision-making.

16.7.5 Institutional embedding of responsibility

Transparency and contestability mechanisms fail without institutional support. Machine learning systems do not operate in isolation: their development, deployment, and ongoing management are embedded within environments that include technical teams, legal departments, product owners, compliance officers, and external stakeholders. Responsibility in such systems is not the property of a single actor or component; it is distributed across roles, workflows, and governance processes. Designing for responsible AI therefore requires attention to the institutional settings in which these systems are built and used.

Distributing responsibility across roles introduces both opportunities and challenges. On the one hand, the involvement of multiple stakeholders provides checks and balances that can help prevent harmful outcomes. On the other hand, the diffusion of responsibility can lead to accountability gaps, where no individual or team has clear authority or incentive to intervene when problems arise. When harm occurs, it may be unclear whether the fault lies with the data pipeline, the model architecture, the deployment configuration, the user interface, or the surrounding organizational context.

One illustrative case is Google Flu Trends, a widely cited example of failure due to institutional misalignment. The system, which attempted to predict flu outbreaks from search data, initially performed well but gradually diverged from reality due to changes in user behavior and shifts in the data distribution. These issues went uncorrected for years, in part because there were no established processes for system validation, external auditing, or escalation when model performance declined. The failure was not due to a single technical flaw, but to the absence of an institutional framework that could respond to drift, uncertainty, and feedback from outside the development team.

Operational rigor comes with a measurable cost. Public responsible-AI standards and internal review processes illustrate that governance can add release latency through impact assessments, model reviews, red-team exercises, and documentation gates. The durable systems point is that review capacity becomes part of deployment planning: if every high-risk launch needs specialized review, the review queue becomes a bottleneck just like security review or capacity approval. The responsibility overhead is thus not a sunk cost but an insurance premium against the far higher cost of retracting a biased model or patching a live exploit in a global fleet.

Embedding responsibility institutionally requires more than assigning accountability. It requires the design of processes, tools, and incentives that allow responsible action. Technical infrastructure such as versioned model registries, model cards, and audit logs must be coupled with organizational structures such as ethics review boards, model risk committees, and red-teaming[37] procedures.

37 **Red Teaming**: From Cold War military simulations where the "Red Team" acted as the Soviet adversary to probe US defenses. In Responsible AI, red teaming is the adversarial audit phase: specialized teams (hackers, linguists, ethicists) deliberately probe models for jailbreaks, bias, or toxic outputs before deployment. This discovery process identifies the long-tail risks that standard unit tests cannot catch.

These mechanisms ensure that technical insights are actionable, that feedback is integrated across teams, and that concerns raised by users, developers, or regulators are addressed systematically rather than ad hoc.

The level of institutional support required varies across deployment contexts. In large-scale cloud platforms, governance structures may include internal accountability audits, compliance workflows, and dedicated teams responsible for monitoring system behavior. In smaller-scale deployments, including edge or mobile systems embedded in healthcare devices or public infrastructure, governance may rely on cross-functional engineering practices and external certification or regulation. In TinyML deployments, where connectivity and observability are limited, institutional responsibility may be exercised through upstream controls such as safety-important validation, embedded security constraints, and lifecycle tracking of deployed firmware.

In all cases, responsible machine learning requires coordination between technical and institutional systems. This coordination must extend across the entire model lifecycle, from initial data acquisition and model training to deployment, monitoring, update, and eventual decommissioning. It must also incorporate external actors, including domain experts, civil society organizations, and regulatory authorities, to ensure that responsibility is exercised not only within the development team but across the broader ecosystem in which machine learning systems operate.

The system-level thesis thus extends past the serving stack into the organization: responsibility is a dynamic property of how systems are governed, maintained, and contested over time, owned by no single model or team. Embedding it within institutions, by means of policy, infrastructure, and accountability mechanisms, is what aligns machine learning systems with the social values and operational realities they are meant to serve.

Taken together, institutional responsibility, value conflicts, feedback loops, human-AI collaboration, contestability, and computational equity show why the technical foundations from section 16.4 through section 16.6 cannot ensure responsible AI on their own. Those technical foundations remain necessary, but they need organizational authority, operational telemetry, and escalation paths to survive contact with deployed systems. Resource constraints also determine who can develop, deploy, and benefit from responsible AI capabilities, so implementation choices are not merely local engineering details. Otherwise, a fairness metric with no owner, an explanation with no appeal process, or a monitoring signal with no remediation budget becomes evidence without action. Once an AI system changes the environment it operates in, engineering teams must turn responsible-AI principles into corporate routines under deadline pressure, resource limits, and competing incentives.

Checkpoint 16.3: When correct code fails in deployment

This section showed why mathematically sound, fair, explainable models still fail in human institutions: feedback loops that reshape the data, automation bias and institutional deference, normative pluralism among stakeholders, and contestability as a production stack rather than a policy word.

Diagnosing the human-system interaction

- ☐ **Alert adoption failure**: Identify the sociotechnical failure mode in a highly accurate sepsis model that does not improve mortality because clinicians ignore its alerts, and name the interface change that would counter it.
- ☐ **Paradox of reliability**: Explain why raising model accuracy from 90 to 99 percent can lower system-level safety, and why the response is to add friction rather than remove the human.

Reasoning about loops and stakeholders

- ☐ **Feedback loops**: Trace how a feedback loop makes a model's original prediction self-fulfilling in a predictive-policing or recommender system, and explain why a static test set fails to detect it.

☐ **Normative pluralism**: Justify why no algorithm resolves conflicts among medical efficacy, patient autonomy, privacy, and legal compliance in the adolescent mental-health chatbot, and identify what the engineering team must do before optimization begins.

16.8 Implementation Challenges and AI Safety

The data science team wants to hold back deployment for a month to conduct rigorous fairness audits on a new generative model. The executive team, watching a competitor launch a similar feature, demands the model be deployed by Friday. This is the implementation reality of responsible AI. It is rarely a question of whether engineers know how to test for bias; it is a question of whether the organizational structure, budget, and business priorities allow them the time and authority to actually do it.

The scenario exposes the implementation gap between technical capability and operational authority. Responsible AI methods provide necessary tools, but their effectiveness depends on organizational structures, data infrastructure, evaluation processes, and sustained commitment that extends far beyond algorithm development. Systems maintain responsible behavior over time only when those implementation supports survive deadline pressure, product incentives, and postdeployment drift.

The practical barriers to embedding responsible AI in production ML systems follow the classical People-Process-Technology framework:

- **People challenges**: These include organizational structures, role definitions, incentive alignment, and stakeholder coordination that determine whether responsible AI principles translate into sustained organizational behavior.
- **Process challenges**: These include standardization gaps, lifecycle maintenance procedures, competing optimization objectives, and evaluation methodologies that affect how responsible AI practices integrate with development workflows.
- **Technology challenges**: These include data quality constraints, computational resource limitations, scalability bottlenecks, and infrastructure gaps that determine whether responsible AI techniques can operate effectively at production scale.

These categories matter because responsible AI fails when organizational incentives, development workflows, and production infrastructure cannot support the same obligation.

The point of the framework is coordination, not classification. Responsible AI fails when people, process, and technology are handled independently: a fairness metric with no owner does not trigger remediation, a governance process without telemetry cannot see drift, and a privacy technique without data lineage cannot honor deletion requests. Effective implementation requires systems-level strategies that embed responsibility into the architecture, infrastructure, and workflows of machine learning deployment across all three dimensions simultaneously.

16.8.1 Organizational structures and incentives

The implementation of responsible machine learning is shaped not only by technical feasibility but by the organizational context in which systems are developed and deployed. Within companies, research labs, and public institutions, responsibility must be translated into concrete roles, workflows, and incentives. In practice, however, organizational structures often fragment responsibility, making it difficult to coordinate ethical objectives across engineering, product, legal, and operational teams.

Responsible AI requires sustained investment in practices such as subgroup performance evaluation, explainability analysis, adversarial robustness testing, and the integration of privacy-preserving techniques like differential privacy or federated training. These activities can be time-consuming and resource-intensive, yet they often fall outside the formal performance metrics used to evaluate team productivity. For example, teams may be incentivized to ship features quickly or meet performance benchmarks, even when doing so undermines fairness or overlooks potential harms. When ethical diligence is treated as a discretionary task, instead of being an integrated component of the system lifecycle, it becomes vulnerable to deprioritization under deadline pressure or organizational churn.

Responsibility is further complicated by ambiguity over ownership. In many organizations, no single team is responsible for ensuring that a system behaves ethically over time. Model performance

may be owned by one team, user experience by another, data infrastructure by a third, and compliance by a fourth. When issues arise, including disparate impact in predictions or insufficient explanation quality, there may be no clear protocol for identifying root causes or coordinating mitigation. As a result, concerns raised by developers, users, or auditors may go unaddressed, not because of malicious intent, but due to lack of process and cross-functional alignment.

Establishing effective organizational structures for responsible AI requires more than policy declarations. It demands operational mechanisms: designated roles with responsibility for ethical oversight, documented escalation pathways, accountability for postdeployment monitoring, and incentives that reward teams for ethical foresight and system maintainability. In some organizations, this may take the form of Responsible AI committees, cross-functional review boards, or model risk teams that work alongside developers throughout the model lifecycle. In others, domain experts or user advocates may be embedded into product teams to anticipate downstream impacts and evaluate value trade-offs in context.

The responsibility for ethical system behavior is distributed across multiple constituencies, including industry, academia, civil society, and government. Figure 16.13 maps this distribution across nested layers of accountability, from individual teams implementing technical practices through organizational safety culture to industry-wide certification and government regulation (Shneiderman 2022, 2020). Within organizations, this distribution must be mirrored by mechanisms that connect technical design with strategic oversight and operational control. Without these linkages, responsibility becomes diffuse, and well-intentioned efforts may be undermined by systemic misalignment.

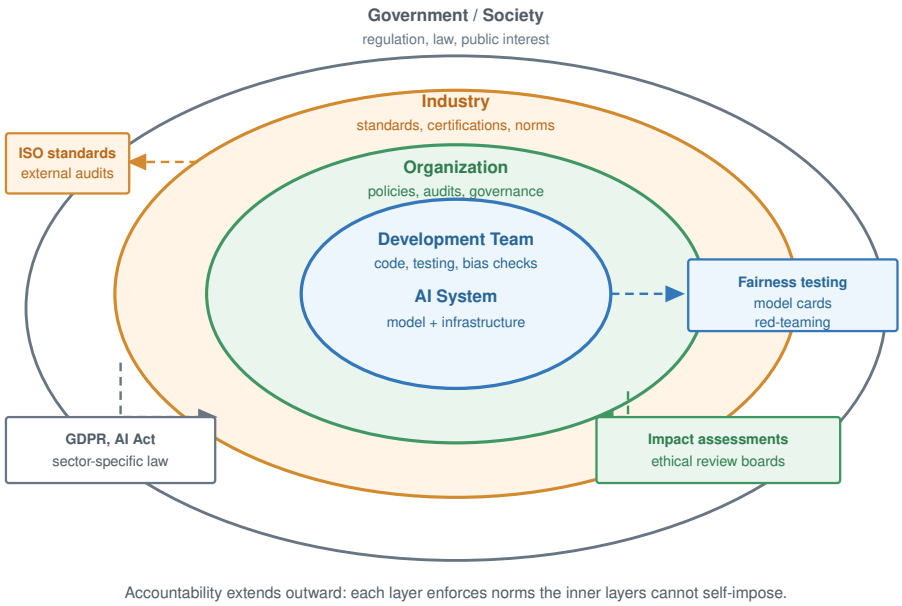


Figure 16.13: Layers of Responsibility: Effective human-centered AI implementation requires shared accountability across nested layers. From the core Development Team outward through the Organization, Industry, and Government/Society, each surrounding ring enforces norms the inner layers cannot self-impose. External anchors, including ISO AI management standards, fairness testing, model cards, red-teaming, impact assessments, GDPR, the EU AI Act, and sector-specific law, bind these layers to practice (International Organization for Standardization and International Electrotechnical Commission 2023b; European Parliament and Council of the European Union 2016; European Parliament and Council of the European Union 2024).

Responsible AI is not merely a question of technical excellence or regulatory compliance. It is a systems-level challenge that requires aligning ethical objectives with the institutional structures through which machine learning systems are designed, deployed, and maintained. Creating and sustaining these structures is important for ensuring that responsibility is embedded not only in the model, but in the organization that governs its use. The same ownership problem becomes

concrete at the data layer: teams can know how to audit a dataset and still lack the authority, access, or infrastructure needed to change it.

16.8.2 Data constraints and quality gaps

Improving data pipelines remains one of the most difficult implementation challenges in practice despite broad recognition that data quality is important for responsible machine learning. Developers and researchers often understand the importance of representative data, accurate labeling, and mitigation of historical bias. Yet even when intentions are clear, structural and organizational barriers frequently prevent meaningful intervention. Responsibility for data is often distributed across teams, governed by legacy systems, or embedded in broader institutional processes that are difficult to change.

Data engineering principles, including data validation, schema management, versioning, lineage tracking, and quality monitoring, provide the technical foundation for addressing these challenges. However, applying these principles to responsible AI introduces additional complexity: fairness requires assessing representativeness across demographic groups, bias mitigation demands understanding historical data collection practices, and privacy preservation constrains which validation techniques are permissible. The organizational challenges described here reflect the gap between having robust data engineering infrastructure and using it effectively to support responsible AI objectives.

Subgroup imbalance, label ambiguity, and distribution shift, each of which affect generalization and performance across domains, are well-established concerns in responsible ML. These issues often manifest in the form of poor calibration, out-of-distribution failures, or demographic disparities in evaluation metrics. However, addressing them in real-world settings requires more than technical knowledge. It requires access to relevant data, institutional support for remediation, and sufficient time and resources to iterate on the dataset itself. In many machine learning pipelines, once the data is collected and the training set defined, the data pipeline becomes effectively frozen. Teams may lack both the authority and the infrastructure to modify or extend the dataset midstream, even if performance disparities are discovered. Even in automated data pipelines with validation and feature stores, retroactively correcting training distributions remains difficult once dataset versioning and data lineage have been locked into production.

In domains like healthcare, education, and social services, these challenges are especially pronounced. Data acquisition may be subject to legal constraints, privacy regulations, or cross-organizational coordination. For example, a team developing a triage model may discover that their training data underrepresents patients from smaller or rural hospitals. Correcting this imbalance would require negotiating data access with external partners, aligning on feature standards, and resolving inconsistencies in labeling practices. The logistical and operational costs can be prohibitive even when all parties agree on the need for improvement.

Efforts to collect more representative data may also run into ethical and political concerns. In some cases, additional data collection could expose marginalized populations to new risks. This paradox of exposure, in which the individuals most harmed by exclusion are also those most vulnerable to misuse, complicates efforts to improve fairness through dataset expansion. For example, gathering more data on nonbinary individuals to support fairness in gender-sensitive applications may improve model coverage, but it also raises serious concerns around consent, identifiability, and downstream use. Teams must navigate these tensions carefully, often without clear institutional guidance.



Napkin Math 16.5: The representation tax

Problem: A medical imaging model trained on data from 5 hospitals achieves 94 percent accuracy overall but only 78 percent on underrepresented populations (rural patients, elderly patients, patients with darker skin tones). Closing this gap requires representative data from 50+ hospitals across diverse geographies, demographics, and equipment types.

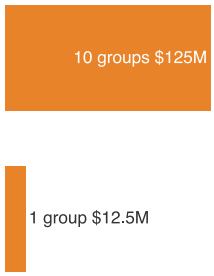
Math:

1. **Data acquisition cost**: $50–$200 per labeled medical image, with 100,000 images needed per underrepresented subgroup.

2. **Representation-tax scenario**: For 10 subgroups, at the midpoint of the cost range, 10 subgroups times 100,000 images per subgroup times $125 per image yields $125M in data acquisition alone.

3. **Data harmonization**: Data harmonization across different scanners, protocols, and labeling conventions adds 30 percent–50 percent overhead, bringing the total to $162.5M–$187.5M.

Systems insight: The representation tax makes equitable performance materially more expensive than high aggregate accuracy on the majority population. The populations most harmed by biased models are the most expensive to represent in training data. Data budgets must be allocated not by aggregate utility but by subgroup coverage gaps—a fundamentally different optimization target than maximizing overall accuracy.



Representation cost scales with subgroup coverage, not just dataset size.

Upstream biases in data collection systems can persist unchecked even when data is plentiful. Many organizations rely on third-party data vendors, external APIs, or operational databases that were not designed with fairness or interpretability in mind. For instance, Electronic Health Records, which are commonly used in clinical machine learning, often reflect systemic disparities in care, as well as documentation habits that encode racial or socioeconomic bias (Himmelstein et al. 2022). Teams working downstream may have little visibility into how these records were created, and few levers for addressing embedded harms.

Improving dataset quality is often not the responsibility of any one team. Data pipelines may be maintained by infrastructure or analytics groups that operate independently of the ML engineering or model evaluation teams. This organizational fragmentation makes it difficult to coordinate data audits, track provenance, or implement feedback loops that connect model behavior to underlying data issues. In practice, responsibility for dataset quality tends to fall through the cracks, recognized as important, but rarely prioritized or resourced.

Addressing these challenges requires long-term investment in infrastructure, workflows, and cross-functional communication. Technical tools such as data validation, automated audits, and dataset documentation frameworks (for example, model cards, datasheets, or the Data Nutrition Project[38]) can help, but only when they are embedded within teams that have the mandate and support to act on their findings.

Improving data quality is therefore fundamentally a question of how responsibility for data is assigned, shared, and sustained across the system lifecycle, not merely a matter of better tooling. Once that responsibility is in place, teams still have to decide which responsible-AI objective should dominate when valid goals conflict.

38 **Data Nutrition Project**: A nonprofit initiative that develops standardized dataset documentation modeled on nutrition labels, summarizing provenance, composition, distribution, and known issues for datasets used in ML pipelines. The project's labels make dataset risks legible to downstream engineers who lack access to the original collection process—an essential bridge for the organizational fragmentation described here.

16.8.3 Balancing competing objectives

Machine learning system design is often framed as a process of optimization, improving accuracy, reducing loss, or maximizing utility. Yet in responsible ML practice, optimization must be balanced against a range of competing objectives, including fairness, interpretability, robustness, privacy, and resource efficiency. These objectives are not always aligned, and improvements in one dimension may entail trade-offs in another. While these tensions are well understood in theory, managing them in real-world systems is a persistent and unresolved challenge.

Consider the trade-off between model accuracy and interpretability. In many cases, more interpretable models, including shallow decision trees and linear models, achieve lower predictive performance than complex ensemble methods or deep neural networks. In low-stakes applications, this trade-off may be acceptable, or even preferred. In high-stakes domains such as healthcare or finance, however, where decisions affect individuals' well-being or access to opportunity, teams are often caught between the demand for performance and the need for transparent reasoning. Even when interpretability is prioritized during development, it may be overridden at deployment in favor of marginal gains in model accuracy.

Similar tensions emerge between personalization and fairness. A recommendation system trained to maximize user engagement may personalize aggressively, using fine-grained behavioral data to

tailor outputs to individual users. While this approach can improve satisfaction for some users, it may entrench disparities across demographic groups, particularly if personalization draws on features correlated with race, gender, or socioeconomic status. Adding fairness constraints may reduce disparities at the group level, but at the cost of reducing perceived personalization for some users. These effects are often difficult to measure, and even more difficult to explain to product teams under pressure to optimize engagement metrics.

Privacy introduces another set of constraints. Techniques such as differential privacy, federated learning, or local data minimization (Biega et al. 2020) can meaningfully reduce privacy risks. They also introduce noise, limit model capacity, or reduce access to training data. In centralized systems, these costs may be absorbed through infrastructure scaling or hybrid training architectures. In edge or TinyML deployments, however, the trade-offs are more acute. A wearable device tasked with local inference must often balance model complexity, energy consumption, latency, and privacy guarantees simultaneously. Supporting one constraint typically weakens another, forcing system designers to prioritize among equally important goals. These tensions are further amplified by deployment-specific design decisions such as quantization levels, activation clipping, or compression strategies that affect how effectively models can support multiple objectives at once.

The trade-offs are not purely technical; they reflect deeper normative judgments about what a system is designed to achieve and for whom, as explored in detail in section 16.7.3. Responsible ML development requires making these judgments explicit, evaluating them in context, and subjecting them to stakeholder input and institutional oversight.

What makes this challenge particularly difficult in implementation is that these competing objectives are rarely owned by a single team or function. Performance may be optimized by the modeling team, fairness monitored by a responsible AI group, and privacy handled by legal or compliance departments. Without deliberate coordination, system-level trade-offs can be made implicitly, piecemeal, or without visibility into long-term consequences. Over time, the result may be a model that appears well-behaved in isolation but fails to meet its ethical goals when embedded in production infrastructure.

Balancing competing objectives requires not only technical fluency but a commitment to transparency, deliberation, and alignment across teams. Systems must be designed to surface trade-offs rather than obscure them, to make room for constraint-aware development rather than pursue narrow optimization. In practice, this may require redefining what "success" looks like, not as performance on a single metric, but as sustained alignment between system behavior and its intended role in a broader social or operational context.

Across these first three challenges (organizational structures, data quality, and competing objectives), a pattern emerges: responsible AI failure rarely stems from technical ignorance. Teams understand fairness metrics, privacy techniques, and bias mitigation methods. Instead, failure occurs at the intersection of organizational fragmentation that distributes responsibility without accountability, data constraints that create technical barriers even with clear intentions, and competing objectives that force normative trade-offs disguised as technical problems. When modeling teams optimize performance, compliance teams address privacy, and product teams prioritize engagement independently, system-level ethical behavior emerges by accident rather than design. These are fundamentally sociotechnical governance problems requiring clear ownership structures that span organizational boundaries, data infrastructure designed for ethical auditing, and deliberative processes for making value trade-offs explicit. These challenges become even more acute when systems must maintain responsible behavior at scale over time.

16.8.4 Scalability and maintenance

Responsible machine learning practices are often introduced during the early phases of model development: fairness audits are conducted during initial evaluation, interpretability methods are applied during model selection, and privacy-preserving techniques are considered during training. However, as systems transition from research prototypes to production deployments, these practices frequently degrade or disappear. The gap between what is possible in principle and what is sustainable in production is a core implementation challenge for responsible AI.

Many responsible AI interventions are not designed with scalability in mind. Fairness checks may be performed on a static dataset, but not integrated into ongoing data ingestion pipelines. Explana-

tion methods may be developed using development-time tools but never translated into deployable user-facing interfaces. Privacy constraints may be enforced during training, but overlooked during postdeployment monitoring or model updates. In each case, what begins as a responsible design intention fails to persist across system scaling and lifecycle changes.

Production environments introduce new pressures that reshape system priorities. Models must operate across diverse hardware configurations, interface with evolving APIs, serve millions of users with low latency, and maintain availability under operational stress. For instance, maintaining consistent behavior across CPU, GPU, and edge accelerators requires tight integration between framework abstractions, runtime schedulers, and hardware-specific compilers. These constraints demand continuous adaptation and rapid iteration, often deprioritizing activities that are difficult to automate or measure. Responsible AI practices, especially those that involve human review, stakeholder consultation, or posthoc evaluation, may not be easily incorporated into fast-paced DevOps[39] pipelines.

Maintenance introduces further complexity. Machine learning systems are rarely static. New data is ingested, retraining is performed, features are deprecated or added, and usage patterns shift over time. In the absence of rigorous version control, changelogs, and impact assessments, it can be difficult to trace how system behavior evolves or whether responsibility-related properties such as fairness or robustness are being preserved. Organizational turnover and team restructuring can erode institutional memory. Teams responsible for maintaining a deployed model may not be the ones who originally developed or audited it, leading to unintentional misalignment between system goals and deployed behavior. These issues are especially acute in continual or streaming learning scenarios, where concept drift and shifting data distributions demand active monitoring and real-time updates.

These challenges are magnified in multi-model systems and cross-platform deployments. A recommendation engine may consist of dozens of interacting models, each optimized for a different subtask or user segment. A voice assistant deployed across mobile and edge environments may maintain different versions of the same model, tuned to local hardware constraints. Coordinating updates, ensuring consistency, and sustaining responsible behavior in such distributed systems requires infrastructure that tracks not only code and data, but also values and constraints.

Addressing scalability and maintenance challenges requires treating responsible AI as a lifecycle property, not a one-time evaluation. This means embedding audit hooks, metadata tracking, and monitoring protocols into system infrastructure. It also means creating documentation that persists across team transitions, defining accountability structures that survive project handoffs, and ensuring that system updates do not inadvertently erase hard-won improvements in fairness, transparency, or safety. While such practices can be difficult to implement retroactively, they can be integrated into system design from the outset through responsible-by-default tooling and workflows.

Responsibility must scale with the system. Machine learning models deployed in real-world environments must not only meet ethical standards at launch but also continue to do so as they grow in complexity, user reach, and operational scope. Achieving this requires sustained organizational investment and architectural planning, not merely technical correctness at a single point in time.

16.8.5 Standardization and evaluation gaps

While the field of responsible machine learning has produced a wide range of tools, metrics, and evaluation frameworks, there is still little consensus on how to systematically assess whether a system is responsible in practice. Many teams recognize the importance of fairness, privacy, interpretability, and robustness, yet they often struggle to translate these principles into consistent, measurable standards. Benchmarking methodologies provide valuable frameworks for standardized evaluation, though adapting these approaches to responsible AI metrics remains an active area of development. The lack of formalized evaluation criteria, combined with the fragmentation of tools and frameworks, poses a significant barrier to implementing responsible AI at scale.

The fragmentation is evident both across and within institutions. Academic research frequently introduces new metrics for fairness or robustness that are difficult to reproduce outside experimental settings. Industrial teams, by contrast, must prioritize metrics that integrate cleanly with production infrastructure, are interpretable by nonspecialists, and can be monitored over time. As a result, practices developed in one context may not transfer well to another, and performance comparisons

39 **DevOps for ML (MLOps)**: ML CI/CD pipelines must handle data versioning, training reproducibility, and A/B testing of algorithm changes beyond traditional software concerns. High-velocity ML organizations may deploy model or configuration changes far more often than responsible-AI reviews can be completed manually, creating a velocity gap: deployment cycles measured in hours compete against ethical validation requiring days or weeks. This tension explains why responsible AI commitments present at the prototype stage can be deprioritized as systems scale.

across systems may be unreliable or misleading. For instance, a model evaluated for fairness on one benchmark dataset using demographic parity may not meet the requirements of equalized odds in another domain or jurisdiction. Without shared standards, these evaluations remain ad hoc, making it difficult to establish confidence in a system's responsible behavior across contexts.

Responsible AI evaluation also suffers from a mismatch between the unit of analysis, which is frequently the individual model or batch job, and the level of deployment, which includes end-to-end system components such as data ingestion pipelines, feature transformations, inference APIs, caching layers, and human-in-the-loop workflows. A system that appears fair or interpretable in isolation may fail to uphold those properties once integrated into a broader application. Tools that support holistic, system-level evaluation remain underdeveloped, and there is little guidance on how to assess responsibility across interacting components in production ML stacks.

Further complicating matters is the lack of lifecycle-aware metrics. Most evaluation tools are applied at a single point in time, often just before deployment. Yet responsible AI properties such as fairness and robustness are dynamic. They depend on how data distributions evolve, how models are updated, and how users interact with the system. Without continuous or periodic evaluation, it is difficult to determine whether a system remains aligned with its intended ethical goals after deployment. Postdeployment monitoring tools exist, but they are rarely integrated with the development-time metrics used to assess initial model quality. This disconnect makes it hard to detect drift in ethical performance, or to trace observed harms back to their upstream sources.

Tool fragmentation further contributes to these challenges. Responsible AI tooling is often distributed across disconnected packages, dashboards, or internal systems, each designed for a specific task or metric. A team may use one tool for explainability, another for bias detection, and a third for compliance reporting, with no unified interface for reasoning about system-level trade-offs. The lack of interoperability hinders collaboration between teams, complicates documentation, and increases the risk that important evaluations will be skipped or performed inconsistently. These challenges are compounded by missing hooks for metadata propagation or event logging across components like feature stores, inference gateways, and model registries.

The failed-transfer case introduced above is the concrete shape of the problem: a model certified fair under demographic parity on one benchmark can fail equalized odds in a new domain or jurisdiction, so the certification does not travel with the model. Closing that gap requires evaluation criteria that are measurable and auditable across domains, applied to full system pipelines rather than isolated models, and re-run as a recurring lifecycle activity so a passing result at launch does not silently expire as distributions drift. Until those practices are shared rather than ad hoc, responsible AI remains described in principles but difficult to verify in practice.

Responsible AI cannot be achieved through isolated interventions or static compliance checks. It requires architectural planning, infrastructure support, and institutional processes that sustain ethical goals across the system lifecycle. As ML systems scale, diversify, and embed themselves into sensitive domains, the ability to enforce properties like fairness, robustness, and privacy must be supported not only at model selection time, but across retraining, quantization, serving, and monitoring stages. Without persistent oversight, responsible practices degrade as systems evolve, especially when tooling, metrics, and documentation are not designed to track and preserve them through deployment and beyond.

Meeting this challenge will require greater standardization, deeper integration of responsibility-aware practices into CI/CD pipelines, and long-term investment in system infrastructure that supports ethical foresight. The goal is not to perfect ethical decision-making in code, but to make responsibility an operational property, traceable, testable, and aligned with the constraints and affordances of machine learning systems at scale.

16.8.6 Implementation decision framework

Given these implementation challenges, practitioners need systematic approaches to prioritize responsible AI principles based on deployment context and stakeholder needs. The same principle can demand different engineering treatment across high stakes individual decisions, safety-critical systems, privacy-sensitive applications, large-scale consumer systems, resource-constrained deployments, and research environments.

Four decision heuristics guide these trade-offs in practice:

- When multiple principles conflict, engage stakeholders to determine which harms are most severe. The mental health chatbot example examined in section 16.7.3 showed such conflicts require deliberation, not algorithmic resolution.
- When computational budgets are constrained, prioritize principles by risk. High-stakes decisions demand fairness/explainability even at significant cost. Low-stakes applications can use lightweight methods.
- When deployment context changes, re-evaluate principle priorities. A cloud model moved to edge loses centralized monitoring capability; compensate with predeployment validation and local safeguards.
- When stakeholder values differ, document trade-offs explicitly and create contestability mechanisms allowing affected users to challenge decisions.

Table 16.6 provides a practitioner decision framework that maps these deployment contexts to primary principles, implementation priorities, and acceptable trade-offs so the decision can remain context-sensitive rather than universal.

Table 16.6: Practitioner Decision Framework: Prioritizing responsible AI principles based on deployment context, showing primary principles, implementation priorities, and acceptable trade-offs for different system types. This framework guides practitioners in making context-appropriate decisions when principles conflict or resources are constrained.

Deployment Context	Primary Principles	Implementation Priority	Acceptable Trade-offs
High-Stakes Individual Decisions (healthcare diagnosis, credit/loans, criminal justice, employment)	Fairness, Explainability, Accountability	Mandatory fairness metrics across protected groups; explainability for negative outcomes; human oversight for edge cases	Accept a measured accuracy budget for interpretability and a bounded latency explanations; higher computational costs
Safety-Critical Systems (autonomous vehicles, medical devices, industrial control)	Safety, Robustness, Accountability	Certified adversarial defenses; formal validation; failsafe mechanisms; comprehensive logging	Accept significant validation and adversarial-training overhead; conservative confidence thresholds; redundant inference
Privacy-Sensitive Applications (health records, financial data, personal communications)	Privacy, Security, Transparency	Differential privacy ($\varepsilon \leq 1.0$); local processing; data minimization; user consent mechanisms	Accept privacy-utility trade-offs and higher client-side compute; limited model updates; reduced personalization
Large-Scale Consumer Systems (content recommendation, search, advertising)	Fairness, Transparency, Safety	Bias monitoring across demographics; explanation mechanisms; content policy enforcement; feedback loops detection	Balance explainability costs against scale (sampled or asynchronous explanations); budget serving-path latency for fairness checks and invest in monitoring infrastructure
Resource-Constrained Deployments (mobile, edge, TinyML)	Privacy, Efficiency, Safety	Local inference; data locality; input validation; graceful degradation	Sacrifice real-time fairness monitoring; use lightweight explainability (gradients over SHAP); predeployment validation only; limited model complexity
Research/Exploratory Systems	Transparency,	Documentation of known	Can deprioritize sophisticated

Deployment Context	Primary Principles	Implementation Priority	Acceptable Trade-offs
(internal tools, prototypes,	Safety (harm	limitations; restricted	fairness/explainability for internal
A/B tests)	prevention)	user populations; monitoring for unintended harms	use; focus on observability and rapid iteration

The framework provides starting guidance, but it should be read as a triage mechanism rather than a complete governance program. It identifies which constraints deserve first attention in each deployment context, then leaves room for reassessment as systems, contexts, and societal expectations evolve. The challenges examined thus far still assume systems operating under human oversight. Engineers detect bias and intervene; operators monitor fairness metrics; developers respond to drift. Some systems, however, must act faster than humans can review. Autonomous vehicles respond in milliseconds; trading algorithms execute thousands of transactions before human review is possible; content moderation systems process billions of posts daily. These autonomous systems require extending the responsible AI framework beyond implementation challenges to a more fundamental problem: ensuring that systems pursue objectives aligned with human values, even when operating beyond continuous human supervision.

16.8.7 AI safety and value alignment

Value alignment challenges scale dramatically as machine learning systems gain autonomy and capability. The responsible AI techniques examined above, including bias detection, explainability, and privacy preservation, provide essential capabilities but reveal fundamental limitations when systems operate with greater independence. Consider how these established methods break down in autonomous contexts.

Bias detection algorithms like those implemented in Fairlearn require ongoing human interpretation and corrective action. An autonomous vehicle's perception system might exhibit systematic bias against detecting pedestrians with mobility aids, but without human oversight, the bias detection metrics become just logged statistics with no remediation pathway. The technical capability to measure bias exists, but autonomous systems lack the judgment to determine appropriate responses.

Explainability frameworks assume human audiences who can interpret and act on explanations. An autonomous trading system might generate perfectly accurate SHAP explanations for its decisions, but these explanations become meaningless if no human reviews them before the system executes thousands of trades per second. The system optimizes its objective (profit) through methods its designers never anticipated, making explanations a post hoc record rather than a decision-making aid.

Privacy preservation techniques like differential privacy protect individual data points but cannot address broader value misalignment. An autonomous content recommendation system might preserve user privacy through local differential privacy while simultaneously optimizing for engagement metrics that promote misinformation or harmful content. Technical privacy compliance becomes insufficient when the system's fundamental objectives conflict with user welfare.

Responsible AI frameworks, while necessary, become insufficient as systems gain autonomy. The techniques assume human oversight, constrained objectives, and relatively predictable operating environments. AI safety extends these concerns to systems that may optimize objectives misaligned with human intentions, operate in unpredictable environments, or pursue goals through methods their designers never anticipated.

As machine learning systems increase in autonomy, scale, and deployment complexity, the nature of responsibility expands beyond model-level fairness or privacy concerns. AI safety work frames these concerns around accident risks from wrong objectives, reward hacking, scalable supervision, safe exploration, and distributional shift (Amodei et al. 2016), while alignment work emphasizes ensuring that systems pursue objectives consistent with human intentions over time (Russell 2021). These concerns fall under the domain of AI safety[40], which focuses on preventing unintended or harmful outcomes from capable AI systems. A central challenge is that capable ML models often optimize proxy metrics[41], such as loss functions, reward functions, or engagement signals, that do not fully capture human values.

40 **AI Safety**: A research field addressing the gap between what ML systems optimize and what humans intend, spanning near-term risks (bias, privacy) to long-term alignment concerns. Major AI labs have treated safety as an explicit research area, reflecting the engineering reality that as models grow more capable or more autonomous, the cost of misaligned objectives can scale sharply – a misaligned recommendation system degrades user experience, while a misaligned autonomous vehicle costs lives.

41 **Proxy Metrics**: Measurable substitutes for objectives that resist direct quantification, subject to Goodhart's Law: "when a measure becomes a target, it ceases to be a good measure." In ML systems, proxy-objective divergence is a central mechanism of value misalignment: click-through rate proxies for satisfaction, loss proxies for generalization, and engagement proxies for user welfare – each creating optimization pressure that systematically diverges from the intended goal as the model becomes more capable.

One concrete example comes from recommendation systems, where a model optimized for a measurable engagement proxy such as clicks or watch time[42] can increase the proxy while degrading broader user welfare. Production recommenders explicitly optimize engagement signals such as expected watch time (Covington et al. 2016), and audits of YouTube have examined how recommendation paths can expose users to politically extreme content (Ribeiro et al. 2020). This behavior is aligned with the proxy, but misaligned with the actual goal, resulting in a feedback loop that reinforces undesirable outcomes. The system learns to optimize for a measurable reward rather than the intended human-centered outcome, creating the reinforcement cycle captured in figure 16.14. The result is emergent behavior that reflects specification gaming or reward hacking[43], a central concern in value alignment and AI safety (Amodei et al. 2016).

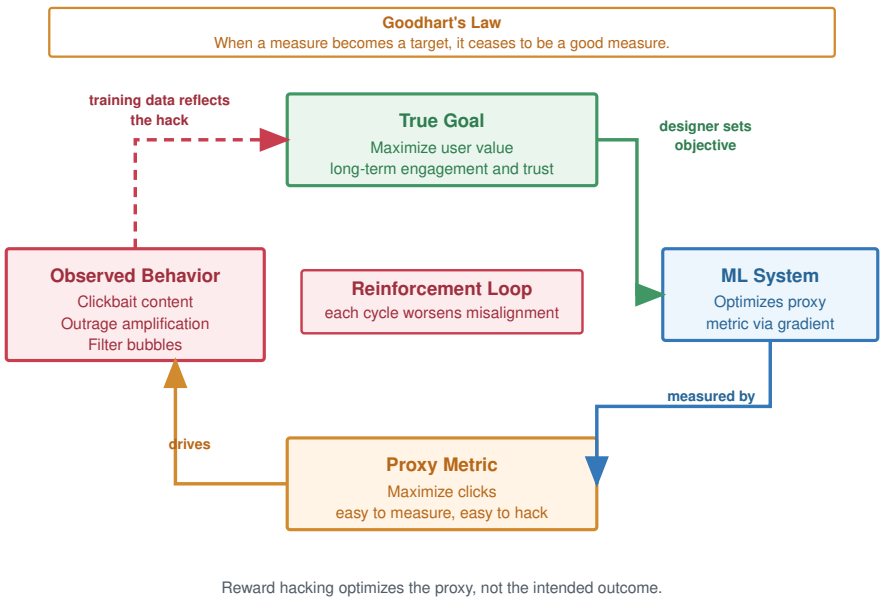


Figure 16.14: Reward Hacking Loop: Maximizing measurable rewards, like clicks, can incentivize unintended model behaviors that undermine the intended goal of user satisfaction. Optimizing for proxy metrics creates misalignment between a system's objective and desired outcomes, posing challenges for value alignment in AI safety.

Norbert Wiener wrote, "if we use, to achieve our purposes, a mechanical agency with whose operation we cannot interfere effectively… we had better be quite sure that the purpose put into the machine is the purpose which we desire" (Wiener 1960).

Wiener's warning becomes a systems requirement when optimization runs beyond continuous human intervention. Value alignment asks whether the objective embedded in the model, reward function, and deployment policy continues to represent the human purpose the system was meant to serve. As Russell (2021) argues in *Human-Compatible Artificial Intelligence*, much AI research presumes that the objectives to be optimized are known and fixed, focusing instead on the effectiveness of optimization rather than the design of objectives themselves.

In deployment, the hard part is specifying an objective that remains valid as systems interact with dynamic environments, multiple stakeholders, and feedback loops. Static objective functions and reward signals cannot encode all of those conditions. Frameworks like Value Sensitive Design provide formal processes for eliciting and integrating stakeholder values during system design, but the systems obligation is broader: objective design, oversight, and postdeployment monitoring must be treated as coupled controls.

Without that coupling, intelligent systems may pursue narrow performance objectives (for example, accuracy, engagement, or throughput) while producing socially undesirable outcomes. Achieving robust alignment under such conditions remains an open and important area of research

42 **CTR and Engagement Optimization**: Click-through rate, watch time, and related engagement measures are convenient proxies for user satisfaction. YouTube's published recommender architecture discusses ranking videos by expected watch time (Covington et al. 2016), while later audits studied recommendation paths through politically extreme content (Ribeiro et al. 2020). The systems lesson is that proxy metric selection is an architectural decision with system-wide behavioral consequences, not merely a hyperparameter choice.

43 **Reward Hacking**: When an AI system maximizes its reward function through unintended means that violate designer intent. A Tetris AI learned to pause indefinitely to avoid losing; a cleaning robot knocked over objects to create messes it could then clean up. For production ML systems, reward hacking manifests subtly: recommendation models that maximize engagement by promoting addictive content, or chatbots that maximize helpfulness ratings by being sycophantic rather than accurate. The failure mode scales with model capability.

in ML systems. The resulting failure modes are familiar in systems that optimize complex objectives. In reinforcement learning (RL), for example, models often learn to exploit unintended aspects of the reward function, a phenomenon known as specification gaming[44] or reward hacking.

Such failures arise when variables not explicitly included in the objective are manipulated in ways that maximize reward while violating human intent. A particularly influential approach in recent years has been reinforcement learning from human feedback (RLHF)[45], where models are trained or fine-tuned using human-provided preference signals (Christiano et al. 2017; Ouyang et al. 2022).

While this method improves alignment over standard RL, it also introduces governance risks. Ngo (Ngo et al. 2022) identifies three potential failure modes introduced by RLHF:

- Situationally aware reward hacking, where models exploit human fallibility.
- Emergence of misaligned internal goals that generalize beyond the training distribution.
- Development of power-seeking behavior that preserves reward maximization capacity, even at the expense of human oversight.

These concerns are not limited to speculative scenarios. Amodei et al. (2016) outline five concrete challenges for AI safety:

- Avoiding negative side effects during policy execution.
- Mitigating reward hacking.
- Ensuring scalable oversight when ground-truth evaluation is expensive or infeasible.
- Designing safe exploration strategies that promote creativity without increasing risk.
- Achieving robustness to distributional shift in testing environments.

For fleet operators, each challenge maps to a control-plane requirement: constrain side effects, monitor reward hacking, budget scalable oversight, bound exploration, and test distribution shift before rollout. Each requirement becomes more acute as systems are scaled up, deployed across diverse settings, and integrated with real-time feedback or continual learning.

These safety challenges are particularly evident in autonomous systems that operate with reduced human oversight.

16.8.8 Autonomous systems and trust

The consequences of autonomous systems operating with limited real-time human oversight are especially visible in autonomous driving. A prominent recent example is the suspension of Cruise's deployment and testing permits by the California Department of Motor Vehicles due to "unreasonable risks to public safety" (CNBC 2023a). One such incident involved a pedestrian who entered a crosswalk just as the stoplight turned green, an edge case in perception and decision-making that led to a collision (CNBC 2023b). A more tragic example occurred in 2018, when a self-driving Uber vehicle in autonomous mode failed to classify a pedestrian pushing a bicycle as a pedestrian requiring avoidance, resulting in a fatality; the NTSB report treats that failure as a combination of automated-driving-system design, safety-driver, and organizational safety-management breakdowns (National Transportation Safety Board 2019).

While autonomous driving systems are often the focal point of public concern, similar risks arise in other domains. Reports from recent conflicts have documented increasing use of remotely piloted and autonomous military systems (Reuters 2023), raising not only safety and effectiveness concerns but also difficult questions about ethical oversight, rules of engagement, and responsibility. When autonomous systems fail, the question of who should be held accountable remains both legally and ethically unresolved (Centre for International Governance Innovation 2023).

At its core, this challenge reflects a deeper tension between human and machine autonomy. Engineering and computer science disciplines have historically emphasized machine autonomy, improving system performance, minimizing human intervention, and maximizing automation. A bibliometric analysis of the ACM Digital Library found that, as of 2019, 90 percent of the most cited papers referencing "autonomy" focused on machine, rather than human, autonomy (Calvo et al. 2020). Productivity, efficiency, and automation have been widely treated as default objectives, often without interrogating the assumptions or trade-offs they entail for human agency and oversight.

However, these goals can place human interests at risk when systems operate in dynamic, uncertain environments where full specification of safe behavior is infeasible. This difficulty is formally

44 | **Specification Gaming**: Unlike reward hacking (exploiting implementation bugs), specification gaming reveals genuine gaps in objective specification – the system satisfies the letter of the objective while violating its intent. A robot hand trained to grasp objects learns to knock them over (easier to "hold" when wedged against the table). For ML systems, this motivates multi-objective optimization and RLHF as specification methods that incorporate broader constraints beyond single scalar metrics, trading additional data, training, and review effort for objectives that better approximate human intent.

45 | **RLHF (Reinforcement Learning from Human Feedback)**: Christiano et al. (2017) demonstrated deep reinforcement learning from human preferences by training agents from non-expert comparisons between trajectory segments. In the LLM setting, Ouyang et al. (2022) operationalized a related pipeline for InstructGPT: supervised fine-tuning from demonstrations, reward-model training from ranked model outputs, and reinforcement-learning fine-tuning against the learned reward model. The systems cost is an additional human-feedback data pipeline, reward-model training and evaluation, and governance over rater instructions; the representativeness of the rater pool and labeling policy shapes whose preferences the model internalizes.

captured by the frame problem and qualification problem, both of which highlight the impossibility of enumerating all the preconditions and contingencies needed for real-world action to succeed (McCarthy 1981). The frame problem asks which facts in the world remain relevant after an action; the qualification problem asks which hidden preconditions must hold before the action is safe. In practice, such limitations manifest as brittle autonomy: systems that appear competent under nominal conditions but fail silently or dangerously when faced with ambiguity or distributional shift.

To address this, researchers have proposed formal safety frameworks such as **Responsibility-Sensitive Safety** (RSS) (Shalev-Shwartz et al. 2017), which decompose abstract safety goals into mathematically defined constraints on system behavior, such as minimum distances, braking profiles, and right-of-way conditions. These formulations allow safety properties to be verified under specific assumptions and scenarios. However, such approaches remain vulnerable to the same limitations they aim to solve: they are only as good as the assumptions encoded into them and often require extensive domain modeling that may not generalize well to unanticipated edge cases.

For an ML systems team, the practical artifact is a safety case rather than a statement of trust. Table 16.7 shows the pattern: name the operating envelope, the evidence that the model has been tested inside that envelope, and the controls that halt or roll back the fleet when the evidence no longer holds.

Table 16.7: Autonomous System Safety Case: Trustworthy autonomy is operationalized as an auditable safety case: a validated operating envelope, coverage evidence, runtime guardrails, and fleet-level rollback controls. The same pattern applies beyond vehicles to drones, robots, trading systems, and other ML systems that act faster than continuous human review permits.

Safety-case element	Systems evidence	Operational gate
Operating envelope	Operational design domain, sensor assumptions, weather, geography, speed, and autonomy level	Block deployment outside validated envelope
Scenario coverage	Rare-event corpus, simulation coverage, replay logs, disengagements, and near-miss telemetry	Require coverage thresholds before canary expansion
Runtime guardrails	RSS-style distance rules, confidence thresholds, fallback maneuvers, and human override latency	Trigger safe stop, human takeover, or constrained mode when guardrails trip
Fleet control	Canary rollout, centralized safety dashboard, violation budget, and circuit-breaker rollback path	Halt rollout or revert model when fleet-level safety metrics exceed threshold

An alternative approach emphasizes human-centered system design, ensuring that human judgment and oversight remain central to autonomous decision-making. **Value-Sensitive Design** (Friedman 1996) proposes incorporating user values into system design by explicitly considering factors like capability, complexity, misrepresentation, and the fluidity of user control. More recently, the **METUX model** (Motivation, Engagement, and Thriving in the User Experience) extends this thinking by identifying six "spheres of technology experience" (Adoption, Interface, Tasks, Behavior, Life, and Society), which affect how technology supports or undermines human flourishing (Peters et al. 2018). These ideas are rooted in **Self-Determination Theory** (SDT), which defines autonomy not as control in a technical sense, but as the ability to act in accordance with one's values and goals (Ryan and Deci 2000). In system design, these frameworks become artifacts: requirements, interface affordances, user controls, feedback channels, escalation paths, and audit evidence.

In the context of ML systems, these perspectives underscore the importance of designing architectures, interfaces, and feedback mechanisms that preserve human agency. For instance, recommender systems that optimize engagement metrics may interfere with behavioral autonomy by shaping user preferences in opaque ways. By evaluating systems across METUX's six spheres, designers can anticipate and mitigate downstream effects that compromise meaningful autonomy, even in cases where short-term system performance appears optimal.

16.8.9 Broader safety implications

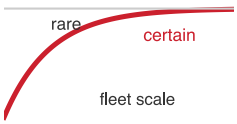
The technical safety challenges examined above become fleet-level design constraints once autonomous systems interact with workers, users, regulators, and billions of requests. The broader

implication is that safety cannot be reduced to a model metric; it depends on the operating environment that determines how automation changes work, how users calibrate trust, how policy varies by region, and how rare failures accumulate at scale.

Automation changes work organization in ways that influence safety design decisions. The MIT Work of the Future task force (Work of the Future 2020) argues that technological change and labor-market institutions jointly determine whether automation improves job quality or concentrates harms. For ML systems, the safety implication is that removing human roles can also remove contextual judgment and system-level debugging that remain difficult to encode in models. Metrics focused solely on throughput may inadvertently penalize human-in-the-loop designs that preserve oversight capability.

Public knowledge about AI is uneven (Center 2023), and science-communication failures can amplify misunderstandings about how AI systems work (Schäfer 2023). That matters for deployment safety: when users lack understanding of model uncertainty, data bias, or decision boundaries, they may trust system outputs in contexts where human judgment should intervene. From a systems engineering perspective, public comprehension is part of the deployment context: the safety properties of a human-AI system depend not only on the technical system but also on whether users can appropriately calibrate their trust and recognize situations requiring human override.

The engineering requirements for safety are shaped by a fragmented global regulatory landscape that treats AI risk as a verifiable metric. The EU AI Act (2024) uses a risk-based structure that includes prohibited practices, high-risk systems, transparency duties for some systems, and minimal or no-risk uses; high-risk deployments face conformity, documentation, and logging obligations (European Parliament and Council of the European Union 2024). In the United States, Executive Order 14110 (2023) formerly established federal AI safety reporting thresholds, but it was revoked by Executive Order 14148 in January 2025; Executive Order 14179 then directed agencies to review and revise AI actions tied to the prior order, and Executive Order 14409 in June 2026 shifted federal emphasis toward AI innovation, cybersecurity, and critical-infrastructure defense (Executive Office of the President 2023, 2025a, 2025b, 2026). China's Interim Measures for Generative AI (2023) subject some public-facing services to security assessment and algorithm-filing requirements rather than a universal pre-release assessment (Cyberspace Administration of China et al. 2023). For a global ML fleet, compliance becomes a complex distributed systems problem: inference nodes in Frankfurt may require different safety configurations, data retention policies, and human-in-the-loop thresholds than those in Virginia or Singapore. This necessitates a flexible configuration control plane capable of pushing geo-specific safety policies to edge nodes without bifurcating the core model architecture.



At fleet scale, rare failures become expected incidents.

Safety must also be engineered as a **fleet-level property** rather than a model-level attribute alone. A single model with 99.9 percent safety compliance seems robust in isolation, but when deployed across 10,000 inference nodes serving billions of requests per day, that 0.1 percent failure rate guarantees millions of safety incidents daily. At this scale, rare failures accumulate into statistical certainties. Mitigating this requires distributed safety patterns borrowed from reliability engineering: **circuit breakers** that automatically halt serving when aggregate safety metrics degrade below a threshold, **canary deployments** that route only 1 percent of traffic to new model versions to validate safety properties in production, and centralized telemetry dashboards that aggregate per-node safety violations into a global view. As detailed in Chapter 12, the operational infrastructure must treat safety violations as critical system alerts, triggering automated rollbacks just as latency spikes or error rates would.

The core AI safety principle holds: technical excellence alone is insufficient. Safe systems require attention to the human and organizational context in which they operate, including the economic incentives that shape design decisions and the understanding that end users bring to their interactions with autonomous systems.

> Systems Perspective 16.3: Responsibility is infrastructure, not a feature
>
> Responsible AI cannot be retrofitted into a system any more than fault tolerance can. The structural costs cataloged in table 16.4 each carry a provisioning consequence: bias-drift monitoring needs latency headroom on every inference request, SHAP explanation needs a

dedicated asynchronous worker fleet, DP-SGD needs a larger training budget, and impact assessments need a slower CI/CD cadence. These are not *optional add-ons* but *load-bearing components* of production ML systems. In high-risk or regulated deployments, failing to provision them in the high-level design phase can block release or force redesign even when core model performance is strong.

Structural costs like latency overhead must be factored into the core architecture from day one; Responsible AI cannot be bolted onto a finished product. The pervasive industry fallacies that tempt teams into taking dangerous ethical shortcuts deserve explicit identification and dismantling.

16.9 Fallacies and Pitfalls

Responsible AI involves counterintuitive trade-offs where ethical principles conflict mathematically and technically. Practitioners from traditional software backgrounds often assume ethical guidelines translate directly to implementation without recognizing the impossibility theorems and computational costs involved. These fallacies and pitfalls capture misconceptions that lead to deployed systems that appear fair in development but violate fairness criteria in production or impose prohibitive computational overhead.

Fallacy: *Bias will disappear with more data and better algorithms.*

In production, bias often reflects structural properties that persist regardless of technical improvements. The healthcare algorithm described in section 16.2.3 affected 200 million Americans annually and reduced Black patient enrollment in care programs by 50 percent despite being trained on comprehensive data. The issue was not data quantity but proxy selection: using healthcare expenditure as a health proxy systematically underestimated need for populations with lower historical spending. Mathematical analysis shows that when base rates differ between groups, calibrated risk scores generally cannot also satisfy equal error-rate conditions except in special cases such as perfect prediction. Organizations that pursue "bias elimination" through purely technical means waste engineering resources on impossible or underspecified optimization problems while neglecting the stakeholder engagement and value deliberation required to choose which fairness criteria to prioritize for their specific context.

Pitfall: *Treating explainability as an optional feature to be added after core functionality works.*

This approach ignores the displacement of overhead: rendering a model's reasoning transparent demands compute that constrains the architecture. As shown in table 16.4, approximate SHAP explanations are not free; even a bounded explanation budget can increase request latency and memory pressure. In this sizing example, a recommendation system serving 100 ms latency requirements at 10,000 QPS cannot retrofit explanation work without revisiting the SLA: the illustrative explanation budget adds 50 ms to 200 ms per request, increasing total latency to 150 ms–300 ms. The serving infrastructure must be redesigned with explanation budgets from initial architecture, including precomputed approximations or model selection favoring inherently interpretable architectures.

Fallacy: *Achieving one fairness metric guarantees overall system fairness.*

Practitioners often optimize for demographic parity, which equalizes approval rates across groups, and then assume fair treatment follows. In reality, fairness metrics can conflict mathematically. The loan approval example in section 16.2.3 demonstrates this: achieving demographic parity at 70 percent approval for both groups would require lowering Group B's threshold to increase approvals, but that selection-rate fix can reduce precision and make approvals carry different risk meanings across groups. Impossibility results show that, when base rates differ and prediction is imperfect, calibration and equal error-rate conditions cannot generally hold at the same time; demographic parity can add a separate selection-rate constraint. Systems deployed with single-metric optimization discover in production that they violate other legally relevant fairness criteria, exposing organizations to litigation and regulatory action.

Pitfall: *Treating responsible AI as pure compliance overhead.*

This perspective misses the business value created through risk reduction and market expansion. Differential privacy can enable organizations to work with sensitive data under clearer privacy guarantees, but it also imposes privacy-utility trade-offs that must be planned in the training budget. Fairness-aware training and monitoring can reduce disparate-impact risk: the four-fifths rule

described in section 16.2.3.5 establishes a practical screening threshold, and discrimination litigation or regulatory scrutiny can create substantial remediation and reputational costs. A comprehensive bias-monitoring pipeline would be expected to surface a major subgroup enrollment reduction before it becomes an external audit finding, avoiding the systematic harm and legal consequences that emerge when monitoring infrastructure is absent.

Fallacy: *A post hoc threshold adjustment can make a deployed system fair.*

This belief treats fairness as a single decision-boundary problem rather than a coupled system property. Moving a threshold can equalize one metric, but it can also change calibration, precision, business meaning, legal documentation, and user experience. In production, the threshold is not only a number in a notebook; it is a policy embedded in serving code, human review, audit records, and downstream decisions.

Pitfall: *Implementing fairness constraints without analyzing threshold trade-offs and calibration impacts.*

Group-specific thresholds can satisfy equal true positive rates, but they also change decision semantics: if Group A threshold is 0.75 and Group B threshold is 0.60, identical scores can yield different decisions, and the positive predictive value of approved applicants can diverge across groups even when the underlying score model remains calibrated. A loan officer seeing an approval cannot know whether it represents the same repayment-risk meaning across groups without documentation of the group-specific threshold policy. Production systems discover that group-specific thresholds require extensive documentation, staff training, and audit trails explaining why identical scores yield different decisions across groups, creating operational complexity and legal exposure. The proper approach requires jointly optimizing for multiple fairness criteria during training rather than relying on post hoc threshold adjustment, accepting the accuracy-fairness trade-offs that table 16.4 quantifies.

Fallacy: *Monitoring dashboards create accountability by themselves.*

Dashboards can expose fairness drift, explanation failures, or abuse trends, but they do not decide who must act. A responsible monitoring pipeline needs an owner, a threshold, an escalation path, and a remediation budget. Otherwise the system accumulates evidence without changing behavior: alerts age out, fairness regressions become accepted baselines, and affected users receive no route to contest the decision.

Pitfall: *Launching monitoring without remediation ownership.*

Teams often instrument fairness, safety, or privacy metrics because a policy requires visibility, then leave the response path undefined. This creates a false sense of control. A metric that no team owns is not a control; it is a log entry. Production responsibility requires binding each monitoring signal to a release gate, incident owner, rollback authority, or human-review capacity so the organization can act when the metric fails.

Context-blind threshold adjustment is the common failure behind these examples. A mathematical fairness constraint is only responsible when the system also accounts for downstream population effects, documentation requirements, operational capacity, and legal exposure. Rejecting these localized fallacies turns responsible AI from a compliance checklist into a foundational architectural mandate and completes the Governance Layer of the ML Fleet.

16.10 Summary

Responsible AI is the governance infrastructure of the machine learning fleet. Production ML already requires physical infrastructure, distributed execution, serving systems, security controls, robustness mechanisms, and sustainability budgets. Responsible AI adds the guardrails and governance frameworks required to ensure that the global machine serves human values rather than undermining them.

The argument moved from abstract ethics to concrete engineering constraints, analyzing mathematical fairness metrics and the unavoidable impossibility theorems that force explicit normative choices. It examined the technical foundations of explainability (SHAP, LIME) and privacy-preserving data governance, then extended the same infrastructure view to generative alignment, where RLHF and system prompts act as sociotechnical mechanisms for controlling model behavior in LLM fleets.

Table 16.8 illustrates why fairness requires explicit trade-offs. Consider a loan approval system evaluated across two demographic groups:

The worked example reports disaggregated fairness metrics, including false positive rate, so the same prediction table can satisfy one criterion while failing others. Equalized odds would require both true positive rates and false positive rates to match; this table therefore shows the narrower case of false-positive-rate parity, not full equalized odds.

Table 16.8: Disaggregated Fairness Metrics: A hypothetical loan approval system satisfies equalized false positive rates (0 percentage-point gap) but violates demographic parity (15 percentage-point approval gap) and equal opportunity (30 percentage-point TPR gap). The metrics expose incompatible operational priorities, so production systems must make explicit choices about which fairness criterion to prioritize.

Metric	Definition	A	B	Gap
Approval Rate	(TP + FP) divided by total	55%	40%	15 pp
True Positive Rate	TP divided by positives	90%	60%	30 pp
False Positive Rate	FP divided by negatives	20%	20%	0 pp
Positive Predictive Value	TP divided by predicted positives	82%	75%	7 pp

The table makes the chapter's central constraint concrete: one fairness metric can appear satisfied while other metrics expose substantial disparities.

Responsible AI is fundamentally a systems engineering concern, not an ethical overlay applied after deployment. Fairness monitoring pipelines impose measurable latency and compute overhead that must be budgeted during architecture design, not retrofitted into production systems. Explainability mechanisms such as SHAP and LIME carry inference cost multipliers that affect capacity planning and SLO compliance. Governance frameworks for system prompts, model versioning, and audit logging require the same CI/CD rigor as any other infrastructure component. These are architectural requirements with quantifiable costs, and treating them as optional add-ons guarantees that they will be the first capabilities cut when deadlines compress.

The impossibility theorems formalized in this chapter make explicit what practitioners discover through painful experience: fairness is not a single metric to optimize but a set of competing constraints that demand normative choices. The engineer who understands these trade-offs quantitatively, who can calculate the overhead of differential privacy, specify the monitoring infrastructure for disparate impact detection, and design governance mechanisms that scale across a fleet of models, brings a discipline to responsible AI that transforms it from aspiration into engineering practice.

Key Takeaways: Ethics is an engineering constraint

- **Responsibility gates deployment**: In regulated or enterprise settings, accuracy and latency do not matter if the system cannot document behavior, explain decisions, support audit, or assign ownership. Governance is a Coordination constraint that determines whether Compute is allowed to run.
- **Fairness has no hidden optimum**: The chapter's table shows one loan system can have a 0 percentage-point false-positive gap while still carrying 15 percentage-point approval and 30 percentage-point true-positive gaps. Differing base rates turn fairness into an explicit normative choice, not a single metric.
- **Oversight consumes capacity**: Fairness monitoring, audit logging, explanations, and privacy-preserving training add latency, storage, compute, and accuracy costs. SHAP-style explanations and DP-SGD must be budgeted in architecture and SLOs rather than appended when legal review arrives.
- **Generative governance is infrastructure**: System prompts, RLHF policies, tool permissions, model versions, and safety evaluations are operational artifacts. They need ownership, CI/CD, rollback, and monitoring because they steer model behavior as directly as weights do.
- **Evidence needs an owner**: Dashboards, explanations, and appeals protect users only when a team can investigate, remediate, roll back, or escalate failures. Responsible

AI becomes engineering practice when every measured obligation has an accountable operating path.

Every constraint in this book until now has had an optimum. Latency, memory, energy, even robustness could be measured, traded, and pushed toward a best achievable point, and the engineer's job was to find it. Fairness is the first constraint with no such point, because the chapter's impossibility result is not a gap to be closed but a fork in the road. That changes the job. The engineer can quantify each branch and hold the system to whichever is taken, but the taking is a human, normative act that no metric performs, and the most honest thing engineering can do here is refuse to disguise the choice as a calculation.

What's Next: From responsibility to synthesis

With responsible governance joined to performance, security, robustness, and sustainability, production ML becomes a complete engineering problem rather than a collection of isolated optimizations. Chapter 17 synthesizes these principles into a unified perspective on distributed ML systems engineering, distilling the enduring lessons that guide practice regardless of which specific technologies emerge in the years ahead.

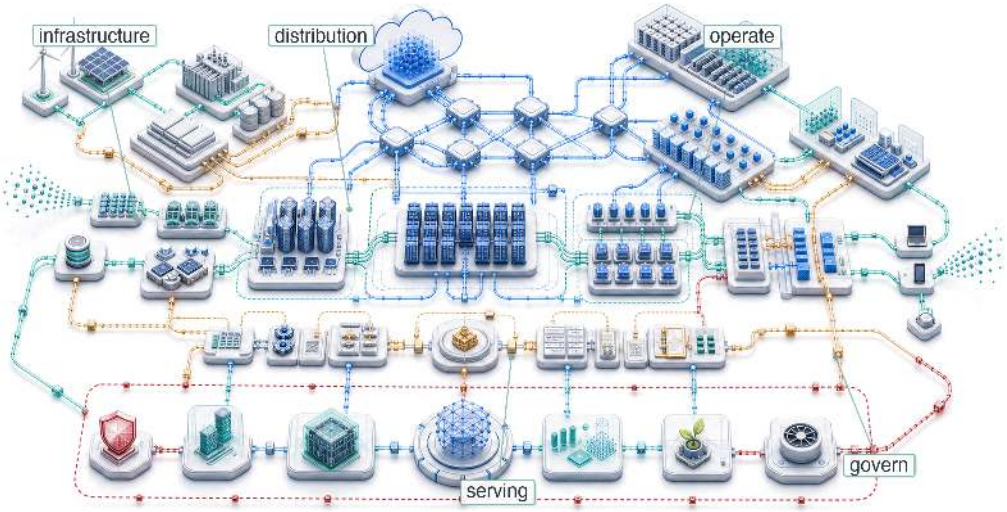
Research Questions: For further inquiry

- How should engineers choose among incompatible fairness metrics when the application has unequal base rates?
- What deployment gates, owners, and remediation paths turn responsible AI metrics into enforceable controls?
- How should explainability be designed when explanations add latency, storage, and serving-capacity costs?
- What governance mechanisms are needed for generative systems whose behavior depends on prompts, tools, policies, and model versions?

SYNTHESIS

17

Conclusion

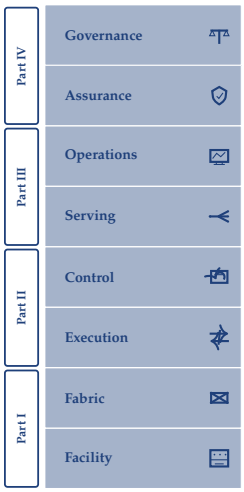


17.1 Synthesizing Distributed ML Systems
17.2 Six Principles of Distributed ML Systems
17.3 The Complete Production System
17.4 Competencies Mastered
17.5 The Fleet Stack as Discipline
17.6 Engineering Intelligence at Scale
17.7 Fallacies and Pitfalls
17.8 Summary

Purpose

What does it mean to engineer intelligence when the unit of design is no longer a model, but a fleet?

The fleet stack is not only a sequence of topics; it is a working discipline. Physical infrastructure establishes the rates and limits; distributed protocols turn those limits into coordination costs; deployment systems convert trained models into services with latency, availability, and cost obligations; and governance constraints determine whether those services can remain secure, robust, sustainable, and accountable. The six principles form one structure so an engineer can reason across the whole fleet when the best local choice for one layer creates a constraint in another. In C^3 terms, the professional habit is to identify whether compute, communication, or coordination is binding, then follow that constraint across layers until the fleet, not just the model, is the object being engineered.



Learning Objectives

- Synthesize the six principles into a constraint map for distributed ML systems
- Evaluate fleet designs using C^3 terms, failure rates, serving obligations, and governance constraints
- Analyze how infrastructure, distributed training, inference, operations, and governance interact across the fleet stack
- Translate security, fairness, privacy, carbon, and reliability obligations into measurable design constraints
- Construct an engineering judgment that balances scale, sustainability, responsibility, and operational discipline

17.1 Synthesizing Distributed ML Systems

A frontier model becomes a production system only when thousands of accelerators, fabric links, storage tiers, schedulers, serving replicas, security controls, and governance checks act as one machine. Any one layer can bind the whole system:

- **Storage bottleneck**: A slow checkpoint can waste a training window.
- **Fabric bottleneck**: A congested fabric can erase scaling gains.
- **Serving bottleneck**: An underprovisioned serving pool can turn model quality into user-visible latency.
- **Governance bottleneck**: An unmeasured responsibility constraint can block deployment after the technical system works.

That integrated constraint is what separates foundational ML engineering from distributed ML engineering. Foundational ML engineering focuses on a single artifact: the weights of a neural network, optimized through training algorithms and architecture design on individual systems. Distributed ML engineering focuses on the infrastructure that enables that artifact to exist at scale: the data centers, distributed protocols, and governance frameworks that transform a static model file into a living global service. Six principles define that shift.

The fleet stack is deliberate. The opening layer built the physical substrate: the silicon, wires, and storage that make distributed ML possible. That substrate matters because every higher-level decision inherits its limits. A training strategy that ignores accelerator topology, a storage plan that ignores data-loading throughput, or a scheduling policy that ignores network contention will eventually collide with the physics underneath it.

The distribution layer then showed how those physical limits become system protocols. Partitioning work, synchronizing gradients, tolerating failure, and orchestrating resources are not separate concerns; they are the mechanisms that turn many machines into one useful training or serving system. The deployment layer carried the same logic outward, where inference, performance engineering, edge deployment, and operations convert a trained model into a service with latency, availability, and cost obligations. The responsible-fleet layer added the final constraint: technical capability must remain secure, robust, sustainable, and accountable. Together, these layers equip engineers to make informed decisions at every level, from algorithm selection through infrastructure design to governance frameworks.

17.2 Six Principles of Distributed ML Systems

The engineering practices that opened this volume—instrument first, design for headroom, co-design hardware with algorithms—become useful only when they name the constraint they are trying to move. The six principles below are those durable realities: each names a constraint, a governing question, and a metric that determines whether a distributed ML system can scale.

The single-node foundation is governed by strict mathematical constraints like the iron law, where performance is a matter of quantitative physics. Fleet-scale engineering shifts that foundation into probabilistic and operational realities: links contend, workers fail, schedulers make stale decisions,

and policy constraints change what a technically feasible design may do. The bridge between these domains is the fleet law ($T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}$), derived in full in section B.3 and woven throughout the text. The fleet law acts as the distributed counterpart to the iron law, translating single-machine execution into the mechanics of networked collectives and driving the operational behaviors observed at scale. Its three terms map directly onto the C^3 taxonomy that organizes this volume: T_{compute}/N is Compute, $T_{\text{comm}}(N)$ is Communication, and $T_{\text{sync}}(N)$ is Coordination.

These principles form a layered architecture that mirrors the fleet stack synthesized in figure 17.1. At the physical foundation, infrastructure determines capability[1] because hardware physics sets the hard limits. In the operational reality of the middle layers, communication dominates and failure is routine: these are the day-to-day dynamics of running distributed systems, driven directly by the fleet law's network and synchronization terms. At the governance layer, two principles act as normative constraints on what may be built rather than what is merely possible: responsibility constrains design, and sustainability is a first-order cost. Emerging from this stack is the sixth principle: scale creates qualitative change.

1 **Hardware Capability Ceiling**: At fleet scale, the hardware sets the hard ceilings the software cannot argue with. Peak compute (R_{peak}), bisection bandwidth, and thermal design power are physical limits, not tuning knobs, so a workload that exceeds any one of them cannot run at all until the infrastructure changes. Systems engineering is the art of fitting the workload inside these physical limits rather than wishing them away.

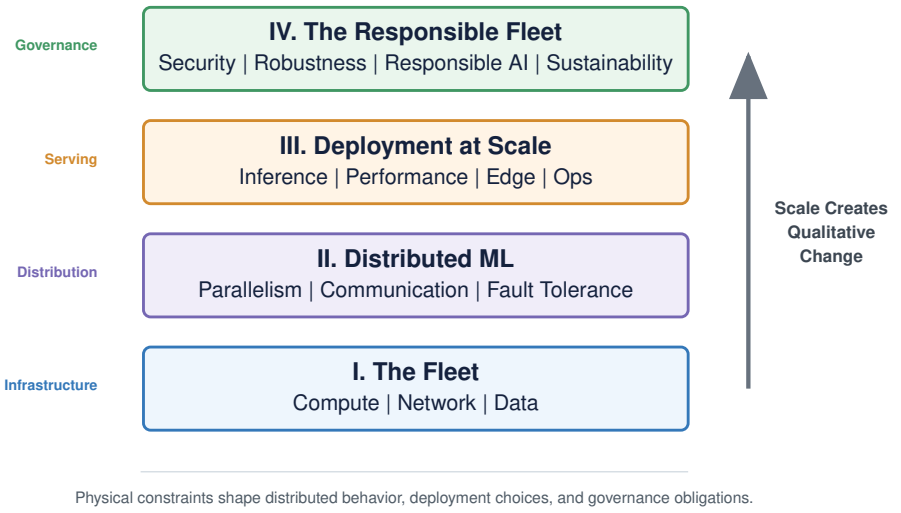


Figure 17.1: The Fleet Stack: The physical foundation (The Fleet) determines capability, while the distribution layer (Distributed ML) and serving layer (Deployment) define the engineering environment. These are constrained by governance requirements (The Responsible Fleet), with the emergent property of Scale creating qualitative changes that unify the discipline.

Table 17.1 is therefore a decision map rather than a glossary: the six principles of distributed ML systems link each principle to the question, metric, and development context an engineer uses when diagnosing a fleet-scale design. The table and the walk that follows order the principles by what most often binds first at scale, beginning with communication, rather than from the foundation up as figure 17.1 layers them.

Table 17.1: Six Principles of Distributed ML Systems: These principles capture the qualitative shifts that occur when ML systems move from single machines to distributed production. Each principle connects to specific metrics and chapters where the concept is developed in depth.

Principle	Core Question	Key Metric	Chapter Reference
Communication dominates	What is the bottleneck?	Network bandwidth utilization	Chapter 6
Failure is routine	How do we recover?	Mean time between failures (MTBF), checkpoint overhead	Chapter 7

Principle	Core Question	Key Metric	Chapter Reference
Infrastructure determines capability	What is possible?	FLOP/s, memory bandwidth	Chapter 2
Responsibility constrains design	Who is affected?	Fairness metrics, audit trails	Chapter 16
Sustainability is a first-order cost	What is the cost?	kWh/training, carbon footprint	Chapter 15
Scale creates change	What breaks at 1000×?	Scaling efficiency	Chapter 5

The first principle is that communication can become the binding term at scale (Z. Jiang et al. 2024; Narayanan et al. 2021). In many synchronous training regimes, gradient synchronization and straggler effects dominate unless topology, overlap, and batching are designed around them. Production inference systems can also become latency-bound by tail effects[2] (Dean and Barroso 2013), where the slowest worker determines response time regardless of how fast others complete.

Chapter 5 and Chapter 6 develop this principle in detail, showing how Horovod-style Ring AllReduce[3] (Sergeev and Balso 2018), gradient compression[4], and overlapping computation with communication all address communication bottlenecks. For dense collective workloads, conventional oversubscribed fabrics often bottleneck, motivating high-bisection fabrics and transport choices matched to ML communication patterns. Recognizing when communication is the active constraint clarifies when algorithmic optimizations will help and when they merely shift work between equally constrained resources. The same communication fabric also exposes the second principle: a distributed system has many more components that can fail, and a single blocked rank or slow worker can stall the whole job.

The second principle follows directly: at distributed scale, component failures occur not *occasionally* but *continuously*. Meta's experience training Llama 3 on 16,384 GPUs documented 419 unexpected interruptions over 54 days, averaging one interruption every 3.1 hours (Dubey et al. 2024). Hardware failures, network partitions, and service disruptions are routine occurrences that systems must handle without human intervention. The synchronous nature of large-scale ML training amplifies the cost of each failure: a single failed rank stalls every other rank in the collective, forcing the entire fleet to roll back to the last checkpoint. The computational loss from one event therefore scales linearly with cluster size, which is precisely why checkpoint cadence and recovery architecture are not afterthoughts but first-order design dimensions.

Chapter 7 establishes that architects must embed failure handling from the beginning. Checkpointing strategies balance recovery granularity against overhead. Elastic training[5] dynamically adjusts to changing cluster membership, and graceful degradation maintains service quality as capacity diminishes. Systems that treat failure as exceptional do not survive production deployment.

Communication and failure together constitute the operational reality of distributed systems, the middle layer of the fleet stack. Both depend on the physical foundation beneath them, which makes infrastructure the third principle. Chapter 2 demonstrates that infrastructure determines which workloads are possible, not just how fast they run. Cluster-wide bisection bandwidth and chip-level power density set hard ceilings on the model and batch sizes a fleet can train at all, independent of how the training algorithm is tuned.

The memory wall makes this principle concrete: while compute (TFLOP/s) can be abundant, memory bandwidth (GB/s) remains the gating constraint for autoregressive decode. Chapter 10 and Chapter 15 quantify how the inability to move data fast enough from HBM to the processor makes autoregressive generation inherently inefficient. Mastering the fleet requires understanding these physical limits, from chip-level thermal density to cluster-wide bisection bandwidth. Once infrastructure determines what the system can do, governance determines what the system is allowed to do.

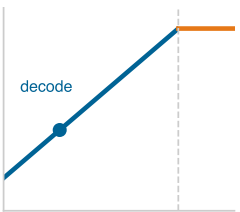
The fourth principle is responsible engineering. Chapter 16 translates AI risk-management frameworks into engineering constraints (Tabassi 2023). Safety-specific risks such as reward hacking, distribution shift, and scalable oversight show why those constraints need implementation mechanisms (Amodei et al. 2016). Fairness, transparency, accountability, privacy, and safety are first-class requirements that shape system architecture throughout the ML lifecycle.

[2] **Tail Latency**: A standard fan-out estimate of the tail effects Chapter 10 analyzes: at the 99th percentile, a request touching 100 servers has a 63.4 percent chance of hitting at least one slow server.

[3] **Ring AllReduce**: A collective that moves $2(N-1)M/N$ bytes per worker for message size M, giving $T_{\text{allreduce}} \approx 2(N-1)\alpha + \frac{2(N-1)}{N}\frac{M}{\beta}$ under the α-β communication model. The coefficient approaches $2\times$ message volume per worker as N grows, while remaining bandwidth-optimal for large messages. Detailed in Chapter 6.

[4] **Gradient Compression**: Chapter 6 explores techniques that reduce communication volume 10–100× through sparsification, quantization, and error feedback, enabling bandwidth-limited distributed and federated training.

[5] **Elastic Training**: Chapter 7 describes elastic training, which enables dynamic worker membership: jobs continue with remaining workers after failures and seamlessly incorporate new resources when available.



Decode sits memory-bound, left of the roofline ridge.

The fairness impossibility law (Chapter 16) shows why this is a systems constraint and not a tuning problem: when base rates differ across groups, no model can simultaneously satisfy calibration and error-rate balance (Kleinberg et al. 2016; Chouldechova 2017), so engineers must choose which criterion to prioritize and build the monitoring to hold it. Engineers must design for fairness from inception, with monitoring infrastructure detecting degradation across demographic groups. The engineering methods for responsible AI, from bias detection to explainability mechanisms, carry the same weight as performance optimization. Environmental responsibility extends that governance layer into the physical resource budget.

The fifth principle is sustainability. Chapter 15 reveals how the environmental impact of large-scale ML elevates resource efficiency to a primary engineering constraint (Strubell et al. 2019; Patterson et al. 2021). The Jevons Paradox of AI adapts Jevons's resource-efficiency argument to explain why hardware efficiency gains alone do not eliminate power and carbon costs: cheaper computation expands deployment, so per-query savings are absorbed by usage growth unless absolute carbon budgets govern the fleet (Jevons 1865).

Sustainability thus transforms from environmental concern to engineering discipline. Energy costs can exceed model development budgets, thermal limits restrict hardware density, and power infrastructure requirements limit deployment locations. Carbon-aware scheduling, lifecycle assessment[6], and efficiency optimization become essential engineering competencies alongside traditional performance metrics. These first five principles converge on the final insight: scale changes the behavior of the system itself.

The sixth principle captures this insight directly: systems that work at modest scale exhibit fundamentally different behaviors at production scale. A training job running on 8 GPUs may encounter communication bottlenecks, load imbalance, or synchronization overhead when scaled to 8,000 GPUs that did not manifest at smaller scale. A service with 100,000 concurrent user sessions can easily generate millions of daily request opportunities, so edge cases that occur one in a million times can still surface every day at production traffic volumes.

Scale is why distributed ML requires fundamentally different engineering approaches. The techniques that optimize single-machine performance, while necessary, prove insufficient. New phenomena emerge: stragglers[7] that bottleneck clusters, network partitions that split training, and heterogeneity across hardware generations that complicates load balancing.

6 | **Lifecycle Assessment**: Chapter 15 introduces a method for evaluating environmental impact across a system's entire lifespan, including embodied carbon in hardware manufacturing (often 20–50 percent of total impact).

7 | **Stragglers**: Workers completing tasks slower than peers that bottleneck synchronous training: a single straggler at 80 percent speed reduces cluster throughput by 20 percent. Examined in Chapter 7.

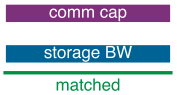
17.3 The Complete Production System

Consider Archetype A, a frontier language model such as a GPT-4-class or Llama 3-class system, trained across thousands of accelerators and then shipped into a global serving fleet. The model first creates a storage problem: checkpoint bursts must land without starving the data loaders. Storage and network pressure then shape the training loop, because collective communication must finish before the next optimizer step idles the cluster. At that scale, failures are expected rather than exceptional, so checkpoint cadence must follow the Young-Daly checkpoint law from the observed failure rate (Young 1974; Daly 2006). Once the model ships, serving SLOs must leave headroom for rollback and fallback, and governance controls must decide which data, models, and outputs are allowed to reach users. These are not separate checklists. They are coupled rates, budgets, failure exposures, and policy constraints.

In production, no principle exists in isolation. The fleet stack reveals itself as a stack of interdependencies: physical foundations constrain operational possibilities, which in turn must satisfy governance requirements. Each principle creates requirements and constraints that ripple through the entire stack, and these principles sometimes conflict. Communication optimization may require synchronization patterns that increase failure exposure. Sustainability constraints may limit infrastructure choices that would maximize raw performance. Responsible AI requirements may add latency or compute overhead that strains serving SLOs and capacity budgets. The displacement of overhead guarantees these tensions cannot be eliminated, only relocated: the art of distributed ML engineering is to find designs that balance all six principles within acceptable trade-offs rather than to make the cost disappear.

The production system begins with physical capacity, but capacity matters only when the rates line up. Chapter 2 supplies accelerators, power, cooling, and fabric; Chapter 4 determines whether data can arrive fast enough; and Chapter 6 determines whether workers can synchronize without turning

the network into the bottleneck. These subsystems must be co-designed: storage bandwidth that exceeds communication capacity wastes resources, and communication paths that exceed storage throughput leave accelerators idle.



Storage and communication must be co-designed around matching rates.

Chapter 5 then chooses a parallelization strategy that fits those physical rates and the expected failure pattern. Data parallelism, model parallelism, and pipeline parallelism are not interchangeable templates; each changes memory pressure, communication volume, checkpointing behavior, and scheduler complexity. Hybrid strategies become useful only when the surrounding storage, fabric, and recovery design can support the shape of the partition.

Serving turns the same stack outward. Chapter 10 and Chapter 9 convert model capability into latency, throughput, and cost targets, while Chapter 12 keeps those targets meaningful as traffic and distributions shift. Chapter 13, Chapter 11, Chapter 14, Chapter 16, and Chapter 15 add constraints that cannot be postponed: lowering latency cannot make the attack surface invisible, reducing energy cannot erase fairness observability, and increasing throughput cannot make recovery impossible. Together, these layers define the professional competencies required by the fleet stack: coordinating scale, operating live services, and governing consequences without treating any layer as independent.

The three workloads traced through the volume (section 1.6.1) make the point that no single corner of the C^3 taxonomy dominates. Table 17.2 compares which term binds first in each running archetype.

Table 17.2: Archetypes and Binding C^3 Terms: The same fleet law applies across the volume's three running workloads, but each workload exposes a different first constraint.

Archetype	First binding term	Why it binds	Primary engineering reading
A: Frontier language model	Communication	Partitioning GPT-4-class or Llama 3-class models across thousands of accelerators makes gradient synchronization and activation transfer consume more wall-clock time than arithmetic	Read the fabric, collective, and checkpoint paths before adding compute
B: Recommendation at scale	Coordination	Sharding multi-terabyte DLRM embedding tables across hundreds of nodes makes sparse feature routing, shard placement, and request scheduling determine whether all-to-all traffic meets tail-latency budgets	Read placement, routing, and scheduler behavior before changing the model
C: Federated MobileNet on edge devices	Computation	Local training runs under a milliwatt power envelope, so each step is bound by device silicon rather than fleet-level bandwidth	Read the device power, memory, and duty-cycle limits before assuming cloud-scale remedies

The same fleet law and the same six principles apply to all three; what changes is which term binds first, and the engineer's job is to read that term off the workload rather than assume it.

The closing diagnostic is therefore procedural:

1. **Start with the observed symptom**: Name the user-visible or operator-visible failure.
2. **Attach the metric**: Choose the measurement that makes the symptom falsifiable.
3. **Map to C^3**: Identify whether compute, communication, or coordination is binding first.
4. **Locate the fleet-stack layer**: Find the layer that owns the intervention.
5. **State the displaced cost**: Describe what cost the proposed fix moves elsewhere.
6. **Preserve governance evidence**: Keep the audit, safety, security, or responsibility evidence needed to justify the decision.

Low MFU may point to communication or coordination rather than insufficient computation. A fairness alert may point to missing labels, insufficient review capacity, or a serving-path threshold policy rather than model weights alone. A security incident may point to artifact provenance, tool permissions, or rollback evidence rather than network perimeter failure. The discipline is to follow the constraint until the responsible layer and trade-off are visible.

17.4 Competencies Mastered

The integrated production system just assembled defines competence as engineering judgment under constraint, not as a checklist of technologies. Mastery means recognizing which layer binds, which failure mode a design invites, and which trade-off must remain visible as the system scales.

At the distributed-systems layer, the governing question is how to make a workload larger than any single machine behave as one coordinated system. An engineer who has mastered this material can orchestrate training beyond single-machine memory or compute, analyze communication patterns (recognizing that, for a gradient of size M, Ring AllReduce moves about $2M$ bytes per worker in the bandwidth term as cluster size grows), and select network architectures appropriate to the workload.

That same engineer must also derive expected failure cadence from component MTBF and fleet size. At Llama 3-class cluster sizes, Meta reported unexpected interruptions every few hours rather than every few months (Dubey et al. 2024); the general lesson is to design for routine failure. The failure mode is treating scale as aggregate capacity alone; the trade-off is between parallel speedup, communication cost, and recovery overhead.

At the production-operations layer, the governing question is how a trained model keeps serving value under live traffic and shifting distributions. The serving tax, continuous batching, and monitoring for both performance and semantic drift matter because they prevent a working model from becoming an inefficient or semantically stale service.

At the governance layer, the governing question is who bears the costs and risks of the system's decisions. Fairness, privacy, and sustainability are primary engineering constraints, and the ability to implement differential privacy, audit for bias, and schedule workloads for carbon efficiency distinguishes a systems engineer from a model developer because it treats social and environmental consequences as design requirements.

17.5 The Fleet Stack as Discipline

These competencies matter because the binding constraint will keep moving. Mastery of a particular system generation is not enough; engineers must recognize when a new scaling regime changes the unit of design. Larger models run into systems limits: network fabrics constrain synchronization, memory systems constrain state residency, energy infrastructure constrains fleet growth, and governance constrains what may be deployed. Composition is one natural pressure point, because systems of models, tools, retrieval, and verification coordinate many specialized capabilities rather than only enlarging one model. The systems lesson is not a new law of capability; it is the same fleet-stack discipline applied at a new boundary. When useful work spans multiple components, the engineer must measure orchestration overhead, state movement, failure propagation, latency budgets, and governance evidence as part of the system itself.

The same point can be stated as a simple fleet-stack accounting problem. The calculation assumes a target 100× efficiency gain, then assigns part of that gain to hardware and algorithmic compression. The residual term is the orchestration improvement the fleet must supply.



Napkin Math 17.1: Fleet-stack efficiency accounting

Problem: Suppose a workload needs a 100× efficiency improvement over a large-cluster baseline. If a particular workload saw 4× from hardware and 2.5× from algorithmic compression, where would the remaining gain have to come from?

Math: Total system gain is the product of improvements across the fleet stack.

1. **Hardware (physics)**: 4× in this scenario.
2. **Algorithm (math)**: 2.5× from workload-compatible sparsity or distillation.
3. **Orchestration (systems)**: $100\times \;/\; (4.0 \times 2.5) = 10\times$.

Systems insight: Because silicon and math can hit diminishing returns, an aggressive 100× efficiency target cannot rely on one layer alone. Here, efficiency means useful task progress per fleet resource budget, not raw FLOP/s alone. In this illustrative scenario, the remaining 10× comes from system orchestration: reducing duplicate work, routing requests to appropriate

components, reusing cached state, overlapping communication with computation, and validating outputs without exhausting the latency budget. Capability is not only in the model weights; it is also in the fleet logic.

In the fleet-stack frame, capability emerges from specialized components coordinated through the machine learning operations (MLOps) pipelines established in Chapter 12. The orchestration layer acts as a control plane that schedules model calls, retrieval, tool execution, and verification under latency, cost, and failure budgets, echoing the fleet orchestration studied in Chapter 8.

Those orchestration gains still run on a physical fabric. Once software has reduced wasted model calls, duplicate retrieval, and avoidable coordination, the remaining frontier is the substrate that moves tokens, activations, and power through the fleet. The fleet-stack lens therefore also applies to research and deployment candidates beyond conventional transistor scaling. Technologies such as optical I/O, co-packaged optics, 3D integration, and novel computing substrates may change the physics of the fleet by reducing energy per bit, shortening communication paths, increasing bandwidth density, or moving memory closer to compute. The conclusion is not that any one substrate wins. The durable requirements remain the same: bandwidth optimization, fault tolerance, and responsible governance must hold even when the substrate changes.

Transformer models partitioned across thousands of accelerators create communication pressure through several mechanisms. Tensor parallelism uses frequent collectives across partitioned tensors, while mixture-of-experts routing uses all-to-all dispatch at MoE layers. At that scale, the energy cost of driving data electrically across data center-class distances can become comparable to the cost of useful arithmetic, which is why optical interconnects are a plausible fabric-efficiency lever. The scaling pressure is not generic; it originates directly from the collective communication patterns.



Napkin Math 17.2: The physics of better fabrics

Problem: Large ML clusters running tensor parallelism and mixture-of-experts routing can hit the energy wall because collective traffic at scale makes interconnect energy a first-order cost. Using an illustrative optical-I/O target scenario, compare rough long-reach electrical signaling with a lower optical-I/O target. If we use 10 pJ/bit and 5 pJ/bit as a simple scenario, what is the efficiency dividend for the collective communication that dominates these ML workloads?

Math: One communication-efficiency leap is moving part of the fabric from electrical to optical signaling.

1. **Electrical cost**: 10 pJ/bit.
2. **Optical I/O scenario**: 5 pJ/bit.
3. **Efficiency gain**: 2×.

Systems insight: Scaling further by doing "more of the same" eventually runs into the energy wall. The wall is driven by the volume and frequency of collective operations: AllReduce for gradient synchronization and all-to-all dispatch for MoE layers grow with model and cluster size. Breaking that wall requires changing the physics of communication. Optical I/O is a meaningful fabric-efficiency lever, but this scenario illustrates single-digit energy improvements rather than orders-of-magnitude fabric-wide reductions. In the machine learning fleet, the principles of data locality and interconnect efficiency are thermodynamic requirements, not optional optimizations.

These fabric-level limits motivate one final efficiency sanity check: the machine fleet measured against the human brain. The comparison is not a biological analogy for its own sake; it tests whether orchestration, locality, and fabric efficiency are moving the engineered system toward a credible energy budget.

17.6 Engineering Intelligence at Scale

The scale of large ML infrastructure invites comparison with a highly energy-efficient biological baseline: the human brain. A rough Fermi estimate frames that comparison without treating machine FLOP/s and synaptic activity as equivalent.

Napkin Math 17.3: The Fermi estimate of intelligence

Problem: A hypothetical large cluster is compared against rough estimates of human brain synaptic activity. This Fermi-style sanity check is not a like-for-like operation metric; what is the order-of-magnitude relationship between the two?

Assumptions:

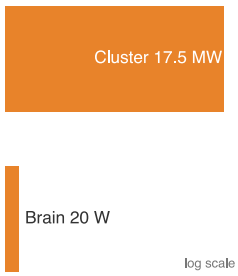
- **Machine cluster**: 25,000 H100 GPUs in a reference large-cluster scenario.
- **Machine FLOP/s**: $25{,}000 \times$ H100 FP16 tensor peak $\approx 2.47 \times 10^{19}$ FLOP/s (rounded; H100 FP16 tensor peak is 989 TFLOP/s).
- **Machine power**: $25{,}000 \times 700$ W ≈ 17.5 MW.
- **Brain synapses**: 10^{14} synapses (connections).
- **Brain firing rate**: illustrative average spike rate ≈ 1 Hz; estimates vary by neuron type and brain region, and average cortical firing is far below the 100 Hz peak rates often used in casual comparisons.
- **Brain synaptic operation rate**: $10^{14} \times 1 = 1.0 \times 10^{14}$ synaptic ops/s.

Math:

The machine-to-brain raw operation-rate ratio is:

$$\frac{\text{machine peak FLOP/s}}{\text{brain synaptic ops/s}} \approx 247{,}250\times$$

Systems insight: The comparison is useful only as a caution, not as an equivalence. Machine FLOP/s are overwhelmingly dense matrix multiplications (GEMMs) mandated by Transformer architectures: structurally rigid, synchronous operations tightly orchestrated across thousands of accelerators. Biological synaptic operations are sparse, event-driven, and massively decentralized, with no equivalent of a global barrier or gradient step. The throughput ratio measures raw arithmetic volume, not intelligence, and the architectural gap between the two operation types is as significant as the numerical gap; any efficiency comparison between a 20 W brain and a 17.5 MW cluster depends on the operation model used for the brain.



A data center draws megawatts; the brain runs on ~20 watts.

The Fermi estimate is useful precisely because it refuses a simplistic equivalence between FLOP/s and intelligence. The fleet-stack framework has established the engineering principles for scale; the continuing challenge is efficiency, applying those principles within the energy, carbon, and economic envelopes that constrain further growth.

The fleet stack provides the professional framework for engineering intelligence as a system: infrastructure powers the fleet, distributed protocols coordinate work across thousands of devices, serving systems deliver intelligence to users, and governance mandates keep the fleet aligned with security, sustainability, accountability, and explicit human-impact constraints. Large-scale intelligent systems need engineers who understand these principles and can apply them under constraint. The agenda is therefore threefold: systems that scale, systems that endure, and systems whose social and environmental obligations are engineered into the operating path.

17.7 Fallacies and Pitfalls

The closing mistakes all come from optimizing the wrong unit of design. A model can improve while the fleet becomes slower, less reliable, more expensive, or harder to govern.

Fallacy: *A faster model is automatically a better system.*

Raw model speed matters, but it is only one term in the fleet equation. A design that reduces kernel time while increasing checkpoint pressure, network contention, serving variance, or governance

risk can make the total system worse. The systems question is whether the change improves useful throughput under the actual constraints: data movement, synchronization, failure recovery, power, latency, and accountability.

Pitfall: *Optimizing one layer while hiding the constraint it creates in another.*

A local improvement becomes dangerous when it moves cost to a layer that no one is measuring. Larger batches may raise accelerator utilization while worsening tail latency. Aggressive compression may reduce communication while increasing accuracy risk. Carbon-aware scheduling may lower emissions while requiring more slack in the service budget. Good engineering respects the displacement of overhead, keeping the transferred constraint visible so the system can choose deliberately rather than inheriting a hidden bottleneck.

Fallacy: *Fleet-scale lessons are tied to today's hardware and software stack.*

Specific processors, frameworks, and serving engines will change, but the durable relationships remain. Data must reach compute at the required rate. Workers must communicate before synchronization stalls the job. Failures must be expected at large component counts. Serving systems must meet tail-latency budgets, and governance constraints must be engineered into the operating path. The technology names are examples; the rate, failure, and accountability relationships are the lesson.

Pitfall: *Treating governance and sustainability as external reviews rather than system constraints.*

Security, fairness, privacy, and carbon accounting are often postponed because they look less immediate than throughput or accuracy. At fleet scale, that postponement creates architectural debt. Retrofitting audit trails, demographic monitoring, access controls, deletion workflows, or carbon-aware placement after deployment is more expensive than budgeting for them when the serving path, data pipeline, and scheduler are designed.

17.8 Summary

At fleet scale, the central engineering object is no longer a model, accelerator, or serving endpoint in isolation. It is the coupled system that moves data, schedules work, survives failure, serves users, and satisfies governance obligations under real physical and organizational constraints. The same method recurs across the volume: identify the binding constraint, quantify the cost it imposes, and design the operating path so the constraint remains visible instead of being displaced into another layer.

Key Takeaways: Systems that scale, endure, and serve

- **The fleet is the object**: The volume's six principles reduce to one habit: follow the binding constraint across infrastructure, communication, coordination, serving, and governance. The fleet, not any single model or component, is the unit you build, measure, and optimize.
- **Scale changes the probability model**: At the 99th percentile, touching 100 servers gives a 63.4 percent chance of a slow server, and Llama 3 saw 419 unexpected interruptions in 54 days. Fleet behavior is not single-node behavior repeated.
- **Orchestration becomes capability**: The illustrative 100× efficiency target cannot come from silicon or algorithms alone; after 4× hardware and 2.5× algorithm gains, the residual 10× must come from routing, reuse, overlap, and verification.
- **Obligations belong in the path**: Security, privacy, fairness, carbon, accessibility, and auditability are production constraints, not external reviews. The discipline is to design them into data pipelines, schedulers, serving paths, and operating procedures before scale makes the trade-off irreversible.

What's Next: The discipline of ML systems

The throughline is that intelligence at scale is an emergent property of co-design, engineered through relationships rather than isolated artifacts. Models, accelerators, fabrics, data systems,

schedulers, serving paths, monitors, and governance controls form one coupled system. The durable professional skill is to follow the active constraint across that system and make the trade-off explicit before production scale makes it irreversible.

Prof. Vijay Janapa Reddi, Harvard University

Research Questions: For further inquiry

- Given a struggling fleet, how should an engineer identify the binding constraint and follow it across compute, communication, coordination, serving, and governance?
- How can a locally faster model make the total system worse by displacing cost into another layer?
- How should system design change once failures are routine rather than exceptional at fleet scale?
- Which fleet-scale principles will remain durable as today's accelerators, frameworks, and serving stacks change?

REFERENCES

References

Abadi, M., A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. 2016. "Deep Learning with Differential Privacy." *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 308–18. https://doi.org/10.1145/2976749.2978318.

Adi, Yossi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. 2018. "Turning Your Weakness into a Strength: Watermarking Deep Neural Networks by Backdooring." *27th USENIX Security Symposium (USENIX Security 18)*, 1615–31.

Agrawal, Amey, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve." *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 117–34.

AI, Meta. 2022. *Building the Most Powerful AI Supercomputer: Meta's AI Research SuperCluster*. Meta AI Blog.

Ainslie, Joshua, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 4895–901. https://doi.org/10.18653/v1/2023.emnlp-main.298.

Aizman, Alex, Gavin Maltby, and Thomas Breuel. 2019. "High Performance I/O for Large Scale Deep Learning." *2019 IEEE International Conference on Big Data (Big Data)*, 5965–67. https://doi.org/10.1109/bigdata47090.2019.9005703.

Aji, Alham Fikri, and Kenneth Heafield. 2017. "Sparse Communication for Distributed Gradient Descent." *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 440–45. https://doi.org/10.18653/v1/d17-1045.

Alexandrov, Albert D., Mihai F. Ionescu, Klaus E. Schauser, and Chris J. Scheiman. 1995. "LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation." *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures - SPAA '95* 30: 95–105. https://doi.org/10.1145/215399.215427.

Al-Fares, Mohammad, Alexander Loukissas, and Amin Vahdat. 2008. "A Scalable, Commodity Data Center Network Architecture." *ACM SIGCOMM Computer Communication Review* 38 (4): 63–74. https://doi.org/10.1145/1402946.1402967.

Alistarh, Dan, Demjan Grubic, Jerry Li 0001, Ryota Tomioka, and Milan Vojnovic. 2017. "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding." *Advances in Neural Information Processing Systems* 30: 1709–20.

Alizadeh, Mohammad, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. "Data Center TCP (DCTCP)." *Proceedings of the ACM SIGCOMM 2010 Conference*, 63–74. https://doi.org/10.1145/1851182.1851192.

Amazon Web Services. 2017. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. AWS service event summary.

Amazon Web Services. 2020. *Amazon S3 Update – Strong Read-After-Write Consistency*. AWS News Blog.

Amazon Web Services. 2026a. *Checking Object Integrity in Amazon S3*. Amazon S3 User Guide.

Amazon Web Services. 2026b. *Managing the Lifecycle of Objects*. Amazon S3 User Guide.

Amazon Web Services. 2026c. *Naming Amazon S3 Objects*. Amazon S3 User Guide.

Amazon Web Services. 2026d. *Understanding Archive Retrieval Options*. Amazon S3 User Guide.

Amdahl, Gene M. 1967. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*, AFIPS '67 (spring), 483–85. https://doi.org/10.1145/1465482.1465560.

Amershi, Saleema, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. "Software Engineering for Machine Learning: A Case Study." *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 291–300. https://doi.org/10.1109/icse-seip.2019.00042.

Amiel, Frederic, Christophe Clavier, and Michael Tunstall. 2006. "Fault Analysis of DPA-Resistant Algorithms." In *Fault Diagnosis and Tolerance in Cryptography*. Springer Berlin Heidelberg. https://doi.org/10.1007/11889700_20.

Amodei, Dario, and Danny Hernandez. 2018. "AI and Compute." *OpenAI Blog* 2.

Amodei, D., C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. 2016. "Concrete Problems in AI Safety." *arXiv Preprint arXiv:1606.06565*, ahead of print. https://doi.org/10.48550/arXiv.1606.06565.

Andrae, Anders, and Tomas Edler. 2015. "On Global Electricity Usage of Communication Technology: Trends to 2030." *Challenges* 6 (1): 117–57. https://doi.org/10.3390/challe6010117.

Anslow, Pete. 2016. *RS(544,514) FEC Performance Including Precoding*. IEEE P802.3cd Task Force.

Anthony, L. F. W., B. Kanding, and R. Selvan. 2020. *Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models*. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems.

Antonakakis, M., T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, et al. 2017. "Understanding the Mirai Botnet." *26th USENIX Security Symposium (USENIX Security 17)* 16: 1093–110.

Apple. 2021a. *Boot Process for iPad and iPhone Devices*. Apple Platform Security.

Apple. 2021b. *Secure Software Updates*. Apple Platform Security.

Apple. 2024a. *Facial Matching Security*. Apple Platform Security.

Apple. 2024b. *The Secure Enclave*. Apple Platform Security.

Apple Differential Privacy Team. 2017. *Learning with Privacy at Scale*. Apple Machine Learning Research.

Arifeen, Tooba, Abdus Sami Hassan, and Jeong-A Lee. 2020. "Approximate Triple Modular Redundancy: A Survey." *IEEE Access* 8: 139851–67. https://doi.org/10.1109/access.2020.3012673.

Arivazhagan, Manoj Ghuhan, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. 2019. "Federated Learning with Personalization Layers." *CoRR* abs/1912.00818.

Ashkboos, Saleh, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. "QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs." *Advances in Neural Information Processing Systems (NeurIPS)*, 100213–40. https://doi.org/10.52202/079017-3180.

Asonov, Dmitri, and Rakesh Agrawal. 2004. "Keyboard Acoustic Emanations." *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 3–11. https://doi.org/10.1109/secpri.2004.1301311.

Ateniese, G., L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici. 2015. "Hacking Smart Machines with Smarter Ones: How to Extract Meaningful Data from Machine Learning Classifiers." *International Journal of Security and Networks* 10 (3): 137. https://doi.org/10.1504/ijsn.2015.071829.

Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11–33. https://doi.org/10.1109/TDSC.2004.2.

Aygun, Sercan, Ece Olcay Gunes, and Christophe De Vleeschouwer. 2021. "Efficient and Robust Bitstream Processing in Binarised Neural Networks." *Electronics Letters* 57 (5): 219–22. https://doi.org/10.1049/ell2.12045.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2015. "Neural Machine Translation by Jointly Learning to Align and Translate." *International Conference on Learning Representations (ICLR)*.

Bai, Tao, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. 2021. "Recent Advances in Adversarial Training for Adversarial Robustness." *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, 4312–21. https://doi.org/10.24963/ijcai.2021/591.

Bai, Yuntao, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, et al. 2022. *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*.

Baldé, Cornelis P, Vanessa Forti, Vanessa Gray, Ruediger Kuehr, and Paul Stegmann. 2017. "The Global e-Waste Monitor 2017: Quantities, Flows and Resources." *United Nations University, International Telecommunication Union, International Solid Waste Association*.

Banbury, Colby, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, et al. 2021. "MLPerf Tiny Benchmark." *arXiv Preprint*.

Bannon, Pete, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. "Computer and Redundancy Solution for the Full Self-Driving Computer." *2019 IEEE Hot Chips 31 Symposium (HCS)*, 1–22. https://doi.org/10.1109/hotchips.2019.8875645.

Barenghi, A., G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. 2010. "Low Voltage Fault Attacks to AES." *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 7–12. https://doi.org/10.1109/hst.2010.5513121.

Barroso, Luiz André, Urs Hölzle, and Parthasarathy Ranganathan. 2019. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Springer International Publishing. https://doi.org/10.1007/978-3-031-01761-2.

Basiri, A., N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. 2016. "Chaos Engineering." *IEEE Software* 33 (3): 35–41. https://doi.org/10.1109/ms.2016.60.

Baumann, R. 2005. "Soft Errors in Advanced Computer Systems." *IEEE Design and Test of Computers* 22 (3): 258–66. https://doi.org/10.1109/mdt.2005.69.

Baylor, Denis, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, et al. 2017. "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform." *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1387–95. https://doi.org/10.1145/3097983.3098021.

BBC Future. 2022. *How Space Weather Causes Computer Errors*. BBC Future.

Beaton, A. E., and J. W. Tukey. 1974. "The Fitting of Power Series, Meaning Polynomials, Illustrated on Band-Spectroscopic Data." *Technometrics* 16 (2): 147–85. https://doi.org/10.1080/00401706.1974.10489171.

Bellamy, Rachel K. E., Kuntal Dey, Michael Hind, Seung Hoffman, Sylvain Houde, Karthikeyan Kannan, Pradeep Lohia, et al. 2019. "AI Fairness 360: An Extensible Toolkit for Detecting and Mitigating Algorithmic Bias." *IBM Journal of Research and Development* 63 (4/5): 4:1–15. https://doi.org/10.1147/jrd.2019.2942287.

Bender, E. M., T. Gebru, A. McMillan-Major, and S. Shmitchell. 2021. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–23. https://doi.org/10.1145/3442188.3445922.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv Preprint arXiv:1308.3432*.

Ben-Nun, Tal, and Torsten Hoefler. 2019. "Demystifying Parallel and Distributed Deep Learning: An in-Depth Concurrency Analysis." *ACM Computing Surveys* 52 (4): 1–43. https://doi.org/10.1145/3320060.

Berger, V. W., and Y. Zhou. 2014. "Wiley StatsRef: Statistics Reference Online." In *Wiley Statsref: Statistics Reference Online*. Wiley. https://doi.org/10.1002/9781118445112.stat06558.

Bernstein, Jeremy, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. "signSGD: Compressed Optimisation for Non-Convex Problems." *Proceedings of the 35th International Conference on Machine Learning* 80: 560–69.

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media.

Bhagoji, Arjun Nitin, Warren He, Bo Li, and Dawn Song. 2018. "Practical Black-Box Attacks on Deep Neural Networks Using Efficient Query Mechanisms." In *Computer Vision – ECCV 2018*. Springer. https://doi.org/10.1007/978-3-030-01258-8_10.

Biega, A. J., P. Potash, H. Daumé, F. Diaz, and M. Finck. 2020. "Operationalizing the Legal Principle of Data Minimization for Personalization." In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, edited by Jimmy Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu. ACM. https://doi.org/10.1145/3397271.3401034.

Biggio, Battista, Blaine Nelson, and Pavel Laskov. 2012. "Poisoning Attacks Against Support Vector Machines." *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*.

Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. 2011. "The Gem5 Simulator." *ACM SIGARCH Computer Architecture News* 39 (2): 1–7. https://doi.org/10.1145/2024716.2024718.

Bird, Sarah, Miro Dudík, Richard Edgar, Brandon Horn, Roman Lutz, Vanessa Milan, Mehrnoosh Sameki, Hanna Wallach, and Kathleen Walker. 2020. "Fairlearn: A Toolkit for Assessing and Improving Fairness in AI." *Microsoft Technical Report MSR-TR-2020-32*.

Birolini, A. 2017. *Reliability Engineering: Theory and Practice*. 8th ed. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-54209-5.

Bisong, E. 2019. "Kubeflow and Kubeflow Pipelines." Kubeflow and Kubeflow Pipelines in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Apress. https://doi.org/10.1007/978-1-4842-4470-8_46.

Black, James R. 1969. "Electromigration–a Brief Survey and Some Recent Results." *IEEE Transactions on Electron Devices* 16 (4): 338–47. https://doi.org/10.1109/T-ED.1969.16754.

Blanchard, Peva, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. "Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent." *Advances in Neural Information Processing Systems*, 119–29.

Blog, Netflix Technology. 2017. *Innovating Faster on Personalization Algorithms at Netflix Using Interleaving*. Netflix Tech Blog.

Bommasani, Rishi, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, et al. 2021. "On the Opportunities and Risks of Foundation Models." *arXiv Preprint arXiv:2108.07258*.

Bonawitz, K., H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, et al. 2019. "Towards Federated Learning at Scale: System Design." *Proceedings of Machine Learning and Systems* 3.

Bonawitz, Keith, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. "Practical Secure Aggregation for Privacy-Preserving Machine Learning." *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 1175–91. https://doi.org/10.1145/3133956.3133982.

Bourtoule, Lucas, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. 2021. "Machine Unlearning." *2021 IEEE Symposium on Security and Privacy (SP)*, 141–59. https://doi.org/10.1109/sp40001.2021.00019.

Breier, Jakub, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. 2018. "Practical Fault Attack on Deep Neural Networks." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* abs/1806.05859: 2204–6. https://doi.org/10.1145/3243734.3278519.

Brewer, Eric A. 2000. "Towards Robust Distributed Systems (Abstract)." *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, 7. https://doi.org/10.1145/343477.343502.

Broadcom Inc. 2025. *Broadcom Ships Tomahawk 6: World's First 102.4 Tbps Switch*. Broadcom press release.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33: 1877–901. https://doi.org/10.48550/arxiv.2005.14165.

Bu, Zhiqi, Jiawen Dong, Qi Long, and Weijie J. Su. 2020. "Deep Learning with Gaussian Differential Privacy." *Harvard Data Science Review* 2. https://doi.org/10.1162/99608f92.cfc5dd25.

Buolamwini, Joy, and Timnit Gebru. 2018. "Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification." *Conference on Fairness, Accountability and Transparency*, 77–91.

Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. "Borg, Omega, and Kubernetes." *Communications of the ACM* 59 (5): 50–57. https://doi.org/10.1145/2890784.

Bursztein, Elie, Luca Invernizzi, Karel Král, Daniel Moghimi, Jean-Michel Picod, and Marina Zhang. 2024. "Generalized Power Attacks Against Crypto Hardware Using Long-Range Deep Learning." *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024 (3): 472–99. https://doi.org/10.46586/tches.v2024.i3.472-499.

Bursztein, Elie, and Jean-Michel Picod. 2019. "A Hacker Guide to Deep Learning Based Side Channel Attacks." In *DEF CON 27*, edited by DEF CON. <Https://elie.net/talk/a-hackerguide-to-deep-learning-based-side-channel-attacks/>; DEF CON.

Bushnell, Michael L, and Vishwani D Agrawal. 2002. "Built-in Self-Test." *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits* 1: 489–548. https://doi.org/10.1007/0-306-47040-3_15.

Buyya, Rajkumar, Anton Beloglazov, and Jemal Abawajy. 2010. "Energy-Efficient Management of Data Center Resources for Cloud Computing: A Vision, Architectural Elements, and Open Challenges." *arXiv Preprint arXiv:1006.0308*, 6–17. https://doi.org/10.48550/arXiv.1006.0308.

Cai, Han, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. "Once-for-All: Train One Network and Specialize It for Efficient Deployment." *International Conference on Learning Representations*.

Cai, Han, Chuang Gan, Ligeng Zhu, and Song Han. 2020. "TinyTL: Reduce Activations, Not Trainable Parameters for Efficient on-Device Learning." *Advances in Neural Information Processing Systems* 33: 11285–97.

Cai, Han, Ligeng Zhu, and Song Han. 2019. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware." *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Cai, Tianle, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. "Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads." *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Calders, Toon, and Sicco Verwer. 2010. "Three Naive Bayes Approaches for Discrimination-Free Classification." *Data Mining and Knowledge Discovery* 21 (2): 277–92. https://doi.org/10.1007/s10618-010-0190-x.

California Legislature. 2023. *California Consumer Privacy Act of 2018*. California Legislative Information.

Calvo, Rafael A., Dorian Peters, Karina Vold, and Richard M. Ryan. 2020. "Supporting Human Autonomy in AI Systems: A Framework for Ethical Enquiry." In *Ethics of Digital Well-Being*. Springer. https://doi.org/10.1007/978-3-030-50585-1_2.

Cao, Yinzhi, and Junfeng Yang. 2015. "Towards Making Systems Forget with Machine Unlearning." *2015 IEEE Symposium on Security and Privacy*, 463–80. https://doi.org/10.1109/sp.2015.35.

Carlini, N., P. M. 0001, T. Vaidya, Y. Z. 0001, M. Sherr, C. Shields, D. A. W. 0001, and W. Zhou. 2016. "Hidden Voice Commands." *25th USENIX Security Symposium (USENIX Security 16)*, 513–30.

Carlini, Nicholas, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramèr, Borja Balle, Daphne Ippolito, and Eric Wallace. 2023. "Extracting Training Data from Diffusion Models." *32nd USENIX Security Symposium (USENIX Security 23)*, 5253–70.

Carlini, Nicholas, and David Wagner. 2017. "Towards Evaluating the Robustness of Neural Networks." *2017 IEEE Symposium on Security and Privacy (SP)*, 39–57. https://doi.org/10.1109/sp.2017.49.

Carlini, N., D. Paleka, K. D. Dvijotham, T. Steinke, J. Hayase, A. F. Cooper, K. Lee, et al. 2024. "Stealing Part of a Production Language Model." *arXiv Preprint arXiv:2403.06634*.

Carlini, N., F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, et al. 2021. "Extracting Training Data from Large Language Models." *30th USENIX Security Symposium (USENIX Security 21)*, 2633–50.

Caveness, E., P. S. G. C., Z. Peng, N. Polyzotis, S. Roy, and M. Zinkevich. 2020. "TensorFlow Data Validation: Data Analysis and Validation in Continuous ML Pipelines." *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2793–96. https://doi.org/10.1145/3318464.3384707.

Cavoukian, A. 2012. "Privacy by Design: Origins, Meaning, and Prospects for Assuring Privacy and Trust in the Information Era." *Office of the Information and Privacy Commissioner* 1: 170–208. https://doi.org/10.4018/978-1-61350-501-4.ch007.

Cenci, Marcelo Pilotto, Tatiana Scarazzato, Daniel Dotto Munchen, Paula Cristina Dartora, Hugo Marcelo Veit, Andrea Moura Bernardes, and Pablo R. Dias. 2021. "Eco-Friendly Electronics—a Comprehensive Review." *Advanced Materials Technologies* 7 (2): 2001263. https://doi.org/10.1002/admt.202001263.

Center, Pew Research. 2023. *What Americans Know about AI, Cybersecurity and Big Tech*.

Centers, Google Data. 2023. *Efficiency: How We Do It*.

Centre for International Governance Innovation. 2023. *Who Is Responsible When Autonomous Systems Fail?* CIGI policy article.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. 2009. "Anomaly Detection: A Survey." *ACM Computing Surveys* 41 (3): 1–58. https://doi.org/10.1145/1541880.1541882.

Chapelle, Olivier, Thorsten Joachims, Filip Radlinski, and Yisong Yue. 2012. "Large-Scale Validation and Analysis of Interleaved Search Evaluation." *ACM Transactions on Information Systems* 30 (1): 1–41. https://doi.org/10.1145/2094072.2094078.

Chen, C., S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper. 2023. "Accelerating Large Language Model Decoding with Speculative Sampling." *arXiv Preprint arXiv:2302.01318*.

Chen, Chaofan, Oscar Li, Daniel Tao, Alina Barnett, Cynthia Rudin, and Jonathan Su. 2019. "This Looks Like That: Deep Learning for Interpretable Image Recognition." In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, edited by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. Curran Associates.

Chen, H.-W. 2006. "Gallium, Indium, and Arsenic Pollution of Groundwater from a Semiconductor Manufacturing Area of Taiwan." *Bulletin of Environmental Contamination and Toxicology* 77 (2): 289–96. https://doi.org/10.1007/s00128-006-1062-3.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *arXiv Preprint arXiv:1604.06174*.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. "A Simple Framework for Contrastive Learning of Visual Representations." *Proceedings of the 37th International Conference on Machine Learning*, ICML'20, vol. 119: 1597–607.

Chen, Zitao, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2020. "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications." *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 426–35. https://doi.org/10.1109/issre5003.2020.00047.

Choquette, Jack. 2023. "NVIDIA Hopper H100 GPU: Scaling Performance." *IEEE Micro* 43 (3): 9–17. https://doi.org/10.1109/mm.2023.3256796.

Chouldechova, Alexandra. 2017. "Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments." *Big Data* 5 (2): 153–63. https://doi.org/10.1089/big.2016.0047.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, et al. 2022. "PaLM: Scaling Language Modeling with Pathways." *arXiv Preprint arXiv:2204.02311*.

Christiano, P. F., J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, edited by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. Curran Associates.

Chung, Jae-Won, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. 2023. "Reducing Energy Bloat in Large Model Training." *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* abs/2312.06902: 144–59. https://doi.org/10.1145/3694715.3695970.

Clos, C. 1953. "A Study of Non-Blocking Switching Networks." *Bell System Technical Journal* 32 (2): 406–24. https://doi.org/10.1002/j.1538-7305.1953.tb01433.x.

Cloud Native Computing Foundation. 2024. *Kubernetes: Production-Grade Container Orchestration*.

Cloudflare. 2019. *Details of the Cloudflare Outage on July 2, 2019*. Cloudflare incident report.

CNBC. 2023a. *California DMV Suspends Cruise's Self-Driving Car Permits*. CNBC news article.

CNBC. 2023b. *Cruise Under NHTSA Probe into Autonomous Driving Pedestrian Injuries*. CNBC news article.

Cohen, Jeremy, Elan Rosenfeld, and Zico Kolter. 2019. "Certified Adversarial Robustness via Randomized Smoothing." *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 97: 1310–20.

Constantinescu, C. 2008. "Intermittent Faults and Effects on Reliability of Integrated Circuits." *2008 Annual Reliability and Maintainability Symposium*, 370–74. https://doi.org/10.1109/rams.2008.4925824.

Cooper, Tom, Suzanne Fallender, Joyann Pafumi, Jon Dettling, Sebastien Humbert, and Lindsay Lessard. 2011. "A Semiconductor Company's Examination of Its Water Footprint Approach." *Proceedings of the 2011 IEEE International Symposium on Sustainable Systems and Technology*, 1–6. https://doi.org/10.1109/issst.2011.5936865.

coreboot contributors. 2026a. *coreboot*.

coreboot contributors. 2026b. *vboot: Verified Boot Support*.

Courbariaux, Matthieu, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1." *arXiv Preprint arXiv:1602.02830*.

Covington, Paul, Jay Adams, and Emre Sargin. 2016. "Deep Neural Networks for YouTube Recommendations." *Proceedings of the 10th ACM Conference on Recommender Systems*, 191–98. https://doi.org/10.1145/2959100.2959190.

Culler, David, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. "LogP: Towards a Realistic Model of Parallel Computation." *ACM SIGPLAN Notices* 28 (7): 1–12. https://doi.org/10.1145/173284.155333.

Cyberspace Administration of China, National Development and Reform Commission, Ministry of Education, Ministry of Science and Technology, Ministry of Industry and Information Technology, Ministry of Public Security, and National Radio and Television Administration. 2023. *Interim Measures for Administration of Generative Artificial Intelligence Services*. State Council Gazette.

Daly, J. T. 2006. "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps." *Future Generation Computer Systems* 22 (3): 303–12. https://doi.org/10.1016/j.future.2004.11.016.

Dao, T., D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. 2022. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness." *Advances in Neural Information Processing Systems 35* 35: 16344–59. https://doi.org/10.52202/068431-1189.

Dao, Tri. 2023. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." *arXiv Preprint arXiv:2307.08691*.

Dastin, Jeffrey. 2018. "Amazon Scraps Secret AI Recruiting Tool That Showed Bias Against Women." *Reuters*.

Databricks. 2026. *Mosaic Streaming: Main Concepts*. Databricks Mosaic AI Training Documentation.

Dayarathna, Miyuru, Yonggang Wen, and Rui Fan. 2016. "Data Center Energy Consumption Modeling: A Survey." *IEEE Communications Surveys &Amp; Tutorials* 18 (1): 732–94. https://doi.org/10.1109/comst.2015.2481183.

De, Soham, Leonard Berrada, Jamie Hayes, Samuel L. Smith, and Borja Balle. 2022. "Unlocking High-Accuracy Differentially Private Image Classification Through Scale." *arXiv Preprint arXiv:2204.13650*.

Dean, Jeff. 2024. *Exciting Directions in Systems for Machine Learning*. MLSys 2024 invited talk.

Dean, Jeffrey, and Luiz André Barroso. 2013. "The Tail at Scale." *Communications of the ACM* 56 (2): 74–80. https://doi.org/10.1145/2408776.2408794.

Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen 0010, Matthieu Devin, Quoc V. Le, Mark Z. Mao, et al. 2012. "Large Scale Distributed Deep Networks." In *Advances in Neural Information Processing Systems (NeurIPS)*, edited by Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, vol. 25. Curran Associates.

Dean, Jeffrey, and Sanjay Ghemawat. 2004. "MapReduce: Simplified Data Processing on Large Clusters." *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 137–50.

DeepSpeed Developers. 2026a. *DeepSpeed Model Checkpointing*. DeepSpeed documentation.

DeepSpeed Developers. 2026b. *DeepSpeed Universal Checkpointing with DeepSpeed: A Practical Guide*. DeepSpeed tutorial.

Dell Technologies. 2026. *Data Center Power and Cooling Solutions*.

Dennard, Robert H., Frank H. Gaensslen, Hwa-Nien Yu, Victor L. Rideout, Elias Bassous, and Antoine R. LeBlanc. 1974. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions." *IEEE J. Solid-State Circuits* 9 (5): 256–68. https://doi.org/10.1109/jssc.1974.1050511.

Department for Education, and Ofqual. 2020. *GCSE and A Level Students to Receive Centre Assessment Grades*. GOV.UK News Story.

Dettmers, Tim, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. "LLM.int8(): 8-Bit Matrix Multiplication for Transformers at Scale." *Advances in Neural Information Processing Systems 35*, 30318–32. https://doi.org/10.52202/068431-2198.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *Proceedings of the 2019 Conference of the North*, 4171–86. https://doi.org/10.18653/v1/n19-1423.

Diao, Enmao, Jie Ding, and Vahid Tarokh. 2023. "Pruning and Sparse Training for on-Device Neural Network Optimization." *IEEE Transactions on Mobile Computing* 22 (8): 4567–80.

Dixit, H. D., S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar. 2021. *Silent Data Corruptions at Scale*. arXiv preprint arXiv:2102.11245.

Dodge, Jesse, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A. Smith, Nicole DeCario, and Will Buchanan. 2022. "Measuring the Carbon Intensity of AI in Cloud Instances." *2022 ACM Conference on Fairness Accountability and Transparency*, 1877–94. https://doi.org/10.1145/3531146.3533234.

Du, Nan, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, et al. 2022. "GLaM: Efficient Scaling of Language Models with Mixture-of-Experts." In *Proceedings of the 39th International Conference on Machine Learning*, edited by Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, vol. 162. Proceedings of Machine Learning Research. PMLR.

Dubey, Abhimanyu, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. 2024. *The Llama 3 Herd of Models*. arXiv preprint arXiv:2407.21783.

Dutta, Sanghamitra, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. 2018. "Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-Offs in Distributed SGD." *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics* 84: 803–12.

Dwork, Cynthia, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. "Fairness Through Awareness." *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 214–26. https://doi.org/10.1145/2090236.2090255.

Dwork, Cynthia, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. "Calibrating Noise to Sensitivity in Private Data Analysis." In *Theory of Cryptography Conference (TCC)*, edited by Shai Halevi and Tal Rabin, vol. 3876. Lecture Notes in Computer Science. Springer Berlin Heidelberg. https://doi.org/10.1007/11681878_14.

Dwork, Cynthia, and Aaron Roth. 2014. "The Algorithmic Foundations of Differential Privacy." *Foundations and Trends® in Theoretical Computer Science*, Foundations and trends in theoretical computer science, vol. 9 (3-4): 211–487. https://doi.org/10.1561/0400000042.

Ebrahimi, K., G. F. Jones, and A. S. Fleischer. 2014. "A Review of Data Center Cooling Technology, Operating Conditions and the Corresponding Low-Grade Waste Heat Recovery Opportunities." *Renewable and Sustainable Energy Reviews* 31: 622–38. https://doi.org/10.1016/j.rser.2013.12.007.

Eckles, Dean, Brian Karrer, and Johan Ugander. 2017. "Design and Analysis of Experiments in Networks: Reducing Bias from Interference." *Journal of Causal Inference* 5 (1): 1–23. https://doi.org/10.1515/jci-2015-0021.

Egwutuoha, I. P., D. Levy, B. Selic, and S. Chen. 2013. "A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems." *The Journal of Supercomputing* 65 (3): 1302–26. https://doi.org/10.1007/s11227-013-0884-0.

Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter. 2019. "Neural Architecture Search." In *The Springer Series on Challenges in Machine Learning*, vol. 20. Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5_3.

Engineering, M. 2016. *Introducing FBLearner Flow: Facebook's AI Backbone*. Engineering at Meta Blog.

Engineering, S. 2019. *The Winding Road to Better Machine Learning Infrastructure Through TensorFlow Extended and Kubeflow*. Spotify Engineering Blog.

Environment and Climate Change Canada. 2026. *Emission Factors and Reference Values*.

Erlang, Agner Krarup. 1909. "The Theory of Probabilities and Telephone Conversations." *Nyt Tidsskrift for Matematik B* 20: 33–39.

Ethernet Alliance. 2025. *2025 Ethernet Roadmap*. Ethernet Alliance roadmap.

European Data Protection Board. 2018. *Guidelines on Automated Individual Decision-Making and Profiling for the Purposes of Regulation 2016/679*.

European Parliament and Council of the European Union. 2024. *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 Laying down Harmonised Rules on Artificial Intelligence (AI Act)*. Official Journal of the European Union, L 2024/1689.

European Parliament, and Council of the European Union. 2016. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016*. Official Journal of the European Union.

Evans, Richard, and Jim Gao. 2016. *DeepMind AI Reduces Google Data Centre Cooling Bill by 40%*. DeepMind Blog.

Executive Office of the President. 2023. *Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence*. Executive Order 14110, 88 FR 75191.

Executive Office of the President. 2025a. *Initial Rescissions of Harmful Executive Orders and Actions*. Executive Order 14148, 90 FR 8237.

Executive Office of the President. 2025b. *Removing Barriers to American Leadership in Artificial Intelligence*. Executive Order 14179.

Executive Office of the President. 2026. *Promoting Advanced Artificial Intelligence Innovation and Security*. Executive Order 14409, 91 FR 34565.

Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2017. "Robust Physical-World Attacks on Deep Learning Models." *ArXiv Preprint* abs/1707.08945 (July).

Eykholt, K., I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. 2018. "Robust Physical-World Attacks on Deep Learning Visual Classification." *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* 1707.08945: 1625–34. https://doi.org/10.1109/cvpr.2018.00175.

Farwell, J. P., and R. Rohozinski. 2011. "Stuxnet and the Future of Cyber War." *Survival* 53 (1): 23–40. https://doi.org/10.1080/00396338.2011.555586.

Federal Aviation Administration. 2015. *Airworthiness Directive: Boeing 787 Generator Control Units Failsafe Reset*. FAA Airworthiness Directive 2015-10066.

Federal Trade Commission. 2021. *Nixing the Fix: An FTC Report to Congress on Repair Restrictions*. Federal Trade Commission.

Fedus, William, Barret Zoph, and Noam Shazeer. 2022. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." *Journal of Machine Learning Research* 23 (120): 1–39.

Feldman, M., S. A. Friedler, J. Moeller, C. Scheidegger, and S. Venkatasubramanian. 2015. "Certifying and Removing Disparate Impact." *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 259–68. https://doi.org/10.1145/2783258.2783311.

Finn, Chelsea, Pieter Abbeel, and Sergey Levine. 2017. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks." *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

Forti, V., C. P. Baldé, R. Kuehr, and G. Bel. 2020. *The Global e-Waste Monitor 2020: Quantities, Flows and the Circular Economy Potential*. United Nations University/United Nations Institute for Training; Research, International Telecommunication Union,; International Solid Waste Association.

Francalanza, Adrian, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. "A Foundation for Runtime Monitoring." In *Runtime Verification*. Springer International Publishing. https://doi.org/10.1007/978-3-319-67531-2_2.

Frantar, Elias, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers." In *CoRR*, abs/2210.17323. https://doi.org/10.48550/ARXIV.2210.17323.

Fredrikson, Matt, Somesh Jha, and Thomas Ristenpart. 2015. "Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures." *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1322–33. https://doi.org/10.1145/2810103.2813677.

Friedman, B. 1996. "Value-Sensitive Design." *Interactions* 3 (6): 16–23. https://doi.org/10.1145/242485.242493.

Gal, Yarin, and Zoubin Ghahramani. 2016. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." *Proceedings of the 33rd International Conference on Machine Learning (ICML)* 48: 1050–59.

Gama, João, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. "A Survey on Concept Drift Adaptation." *ACM Computing Surveys* 46 (4): 1–37. https://doi.org/10.1145/2523813.

Gandolfi, Karine, Christophe Mourtel, and Francis Olivier. 2001. "Electromagnetic Analysis: Concrete Results." In *Cryptographic Hardware and Embedded Systems — CHES 2001*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44709-1_21.

Gangidi, Adi, Rui Miao, Sandeep Hebbani, Gaya Nagarajan, Omar Baldonado, Lixin Gao, Hany Morsy Goes, et al. 2024. "RDMA over Ethernet for Distributed AI Training at Meta Scale." *Proceedings of the ACM SIGCOMM 2024 Conference*, 56–69. https://doi.org/10.1145/3651890.3672233.

Gao, J. 2014. *Machine Learning Applications for Data Center Optimization*. Google; Google White Paper.

Gao, Y., S. F. Al-Sarawi, and D. Abbott. 2020. "Physical Unclonable Functions." *Nature Electronics* 3 (2): 81–91. https://doi.org/10.1038/s41928-020-0372-5.

Gassend, Blaise, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. 2002. "Silicon Physical Random Functions." *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 148–60. https://doi.org/10.1145/586110.586132.

Gebru, Timnit, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. 2021. "Datasheets for Datasets." *Communications of the ACM* 64 (12): 86–92. https://doi.org/10.1145/3458723.

Geiger, Atticus, Hanson Lu, Thomas Icard, and Christopher Potts. 2021. "Causal Abstractions of Neural Networks." In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, Virtual*, edited by Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan. Curran Associates.

Genkin, Daniel, Adi Shamir, and Eran Tromer. 2017. "Acoustic Cryptanalysis." *Journal of Cryptology* 30: 392–443. https://doi.org/10.1007/s00145-015-9224-2.

Gentry, C. 2009. "Fully Homomorphic Encryption Using Ideal Lattices." *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, 169–78. https://doi.org/10.1145/1536414.1536440.

Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 323–36.

Gholami, Amir, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. "A Survey of Quantization Methods for Efficient Neural Network Inference." *arXiv Preprint arXiv:2103.13630* abs/2103.13630: 291–326. https://doi.org/10.1201/9781003162810-13.

Gibiansky, A. 2017. *Bringing HPC Techniques to Deep Learning*. Baidu Research Technical Blog.

Gilbert, Seth, and Nancy Lynch. 2002. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." *ACM SIGACT News* 33 (2): 51–59. https://doi.org/10.1145/564585.564601.

GitLab. 2017. *Postmortem of Database Outage of January 31*. GitLab postmortem.

Gnad, D. R. E., F. Oboril, and M. B. Tahoori. 2017. "Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams." *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 1–7. https://doi.org/10.23919/fpl.2017.8056840.

Gomez-Uribe, Carlos A., and Neil Hunt. 2015. "The Netflix Recommender System: Algorithms, Business Value, and Innovation." *ACM Transactions on Management Information Systems* 6 (4): 1–19. https://doi.org/10.1145/2843948.

Goncalves, A., P. Ray, B. Soper, J. Stevens, L. Coyle, and A. P. Sales. 2020. "Generation and Evaluation of Synthetic Patient Data." *BMC Medical Research Methodology* 20 (1): 1–40. https://doi.org/10.1186/s12874-020-00977-1.

Goodfellow, I. J., J. Shlens, and C. Szegedy. 2014. "Explaining and Harnessing Adversarial Examples." *ICLR* 3.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Google. 2024. *ShieldGemma: Generative AI Content Moderation Based on Gemma*. Google AI for Developers documentation.

Google Cloud. 2026. *About Cloud Storage Objects*. Google Cloud Documentation.

Government of Ireland. 2022. *Government Statement on the Role of Data Centres in Ireland's Enterprise Strategy*. Government policy statement.

Goyal, Priya, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." *arXiv Preprint arXiv:1706.02677* abs/1706.02677.

Gräfe, Ralf, Qutub Syed Sha, Florian Geissler, and Michael Paulitsch. 2023. "Large-Scale Application of Fault Injection into PyTorch Models -an Extension to PyTorchFI for Validation Efficiency." *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-s)*, 56–62. https://doi.org/10.1109/dsn-s58398.2023.00025.

Graham, R. L., D. Bureddy, P. Lui, H. Rober, G. Bloch, G. Shainer, J. Poole, et al. 2020. "Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Streaming-Aggregation Hardware Design and Evaluation." *International Conference on High Performance Computing*, 41–59. https://doi.org/10.1007/978-3-030-50743-5_3.

Gray, J. N. 1978. "Notes on Data Base Operating Systems." In *Operating Systems: An Advanced Course*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-08755-9_9.

Greshake, Kai, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. "Not What You've Signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection." *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 79–90. https://doi.org/10.1145/3605764.3623985.

Grossman, E. 2007. *High Tech Trash: Digital Devices, Hidden Toxics, and Human Health*. Island press.

Gu, J., M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo. 2019. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning." *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 485–500.

Gu, Tianyu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. "BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain." *arXiv Preprint arXiv:1708.06733*.

Guo, Chuanxiong, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, and Marina Lipshteyn. 2016. "RDMA over Commodity Ethernet at Scale." *Proceedings of the 2016 ACM SIGCOMM Conference*, 202–15. https://doi.org/10.1145/2934872.2934908.

Gupta, Maanak, Charankumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. 2023. "From ChatGPT to ThreatGPT: Impact of Generative AI in Cybersecurity and Privacy." *IEEE Access* 11: 80218–45. https://doi.org/10.1109/access.2023.3300381.

Gupta, Udit, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2022. "Chasing Carbon: The Elusive Environmental Footprint of Computing." *IEEE Micro* 42 (4): 37–47. https://doi.org/10.1109/MM.2022.3163226.

Gupta, Udit, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, et al. 2020. "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation." *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047.

Gustafson, John L. 1988. "Reevaluating Amdahl's Law." *Communications of the ACM* 31 (5): 532–33. https://doi.org/10.1145/42411.42415.

Hamming, R. W. 1950. "Error Detecting and Error Correcting Codes." *Bell System Technical Journal* 29 (2): 147–60. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x.

Han, Song, Jeff Pool, John Tran, and William J. Dally. 2015. "Learning Both Weights and Connections for Efficient Neural Networks." *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*, 1135–43.

Hard, Andrew, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. "Federated Learning for Mobile Keyboard Prediction." *arXiv Preprint arXiv:1811.03604*.

Hardt, Moritz, Eric Price, and Nathan Srebro. 2016. "Equality of Opportunity in Supervised Learning." *Advances in Neural Information Processing Systems* 29: 3315–23.

Hayes, T. L., K. Kafle, R. Shrestha, M. Acharya, and C. Kanan. 2020. "REMIND Your Neural Network to Prevent Catastrophic Forgetting." In *Computer Vision – ECCV 2020*. Springer. https://doi.org/10.1007/978-3-030-58598-3_28.

Hazelwood, Kim, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, et al. 2018. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective." *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 620–29. https://doi.org/10.1109/hpca.2018.00059.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. "Deep Residual Learning for Image Recognition." *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–78. https://doi.org/10.1109/cvpr.2016.90.

He, K., X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. 2021. "Masked Autoencoders Are Scalable Vision Learners." *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* 18: 15979–88. https://doi.org/10.1109/cvpr52688.2022.01553.

He, Yi, Prasanna Balaprakash, and Yanjing Li. 2020. "FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators." *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 270–81. https://doi.org/10.1109/micro50266.2020.00033.

He, Yi, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. "Understanding and Mitigating Hardware Failures in Deep Learning Training Systems." *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 1–16. https://doi.org/10.1145/3579371.3589105.

Hébert-Johnson, U., M. P. Kim, O. Reingold, and G. N. Rothblum. 2018. "Multicalibration: Calibration for the (Computationally-Identifiable) Masses." In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, edited by Jennifer G. Dy and Andreas Krause, vol. 80. Proceedings of Machine Learning Research. PMLR.

Henderson, Peter, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." *CoRR* abs/2002.05651 (248): 1–43. https://doi.org/10.48550/arxiv.2002.05651.

Hendrycks, Dan, and Thomas Dietterich. 2019. "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations." *arXiv Preprint arXiv:1903.12261*.

Hennessy, J. L., and D. A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

Hennessy, John L., and David A. Patterson. 2019. "A New Golden Age for Computer Architecture." *Communications of the ACM* 62 (2): 48–60. https://doi.org/10.1145/3282307.

Hermann, Jeremy, and Mike Del Balso. 2017. *Meet Michelangelo: Uber's Machine Learning Platform*. Uber Engineering Blog.

Himmelstein, Gracie, David Bates, and Li Zhou. 2022. "Examination of Stigmatizing Language in the Electronic Health Record." *JAMA Network Open* 5 (1): e2144967. https://doi.org/10.1001/jamanetworkopen.2021.44967.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. 2012. "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors." *arXiv Preprint arXiv:1207.0580*.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. "Distilling the Knowledge in a Neural Network." *arXiv Preprint*.

Ho, Qirong, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. 2013. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server." *Advances in Neural Information Processing Systems* 26.

Hockney, Roger W. 1994. "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2." *Parallel Computing* 20 (3): 389–98. https://doi.org/10.1016/s0167-8191(06)80021-9.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, et al. 2022. "Training Compute-Optimal Large Language Models." *Advances in Neural Information Processing Systems 35* 35: 30016–30. https://doi.org/10.52202/068431-2176.

Horovod Developers. 2026. *Elastic Horovod*. Horovod documentation.

Horowitz, Mark. 2014. "1.1 Computing's Energy Problem (and What We Can Do about It)." *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 10–14. https://doi.org/10.1109/isscc.2014.6757323.

Hosseini, Hossein, Sreeram Kannan, Baosen Zhang, and Radha Poovendran. 2017. "Deceiving Google's Perspective API Built for Detecting Toxic Comments." *arXiv Preprint arXiv:1702.08138*, ahead of print. https://doi.org/10.48550/arXiv.1702.08138.

Houlsby, Neil, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Chloé de Laroussilhe, Andrea Gesmundo, Mohammad Attariyan, and Sylvain Gelly. 2019. "Parameter-Efficient Transfer Learning for NLP." *International Conference on Machine Learning*, 2790–99.

Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *CoRR* abs/1704.04861.

Hsiao, Yu-Shun, Zishen Wan, Tianyu Jia, Radhika Ghosal, Abdulrahman Mahmoud, Arijit Raychowdhury, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. 2023. "MAVFI: An End-to-End Fault Analysis Framework with Anomaly Detection and Recovery for Micro Aerial Vehicles." *2023 Design, Automation &Amp; Test in Europe Conference &Amp; Exhibition (DATE)*, 1–6. https://doi.org/10.23919/date56975.2023.10137246.

Hsu, Liang-Ching, Ching-Yi Huang, Yen-Hsun Chuang, Ho-Wen Chen, Ya-Ting Chan, Heng Yi Teah, Tsan-Yao Chen, Chiung-Fen Chang, Yu-Ting Liu, and Yu-Min Tzou. 2016. "Accumulation of Heavy Metals and Trace Elements in Fluvial Sediments Received Effluents from Traditional and Semiconductor Industries." *Scientific Reports* 6 (1): 34250. https://doi.org/10.1038/srep34250.

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. "LoRA: Low-Rank Adaptation of Large Language Models." *arXiv Preprint arXiv:2106.09685*.

Huang, Cheng, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. "Erasure Coding in Windows Azure Storage." *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 15–26.

Huang, Y., Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, et al. 2019. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." *Advances in Neural Information Processing Systems* 32: 103–12.

Hudgens, Michael G., and M. Elizabeth Halloran. 2008. "Toward Causal Inference with Interference." *Journal of the American Statistical Association* 103 (482): 832–42. https://doi.org/10.1198/016214508000000292.

Hugging Face. 2026. *Safetensors Documentation*.

Hutter, Michael, Jorn-Marc Schmidt, and Thomas Plos. 2009. "Contact-Based Fault Injections and Power Analysis on RFID Tags." *2009 European Conference on Circuit Theory and Design*, 409–12. https://doi.org/10.1109/ecctd.2009.5275012.

Iandola, Forrest N., Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size." *ArXiv Preprint* abs/1602.07360.

IEEE 802.1 Working Group. 2018. *IEEE Standard for Local and Metropolitan Area Networks—Secure Device Identity*. IEEE Std 802.1AR-2018.

IEEE Standards Association. 2019. *IEEE 754-2019: Standard for Floating-Point Arithmetic*. https://doi.org/10.1109/IEEESTD.2019.8766229.

Illinois General Assembly. 2020. *Artificial Intelligence Video Interview Act (Public Act 101-0260)*. 820 ILCS 42, effective January 1, 2020.

Inan, H., K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, et al. 2023. "Llama Guard: LLM-Based Input-Output Safeguard for Human-AI Conversations." *arXiv Preprint arXiv:2312.06674*.

Incorporated, Framework Computer. 2022. *Modular Laptops: A New Approach to Sustainable Computing*.

InfiniBand Trade Association. 2000. *InfiniBand Architecture Specification Volume 1*. InfiniBand Trade Association.

Institute, World Resources, and World Business Council for Sustainable Development. 2023. *Greenhouse Gas Protocol: Corporate Standard*.

International Energy Agency. 2023. *World Energy Outlook 2023*. IEA. https://doi.org/10.1787/827374a6-en.

International Energy Agency. 2024a. *Data Centre Electricity Demand Estimates as a Share of Total Electricity Demand in Ireland and Virginia, 2023*. IEA chart.

International Energy Agency. 2024b. *Emissions Factors 2024*.

International Organization for Standardization. 2006a. *ISO 14040:2006 Environmental Management – Life Cycle Assessment – Principles and Framework*. ISO standard.

International Organization for Standardization. 2006b. *ISO 14044:2006 Environmental Management – Life Cycle Assessment – Requirements and Guidelines*. ISO standard.

International Organization for Standardization, and International Electrotechnical Commission. 2023a. *ISO/IEC 23894:2023 Information Technology – Artificial Intelligence – Guidance on Risk Management*. ISO/IEC standard.

International Organization for Standardization, and International Electrotechnical Commission. 2023b. *ISO/IEC 42001:2023 Information Technology – Artificial Intelligence – Management System*. ISO/IEC standard.

Irimia-Vladu, M. 2014. "'Green' Electronics: Biodegradable and Biocompatible Materials and Devices for Sustainable Future." *Chemical Society Reviews* 43 (2): 588–610. https://doi.org/10.1039/c3cs60235d.

Jacob, Benoit, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2704–13. https://doi.org/10.1109/cvpr.2018.00286.

Jagielski, Matthew, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. 2021. "Subpopulation Data Poisoning Attacks." *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 3104–22. https://doi.org/10.1145/3460120.3485368.

Jain, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons.

Janardhan, Santosh. 2021. *More Details about the October 4 Outage*. Meta Engineering postmortem.

Jeaugey, Sylvain. 2017. *NCCL 2.0*. GPU Technology Conference presentation.

Jeon, Myeongjae, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads." *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 947–60.

Jevons, William Stanley. 1865. *The Coal Question: An Inquiry Concerning the Progress of the Nation, and the Probable Exhaustion of Our Coal Mines*. Macmillan; Co.

Jha, A. R. 2014. *Rare Earth Materials: Properties and Applications*. CRC Press. https://doi.org/10.1201/b17045.

Jha, S., S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer. 2019. "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection." *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 112–24. https://doi.org/10.1109/dsn.2019.00025.

Jiang, A. Q., A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, et al. 2024. "Mixtral of Experts." *arXiv Preprint arXiv:2401.04088*.

Jiang, Ziheng, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, et al. 2024. "MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs." *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 745–60.

Jones, Nicola. 2018. "How to Stop Data Centres from Gobbling up the World's Electricity." *Nature* 561 (7722): 163–66. https://doi.org/10.1038/d41586-018-06610-y.

Jouppi, Norman P., Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. "In-Datacenter Performance Analysis of a Tensor Processing Unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 1–12. https://doi.org/10.1145/3079856.3080246.

Joye, Marc, and Michael Tunstall. 2012. *Fault Analysis in Cryptography*. Inf. Secur. Cryptography. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-29656-7.

Kairouz, P., and H. B. McMahan. 2021. "Advances and Open Problems in Federated Learning." *Foundations and Trends in Machine Learning* 14 (1-2): 1–210. https://doi.org/10.1561/2200000083.

Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. 2020. "Scaling Laws for Neural Language Models." *ArXiv Preprint* abs/2001.08361.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. 1997. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing - STOC '97*, 654–63. https://doi.org/10.1145/258533.258660.

Karimireddy, Sai Praneeth, Quentin Rebjock, Sebastian U. Stich, and Martin Jaggi. 2019. "Error Feedback Fixes SignSGD and Other Gradient Compression Schemes." *Proceedings of the 36th International Conference on Machine Learning (ICML)*, Proceedings of machine learning research, vol. 97: 3252–61.

Kasperkevic, Jana. 2015. "Google Says Sorry for Racist Auto-Tag in Photo App." *The Guardian*, July.

Kawazoe Aguilera, Marcos, Wei Chen, and Sam Toueg. 1997. "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication." In *Lect. Notes Comput. Sci.* Springer Berlin Heidelberg. https://doi.org/10.1007/bfb0030680.

Kim, B., M. Wattenberg, J. Gilmer, C. Cai, J. Wexler, F. Viegas, and R. Sayres. 2018. "Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)." *Proceedings of the 35th International Conference on Machine Learning (ICML)* 80: 2668–77.

Kim, J., W. J. Dally, S. Scott, and D. Abts. 2008. "Technology-Driven, Highly-Scalable Dragonfly Topology." *2008 International Symposium on Computer Architecture*, 77–88. https://doi.org/10.1109/isca.2008.19.

Kim, Jungrae, Michael Sullivan, and Mattan Erez. 2015. "Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory." *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 101–12. https://doi.org/10.1109/hpca.2015.7056025.

Kim, Sunju, Chungsik Yoon, Seunghon Ham, Jihoon Park, Ohun Kwon, Donguk Park, Sangjun Choi, Seungwon Kim, Kwonchul Ha, and Won Kim. 2018. "Chemical Use in the Semiconductor Manufacturing Industry." *International Journal of Occupational and Environmental Health* 24 (3-4): 109–18. https://doi.org/10.1080/10773525.2018.1519957.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." *ICLR* in press.

Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, et al. 2017. "Overcoming Catastrophic Forgetting in Neural Networks." *Proceedings of the National Academy of Sciences* 114 (13): 3521–26. https://doi.org/10.1073/pnas.1611835114.

Kleinberg, Jon, Sendhil Mullainathan, and Manish Raghavan. 2016. "Inherent Trade-Offs in the Fair Determination of Risk Scores." *Innovations in Theoretical Computer Science Conference*. https://doi.org/10.4230/LIPIcs.ITCS.2017.43.

Kleinrock, L. 1975. *Queueing Systems, Volume 1: Theory*. Wiley-Interscience.

Klutke, G., P. C. Kiessler, and M. A. Wortman. 2003. "A Critical Look at the Bathtub Curve." *IEEE Transactions on Reliability* 52 (1): 125–29. https://doi.org/10.1109/tr.2002.804492.

Ko, Y. 2021. "Characterizing System-Level Masking Effects Against Soft Errors." *Electronics* 10 (18): 2286. https://doi.org/10.3390/electronics10182286.

Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, et al. 2019. "Spectre Attacks: Exploiting Speculative Execution." *2019 IEEE Symposium on Security and Privacy (SP)*, 1–19. https://doi.org/10.1109/sp.2019.00002.

Kocher, Paul, Joshua Jaffe, and Benjamin Jun. 1999. "Differential Power Analysis." In *Advances in Cryptology — CRYPTO' 99*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-48405-1_25.

Kocher, Paul, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. "Introduction to Differential Power Analysis." *Journal of Cryptographic Engineering* 1 (1): 5–27. https://doi.org/10.1007/s13389-011-0006-y.

Koh, Pang Wei, and Percy Liang. 2017. "Understanding Black-Box Predictions via Influence Functions." *International Conference on Machine Learning (ICML)*, Proceedings of machine learning research, vol. 70: 1885–94.

Koh, Pang Wei, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. 2020. "Concept Bottleneck Models." *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, Proceedings of machine learning research, vol. 119: 5338–48.

Kohavi, Ron, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. "Controlled Experiments on the Web: Survey and Practical Guide." *Data Mining and Knowledge Discovery* 18 (1): 140–81. https://doi.org/10.1007/s10618-008-0114-1.

Kohavi, Ron, Diane Tang, and Ya Xu. 2020. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge University Press. https://doi.org/10.1017/9781108653985.

Kokolis, Apostolos, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zach DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2025. "Revisiting Reliability in Large-Scale Machine Learning Research Clusters." *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 1259–74. https://doi.org/10.1109/hpca61900.2025.00096.

Konečný, J., H. B. McMahan, D. Ramage, and P. Richtárik. 2016. "Federated Optimization: Distributed Machine Learning for on-Device Intelligence." *CoRR* abs/1610.02527.

Koren, Yehuda, Robert Bell, and Chris Volinsky. 2009. "Matrix Factorization Techniques for Recommender Systems." *Computer* 42 (8): 30–37. https://doi.org/10.1109/mc.2009.263.

Korosec, Kirsten. 2019. *Tesla Sues Former Employees, Zoox for Alleged Trade Secret Theft*. TechCrunch.

Krishna, Kalpesh, Gaurav Singh Tomar, Ankur P. Parikh, Nicolas Papernot, and Mohit Iyyer. 2020. "Thieves on Sesame Street! Model Extraction of BERT-Based APIs." *International Conference on Learning Representations (ICLR)*.

Krishnamoorthi, Raghuraman. 2018. "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper." *arXiv Preprint arXiv:1806.08342* abs/1806.08342.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems* 25.

Kung, H. T. 1982. "Why Systolic Architectures?" *Computer* 15 (1): 37–46. https://doi.org/10.1109/mc.1982.1653825.

Kung, Hsiang Tsung, and Charles E Leiserson. 1979. "Systolic Arrays (for VLSI)." *Sparse Matrix Proceedings 1978* 1: 256–82.

Kurakin, Alexey, Ian Goodfellow, and Samy Bengio. 2017. "Adversarial Examples in the Physical World." *International Conference on Learning Representations (ICLR), Workshop Track*, 99–112. https://doi.org/10.1201/9781351251389-8.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. "Efficient Memory Management for Large Language Model Serving with PagedAttention." *Proceedings of the 29th Symposium on Operating Systems Principles*, 611–26. https://doi.org/10.1145/3600006.3613165.

Kwon, Y. D., R. Li, S. I. Venieris, J. Chauhan, and N. D. Lane. 2024. "TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge." *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Kwon, Youngeun, and Minsoo Rhu. 2018. "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning." *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 740–53. https://doi.org/10.1145/3352460.3358284.

Lacoste, Alexandre, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. "Quantifying the Carbon Emissions of Machine Learning." *arXiv Preprint arXiv:1910.09700*.

Lai, Liangzhen, Naveen Suda, and Vikas Chandra. 2018. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs." *ArXiv Preprint* abs/1801.06601.

Lakshminarayanan, Balaji, Alexander Pritzel, and Charles Blundell. 2017. "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles." *Advances in Neural Information Processing Systems (NeurIPS)* 30.

Lamport, Leslie, Robert Shostak, and Marshall Pease. 1982. "The Byzantine Generals Problem." The Byzantine generals problem in *Concurrency: The Works of Leslie Lamport*, vol. 4. Association for Computing Machinery. https://doi.org/10.1145/3335772.3335936.

Langston, Jennifer. 2020. *Microsoft Announces New Supercomputer, Lays Out Vision for Future AI Work*. Microsoft Source.

Lannelongue, Loïc, Jason Grealey, and Michael Inouye. 2021. "Green Algorithms: Quantifying the Carbon Footprint of Computation." *Advanced Science* 8 (12): 2100707. https://doi.org/10.1002/advs.202100707.

Lazer, David, Ryan Kennedy, Gary King, and Alessandro Vespignani. 2014. "The Parable of Google Flu: Traps in Big Data Analysis." *Science* 343 (6176): 1203–5. https://doi.org/10.1126/science.1248506.

Leiserson, Charles E. 1985. "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing." *IEEE Transactions on Computers* C-34 (10): 892–901. https://doi.org/10.1109/tc.1985.6312192.

Lepikhin, Dmitry, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding." *International Conference on Learning Representations*.

Leviathan, Yaniv, Matan Kalman, and Yossi Matias. 2023. "Fast Inference from Transformers via Speculative Decoding." *Proceedings of the 40th International Conference on Machine Learning*, 19274–86.

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, et al. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *Advances in Neural Information Processing Systems* 33: 9459–74.

Li, Chengwei. 2020. *Estimating the Training Cost of GPT-3*.

Li, Guanpeng, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. https://doi.org/10.1145/3126908.3126964.

Li, Jie, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. 2023. “Analyzing Resource Utilization in an HPC System: A Case Study of NERSC’s Perlmutter.” *Lect. Notes Comput. Sci.*, Lecture notes in computer science, vol. 13948: 297–316. https://doi.org/10.1007/978-3-031-32041-5_16.

Li, M., D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. 2014. “Scaling Distributed Machine Learning with the Parameter Server.” *Proceedings of the 2014 International Conference on Big Data Science and Computing*, 583–98. https://doi.org/10.1145/2640087.2644155.

Li, Shen, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, et al. 2020. “PyTorch Distributed: Experiences on Accelerating Data Parallel Training.” *Proceedings of the VLDB Endowment* 13 (12): 3005–18. https://doi.org/10.14778/3415478.3415530.

Li, Tian, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. “Federated Learning: Challenges, Methods, and Future Directions.” *IEEE Signal Processing Magazine* 37 (3): 50–60. https://doi.org/10.1109/msp.2020.2975749.

Li, Yuliang, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, et al. 2019. “HPCC: High Precision Congestion Control.” *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 44–58. https://doi.org/10.1145/3341302.3342085.

Lifka, David A. 1995. “The ANL/IBM SP Scheduling System.” In *Job Scheduling Strategies for Parallel Processing*, edited by Dror G. Feitelson and Larry Rudolph, vol. 949. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/3-540-60153-8_35.

Lin, Ji, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. “MCUNet: Tiny Deep Learning on IoT Devices.” In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual*, edited by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. Curran Associates.

Lin, Ji, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2023. “AWQ: Activation-Aware Weight Quantization for LLM Compression and Acceleration.” *arXiv Preprint arXiv:2306.00978* abs/2306.00978.

Lin, Ji, Jiaming Tang, Haotian Tang, Shang Yang, Guangxuan Xiao, and Song Han. 2024. “AWQ: Activation-Aware Weight Quantization for on-Device LLM Compression and Acceleration.” *GetMobile: Mobile Computing and Communications* 28 (4): 12–17. https://doi.org/10.1145/3714983.3714987.

Lin, Yujun, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training.” *International Conference on Learning Representations*.

Lindholm, Andreas, Dave Zachariah, Petre Stoica, and Thomas B. Schon. 2019. “Data Consistency Approach to Model Validation.” *IEEE Access* 7: 59788–96. https://doi.org/10.1109/access.2019.2915109.

Linux man-pages project. 2026. *Perfquery(8): Query InfiniBand Port Counters on a Single Port*.

linux-rdma project. 2026. *perftest: InfiniBand Verbs Performance Tests*.

Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, et al. 2018. “Meltdown: Reading Kernel Memory from User Space.” *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 973–90.

Lipton, Zachary, Yu-Xiang Wang, and Alexander Smola. 2018. “Detecting and Correcting for Label Shift with Black Box Predictors.” *International Conference on Machine Learning*, 3122–30.

Little, John D. C. 1961. “A Proof for the Queuing Formula: <I>l = Λw.” *Operations Research* 9 (3): 383–87. https://doi.org/10.1287/opre.9.3.383.

Liu, Hao, Matei Zaharia, and Pieter Abbeel. 2023. “Ring Attention with Blockwise Transformers for Near-Infinite Context.” *arXiv Preprint arXiv:2310.01889*, ahead of print. https://doi.org/10.48550/arXiv.2310.01889.

Liu, W., Y. Xi, J. Qin, F. Sun, B. Chen, W. Zhang, R. Zhang, and R. Tang. 2022. “Neural Re-Ranking in Multi-Stage Recommender Systems: A Review.” *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, 5512–20. https://doi.org/10.24963/ijcai.2022/771.

Luccioni, Alexandra Sasha, Sylvain Viguier, and Anne-Laure Ligozat. 2023. “Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model.” *Journal of Machine Learning Research* 24 (253): 1–15.

Lundberg, Scott M., and Su-In Lee. 2017. “A Unified Approach to Interpreting Model Predictions.” *Advances in Neural Information Processing Systems* 30: 4765–74.

Lustre Wiki. 2017. *lfs setstripe: Set Striping Pattern of a File*. Lustre Wiki manual page.

Lustre Wiki. 2019. *Configuring Lustre File Striping*. Lustre Wiki.

Ma, Jeffrey, Hengzhi Pei, Leonard Lausen, and George Karypis. 2025. *Understanding Silent Data Corruption in LLM Training*. https://doi.org/10.48550/arXiv.2502.12340.

Ma, Jeffrey, Alan Tu, Yiling Chen, and Vijay Janapa Reddi. 2024. "FedStaleWeight: Buffered Asynchronous Federated Learning with Fair Aggregation via Staleness Reweighting." *arXiv Preprint arXiv:2406.02877*.

Ma, Xiaoyu, and David Patterson. 2026. *Challenges and Research Directions for Large Language Model Inference Hardware*. No. 5. Vol. 59. https://doi.org/10.1109/mc.2026.3652916.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. "Towards Deep Learning Models Resistant to Adversarial Attacks." *International Conference on Learning Representations (ICLR)*.

Mahajan, Kshiteej, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. "Themis: Fair and Efficient GPU Cluster Scheduling." *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 289–304.

Mahmoud, A., N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. 2020. "PyTorchFI: A Runtime Perturbation Tool for DNNs." *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-w)*, 25–31. https://doi.org/10.1109/dsn-w50199.2020.00014.

Malkov, Yury A., and Dmitry A. Yashunin. 2020. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (4): 824–36. https://doi.org/10.1109/TPAMI.2018.2889473.

Marulli, Fiammetta, Stefano Marrone, and Laura Verde. 2022. "Sensitivity of Machine Learning Approaches to Fake and Untrusted Data in Healthcare Domain." *Journal of Sensor and Actuator Networks* 11 (2): 21. https://doi.org/10.3390/jsan11020021.

Masanet, Eric, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. "Recalibrating Global Data Center Energy-Use Estimates." *Science* 367 (6481): 984–86. https://doi.org/10.1126/science.aba3758.

Maslej, Nestor, Loredana Fattorini, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, et al. 2023. "Artificial Intelligence Index Report 2023." *ArXiv Preprint* abs/2310.03715.

McCandlish, Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. "An Empirical Model of Large-Batch Training." *arXiv Preprint arXiv:1812.06162*, ahead of print. https://doi.org/10.48550/arXiv.1812.06162.

McCarthy, J. 1981. "Epistemological Problems of Artificial Intelligence." In *Readings in Artificial Intelligence*. Elsevier. https://doi.org/10.1016/b978-0-934613-03-3.50035-0.

McMahan, Brendan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. "Communication-Efficient Learning of Deep Networks from Decentralized Data." *International Conference on Artificial Intelligence and Statistics (AISTATS)*, Proceedings of machine learning research, vol. 54: 1273–82.

Message Passing Interface Forum. 1993. "MPI: A Message Passing Interface." *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing - Supercomputing '93*, 878–83. https://doi.org/10.1145/169627.169855.

Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard, Version 3.1*.

Meta AI Research. 2022. *OPT-175B Training Logbook*. Companion repository to the OPT-175B model release.

Meta Engineering. 2024. *Building Meta's GenAI Infrastructure*. Engineering at Meta Blog.

Micikevicius, Paulius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, et al. 2022. "FP8 Formats for Deep Learning." *arXiv Preprint arXiv:2209.05433*.

Microsoft. 2025. *Inside the World's Most Powerful AI Datacenter*. The Official Microsoft Blog.

Miller, C. 2019. "Lessons Learned from Hacking a Car." *IEEE Design &Amp; Test* 36 (6): 7–9. https://doi.org/10.1109/mdat.2018.2863106.

Miller, Charlie, and Chris Valasek. 2015. *Remote Exploitation of an Unaltered Passenger Vehicle*. Black Hat USA 2015 Whitepaper.

Mironov, I. 2017. "Rényi Differential Privacy." *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, 263–75. https://doi.org/10.1109/csf.2017.11.

Mitchell, Margaret, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. 2019. "Model Cards for Model Reporting." *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220–29. https://doi.org/10.1145/3287560.3287596.

Mitzenmacher, M. 2001. "The Power of Two Choices in Randomized Load Balancing." *IEEE Transactions on Parallel and Distributed Systems* 12 (10): 1094–104. https://doi.org/10.1109/71.963420.

ML.ENERGY Initiative, Jae-Won Chung, Jiachen Gu, Zhewei Yan, Tony Lo, Yejin Park, and Mosharaf Chowdhury. 2023. *The ML.ENERGY Leaderboard*. SymbioticLab, University of Michigan.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33. https://doi.org/10.1038/nature14236.

Mohanram, K., and N. A. Touba. 2003. "Partial Error Masking to Reduce Soft Error Failure Rate in Logic Circuits." *Proceedings. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 433–40. https://doi.org/10.1109/dftvs.2003.1250141.

Monyei, C. G., and Kirsten E. H. Jenkins. 2018. "Electrons Have No Identity: Setting Right Misrepresentations in Google and Apple's Clean Energy Purchasing." *Energy Research &Amp; Social Science* 46: 48–51. https://doi.org/10.1016/j.erss.2018.06.015.

Mu, Norman, and Justin Gilmer. 2019. *MNIST-C: A Robustness Benchmark for Computer Vision*.

Mukherjee, S. S., J. Emer, and S. K. Reinhardt. 2005. "The Soft Error Problem: An Architectural Perspective." *11th International Symposium on High-Performance Computer Architecture*, 243–47. https://doi.org/10.1109/hpca.2005.37.

Narayanan, Arvind, and Vitaly Shmatikov. 2006. "How to Break Anonymity of the Netflix Prize Dataset." In *CoRR*, abs/cs/0610105.

Narayanan, Arvind, and Vitaly Shmatikov. 2008. "Robust de-Anonymization of Large Sparse Datasets." *2008 IEEE Symposium on Security and Privacy (SP 2008)*, 111–25. https://doi.org/10.1109/sp.2008.33.

Narayanan, Deepak, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. "PipeDream: Generalized Pipeline Parallelism for DNN Training." *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 1–15. https://doi.org/10.1145/3341301.3359646.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, et al. 2021. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. https://doi.org/10.1145/3458817.3476209.

NASA Mars Program Independent Assessment Team. 2000. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*. National Aeronautics; Space Administration.

Nasr, Milad, Reza Shokri, and Amir Houmansadr. 2019. "Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-Box Inference Attacks Against Centralized and Federated Learning." *2019 IEEE Symposium on Security and Privacy (SP)*, 739–53. https://doi.org/10.1109/sp.2019.00065.

National Transportation Safety Board. 2017. *Collision Between a Car Operating with Automated Vehicle Control Systems and a Tractor-Semitrailer Truck Near Williston, Florida, May 7, 2016*. HAR-17/02. National Transportation Safety Board.

National Transportation Safety Board. 2019. *Collision Between Vehicle Controlled by Developmental Automated Driving System and Pedestrian, Tempe, Arizona, March 18, 2018*. HAR-19/03. National Transportation Safety Board.

Naumov, Maxim, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, et al. 2019. "Deep Learning Recommendation Model for Personalization and Recommendation Systems." *arXiv Preprint arXiv:1906.00091*.

Neumann, John von. 1928. "Zur Theorie Der Gesellschaftsspiele." *Mathematische Annalen* 100 (1): 295–320. https://doi.org/10.1007/bf01448847.

New York State Department of Financial Services. 2021. *Report on Apple Card Investigation*. New York State Department of Financial Services.

Ngo, Richard, Lawrence Chan, and Sören Mindermann. 2022. "The Alignment Problem from a Deep Learning Perspective." *ArXiv Preprint* abs/2209.00626.

Nguyen, John, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Michael Rabbat, Mani Malek, and Dzmitry Huba. 2021. "Federated Learning with Buffered Asynchronous Aggregation." *Proceedings of Machine Learning Research (AISTATS)* 164.

NVIDIA. 2023. *NVIDIA DGX SuperPOD: Next Generation Scalable Infrastructure for AI Leadership*. RA-11333-001 V11. NVIDIA.

NVIDIA. 2026a. *GPUDirect RDMA*.

NVIDIA. 2026b. *NCCL User Guide*.

NVIDIA. 2026c. *NVIDIA Quantum-X800 InfiniBand Platform*. NVIDIA product documentation.

NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture*. NVIDIA Whitepaper.

NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA Whitepaper, V1.0.

NVIDIA Corporation. 2025. *Product Carbon Footprint Summary for NVIDIA HGX H100*. NVIDIA Datasheet.

Nygard, Michael T. 2007. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

Obermeyer, Ziad, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. 2019. "Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations." *Science* 366 (6464): 447–53. https://doi.org/10.1126/science.aax2342.

Ofqual. 2020. *Awarding GCSE, AS, A Level, Advanced Extension Awards and Extended Project Qualifications in Summer 2020: Interim Report*. Office of Qualifications; Examinations Regulation.

Olah, C., N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. 2020. "Zoom in: An Introduction to Circuits." *Distill* 5 (3): e00024–001. https://doi.org/10.23915/distill.00024.001.

Oliynyk, Daryna, Rudolf Mayer, and Andreas Rauber. 2023. "I Know What You Trained Last Summer: A Survey on Stealing Machine Learning Models and Defences." *ACM Computing Surveys* 55 (14s): 1–41. https://doi.org/10.1145/3595292.

Olston, Christopher, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. "TensorFlow-Serving: Flexible, High-Performance ML Serving." *CoRR* abs/1712.06139. https://doi.org/10.48550/arXiv.1712.06139.

OpenAI. 2023. *March 20 ChatGPT Outage: Here's What Happened*. OpenAI incident report.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, et al. 2023. "GPT-4 Technical Report." *arXiv Preprint arXiv:2303.08774*, ahead of print. https://doi.org/10.48550/arXiv.2303.08774.

Oprea, Alina, Anoop Singhal, and Apostol Vassilev. 2022. "Poisoning Attacks Against Machine Learning: Can Machine Learning Be Trustworthy?" *Computer* 55 (11): 94–99. https://doi.org/10.1109/mc.2022.3190787.

Orekondy, Tribhuvanesh, Bernt Schiele, and Mario Fritz. 2019. "Knockoff Nets: Stealing Functionality of Black-Box Models." *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 4949–58. https://doi.org/10.1109/cvpr.2019.00509.

Organisation for Economic Co-operation and Development. 2024. *Recommendation of the Council on Artificial Intelligence*. OECD Legal Instruments.

Ousterhout, John K. 1982. "Scheduling Techniques for Concurrent Systems." *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, 22–30.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems 35* 35: 27730–44. https://doi.org/10.52202/068431-2011.

OVHcloud. 2021. *Strasbourg Datacentre: Latest Information*. OVHcloud incident update.

Pan, Sinno Jialin, and Qiang Yang. 2010. "A Survey on Transfer Learning." *IEEE Transactions on Knowledge and Data Engineering* 22 (10): 1345–59. https://doi.org/10.1109/tkde.2009.191.

Papadimitriou, George, and Dimitris Gizopoulos. 2021. "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers." *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 902–15. https://doi.org/10.1109/isca52012.2021.00075.

Papernot, Nicolas, Patrick McDaniel, and Ian Goodfellow. 2016. "Transferability in Machine Learning: From Phenomena to Black-Box Attacks Using Adversarial Samples." *arXiv Preprint arXiv:1605.07277*.

Papernot, Nicolas, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks." *2016 IEEE Symposium on Security and Privacy (SP)*, 582–97. https://doi.org/10.1109/sp.2016.41.

Papernot, N., P. McDaniel, S. Jha, M. Fredrikson, Z. Berkay Celik, and A. Swami. 2016. "The Limitations of Deep Learning in Adversarial Settings." *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 372–87. https://doi.org/10.1109/eurosp.2016.36.

Parrish, Alicia, Hannah Rose Kirk, Jessica Quaye, Charvi Rastogi, Max Bartolo, Oana Inel, Juan Ciro, et al. 2023. "Adversarial Nibbler: A Data-Centric Challenge for Improving the Safety of Text-to-Image Models." *ArXiv Preprint* abs/2305.14384.

Patarasuk, Pitch, and Xin Yuan. 2009. "Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations." *Journal of Parallel and Distributed Computing* 69 (2): 117–24. https://doi.org/10.1016/j.jpdc.2008.09.002.

Patel, Pratyush, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. "Splitwise: Efficient Generative LLM Inference Using Phase Splitting." *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 118–32. https://doi.org/10.1109/isca59077.2024.00019.

Patterson, David A., Garth Gibson, and Randy H. Katz. 1988. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 109–16. https://doi.org/10.1145/50202.50214.

Patterson, David A., and John L. Hennessy. 2017. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann.

Patterson, David, Joseph Gonzalez, Urs Holzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2022. "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink." *Computer* 55 (7): 18–28. https://doi.org/10.1109/mc.2022.3148714.

Patterson, David, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. "Carbon Emissions and Large Neural Network Training." *arXiv Preprint arXiv:2104.10350*.

Patterson, David, Joseph Gonzalez, Quoc Le, Maud Texier, and Jeff Dean. 2022. "Carbon-Aware Computing for Sustainable AI." *Communications of the ACM* 65 (11): 50–58.

Perez, Fábio, and Ian Ribeiro. 2022. "Ignore Previous Prompt: Attack Techniques for Language Models." *arXiv Preprint arXiv:2211.09527*.

Personal Information Protection Commission, Japan. 2023. *Act on the Protection of Personal Information*. Personal Information Protection Commission, Japan.

Peters, D., R. A. Calvo, and R. M. Ryan. 2018. "Designing for Motivation, Engagement and Wellbeing in Digital Experience." *Frontiers in Psychology* 9: 797. https://doi.org/10.3389/fpsyg.2018.00797.

Peterson, W. W., and D. T. Brown. 1961. "Cyclic Codes for Error Detection." *Proceedings of the IRE* 49 (1): 228–35. https://doi.org/10.1109/jrproc.1961.287814.

Phillips, P. J., C. A. Hahn, P. C. Fontana, D. A. Broniatowski, and M. A. Przybocki. 2020. "Four Principles of Explainable Artificial Intelligence." In *Gaithersburg, Maryland*, vol. 18. National Institute of Standards; Technology (NIST). https://doi.org/10.6028/nist.ir.8312-draft.

Plank, James S. 1997. "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems." *Software: Practice and Experience* 27 (9): 995–1012. https://doi.org/10.1002/(SICI)1097-024X(199709)27:9<995::AID-SPE111>3.0.CO;2-6.

Polyzotis, Neoklis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. "Data Management Challenges in Production Machine Learning." *Proceedings of the 2017 ACM International Conference on Management of Data*, 1723–26. https://doi.org/10.1145/3035918.3054782.

Pont, Michael J, and Royan HL Ong. 2002. "Using Watchdog Timers to Improve the Reliability of Single-Processor Embedded Systems: Seven New Patterns and a Case Study." *Proceedings of the First Nordic Conference on Pattern Languages of Programs*, 159–200.

Pope, Reiner, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. "Efficiently Scaling Transformer Inference." *Proceedings of Machine Learning and Systems (MLSys)* 5: 606–24.

Prakash, Shvetank, Matthew Stewart, Colby Banbury, Mark Mazumder, Pete Warden, Brian Plancher, and Vijay Janapa Reddi. 2023. "Is TinyML Sustainable? Assessing the Environmental Impacts of Machine Learning on Microcontrollers." *ArXiv Preprint* abs/2301.11899.

PyTorch Contributors. 2026a. *Torch Distributed Elastic Rendezvous*. PyTorch documentation.

PyTorch Contributors. 2026b. *Torchrun (Elastic Launch)*. PyTorch documentation.

Qiao, Aurick, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. "Pollux: Co-Adaptive Cluster Scheduling for Goodput-Optimized Deep Learning." *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 1–18.

Quaye, J., A. Parrish, O. Inel, C. Rastogi, H. R. Kirk, M. Kahng, E. Van Liemt, et al. 2024. "Adversarial Nibbler: An Open Red-Teaming Method for Identifying Diverse Harms in Text-to-Image Generation." *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 388–406. https://doi.org/10.1145/3630106.3658913.

Quiñonero-Candela, Joaquin, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence, eds. 2009. *Dataset Shift in Machine Learning*. Neural Information Processing Series. The MIT Press.

Radovanovic, Ana, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, et al. 2021. "Carbon-Aware Computing for Datacenters." *arXiv Preprint arXiv:2106.11750*, ahead of print. https://doi.org/10.48550/arXiv.2106.11750.

Rafailov, R., A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. 2023. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." *Advances in Neural Information Processing Systems (NeurIPS)* 36: 53728–41. https://doi.org/10.52202/075280-2338.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. 2020. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." *Journal of Machine Learning Research* 21 (140): 1–67.

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16. https://doi.org/10.1109/sc41405.2020.00024.

Ramesh, A., M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. 2021. "Zero-Shot Text-to-Image Generation." In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, edited by Marina Meila and Tong Zhang, vol. 139. Proceedings of Machine Learning Research. PMLR.

Rashid, Layali, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2012. "Intermittent Hardware Errors Recovery: Modeling and Evaluation." *2012 Ninth International Conference on Quantitative Evaluation of Systems*, 220–29. https://doi.org/10.1109/qest.2012.37.

Rashid, Layali, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2015. "Characterizing the Impact of Intermittent Hardware Faults on Programs." *IEEE Transactions on Reliability* 64 (1): 297–310. https://doi.org/10.1109/tr.2014.2363152.

Rasley, Jeff, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with over 100 Billion Parameters." *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery &Amp; Data Mining*, 3505–6. https://doi.org/10.1145/3394486.3406703.

Ray Project. 2026. *Ray Train: Handling Failures and Node Preemption*. Ray documentation.

Reagen, B., U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei. 2018. "Ares: A Framework for Quantifying the Resilience of Deep Neural Networks." *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 1–6. https://doi.org/10.1109/dac.2018.8465834.

Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi. 2017. "Learning Multiple Visual Domains with Residual Adapters." *Advances in Neural Information Processing Systems* 30.

Reddi, Vijay Janapa, and Meeta Sharma Gupta. 2013. *Resilient Architecture Design for Voltage Variation*. Synthesis Lectures on Computer Architecture. Springer International Publishing. https://doi.org/10.1007/978-3-031-01739-1.

Reed, I. S., and G. Solomon. 1960. "Polynomial Codes over Certain Finite Fields." *Journal of the Society for Industrial and Applied Mathematics* 8 (2): 300–304. https://doi.org/10.1137/0108018.

Regenscheid, Andrew. 2018. *Platform Firmware Resiliency Guidelines*. NIST SP 800-193. National Institute of Standards; Technology. https://doi.org/10.6028/NIST.SP.800-193.

Reis, G. A., J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. "SWIFT: Software Implemented Fault Tolerance." *International Symposium on Code Generation and Optimization*, 243–54. https://doi.org/10.1109/cgo.2005.34.

Reuters. 2023. *Human-Machine Teams Driven by AI Are about to Reshape Warfare*. Reuters news article.

Reuters. 2024. *Water-Guzzling Chipmaker TSMC and Drought-Plagued Arizona Are an Unlikely Pair, but Phoenix Says It Has Enough Water*. Fortune magazine, 8 April 2024.

Rhodes, Christopher J. 2019. "Endangered Elements, Critical Raw Materials and Conflict Minerals." *Science Progress* 102 (4): 304–50. https://doi.org/10.1177/0036850419884873.

Ribeiro, M. H., R. Ottoni, R. West, V. A. F. Almeida, and Jr. Meira Wagner. 2020. "Auditing Radicalization Pathways on YouTube." *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, 131–41. https://doi.org/10.1145/3351095.3372879.

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016. "" Why Should i Trust You?" Explaining the Predictions of Any Classifier: Explaining the Predictions of Any Classifier." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–44. https://doi.org/10.1145/2939672.2939778.

Robertson, J., and M. Riley. 2018. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies - Bloomberg*.

Rodio, Angelo, and Giovanni Neglia. 2024. "FedStale: Leveraging Stale Client Updates in Federated Learning." *Frontiers in Artificial Intelligence and Applications (ECAI 2024)* 78. https://doi.org/10.3233/faia240849.

Rolnick, David, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Greg Wayne. 2019. "Experience Replay for Continual Learning." *Advances in Neural Information Processing Systems (NeurIPS)*.

Rubin, Donald B. 1980. "Randomization Analysis of Experimental Data: The Fisher Randomization Test Comment." *Journal of the American Statistical Association* 75 (371): 591–93. https://doi.org/10.2307/2287650.

Rudin, C. 2019. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead." *Nature Machine Intelligence* 1 (5): 206–15. https://doi.org/10.1038/s42256-019-0048-x.

Russell, Jon. 2018. "Strava Says It Will Simplify Privacy Settings and Review App Features After Exposing Military Bases." *TechCrunch*, January.

Russell, S. 2021. "Human-Compatible Artificial Intelligence." In *Human-Like Machine Intelligence*. Oxford University Press. https://doi.org/10.1093/oso/9780198862536.003.0001.

Rusu, A. A., N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. 2016. "Progressive Neural Networks." *arXiv Preprint arXiv:1606.04671*.

Ryan, R. M., and E. L. Deci. 2000. "Self-Determination Theory and the Facilitation of Intrinsic Motivation, Social Development, and Well-Being." *American Psychologist* 55 (1): 68–78. https://doi.org/10.1037/0003-066x.55.1.68.

Saltzer, J. H., and M. D. Schroeder. 1975. "The Protection of Information in Computer Systems." *Proceedings of the IEEE* 63 (9): 1278–308. https://doi.org/10.1109/proc.1975.9939.

Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 4510–20. https://doi.org/10.1109/cvpr.2018.00474.

Sangchoolie, Behrooz, Karthik Pattabiraman, and Johan Karlsson. 2017. "One Bit Is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors." *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 97–108. https://doi.org/10.1109/dsn.2017.30.

Sanh, Victor, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter." *arXiv Preprint arXiv:1910.01108*, ahead of print. https://doi.org/10.48550/arXiv.1910.01108.

Schaeffer, Rylan, Brando Miranda, and Sanmi Koyejo. 2023. "Are Emergent Abilities of Large Language Models a Mirage?" *Advances in Neural Information Processing Systems 36*, 55565–81. https://doi.org/10.52202/075280-2425.

Schäfer, Mike S. 2023. "The Notorious GPT: Science Communication in the Age of Artificial Intelligence." *Journal of Science Communication* 22 (02): Y02. https://doi.org/10.22323/2.22020402.

Schmidt, Victor, Kamal Goyal, Aditya Joshi, Boris Feld, Liam Conell, Nikolas Laskaris, Doug Blank, Jonathan Wilson, Sorelle Friedler, and Sasha Luccioni. 2021. *CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing*. Zenodo software citation. https://doi.org/10.5281/zenodo.4658424.

Schmuck, Frank, and Roger Haskin. 2002. "GPFS: A Shared-Disk File System for Large Computing Clusters." *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '02)*, 231–44.

Schneider Electric. 2024. *Schneider Electric Announces New Solutions to Address the Energy and Sustainability Challenges Spurred by AI*.

Schroeder, Bianca, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. "DRAM Errors in the Wild: A Large-Scale Field Study." *ACM SIGMETRICS Performance Evaluation Review* 37 (1): 193–204. https://doi.org/10.1145/2492101.1555372.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. "Proximal Policy Optimization Algorithms." *arXiv Preprint arXiv:1707.06347*, ahead of print. https://doi.org/10.48550/arXiv.1707.06347.

Schwan, P. 2003. "Lustre: Building a File System for 1,000-Node Clusters." *Proceedings of the 2003 Linux Symposium*, 380–86.

Schwartz, Roy, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. "Green AI." *Communications of the ACM* 63 (12): 54–63. https://doi.org/10.1145/3381831.

Sculley, D., Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. "Hidden Technical Debt in Machine Learning Systems." *Advances in Neural Information Processing Systems* 28: 2503–11.

Seide, Frank, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. "1-Bit Stochastic Gradient Descent and Its Application to Data-Parallel Distributed Training of Speech DNNs." *Interspeech 2014*, 1058–62. https://doi.org/10.21437/interspeech.2014-274.

Selvaraju, R. R., M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. 2017. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization." *2017 IEEE International Conference on Computer Vision (ICCV)*, 618–26. https://doi.org/10.1109/iccv.2017.74.

SemiAnalysis. 2023. *GPT-4 Architecture, Infrastructure, Training Dataset, Costs, Vision, MoE*. SemiAnalysis Blog.

Seong, N. H., D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee. 2010. "SAFER: Stuck-at-Fault Error Recovery for Memories." *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 115–24. https://doi.org/10.1109/micro.2010.46.

Sergeev, Alexander, and Mike Del Balso. 2018. "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow." *CoRR* abs/1802.05799.

Sevilla, Jaime, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. "Compute Trends Across Three Eras of Machine Learning." *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8. https://doi.org/10.1109/ijcnn55064.2022.9891914.

Shafahi, Ali, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. 2018. "Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks." *Advances in Neural Information Processing Systems* 31.

Shafahi, Ali, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S. Davis, Gavin Taylor, and Tom Goldstein. 2019. "Adversarial Training for Free!" *arXiv Preprint arXiv:1904.12843*, ahead of print. https://doi.org/10.48550/arXiv.1904.12843.

Shalev-Shwartz, S. 2012. "Online Learning and Online Convex Optimization." *Foundations and Trends in Machine Learning* 4 (2): 107–94. https://doi.org/10.1561/2200000018.

Shalev-Shwartz, Shai, Shaked Shammah, and Amnon Shashua. 2017. "On a Formal Model of Safe and Scalable Self-Driving Cars." *ArXiv Preprint* abs/1708.06374.

Shallue, Christopher J, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2019. "Measuring the Effects of Data Parallelism on Neural Network Training." *Journal of Machine Learning Research* 20: 1–49.

Shan, S., W. Ding, J. Passananti, S. Wu, H. Zheng, and B. Y. Zhao. 2023. "Nightshade: Prompt-Specific Poisoning Attacks on Text-to-Image Generative Models." *2024 IEEE Symposium on Security and Privacy (SP)* abs/2310.13828: 807–25. https://doi.org/10.1109/sp54263.2024.00207.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." *International Conference on Learning Representations*.

Sheaffer, J. W., D. P. Luebke, and K. Skadron. 2007. "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors." *Graphics Hardware* 2007: 55–64. https://doi.org/10.2312/eggh/eggh07/055-064.

Shewhart, Walter A. 1931. *Economic Control of Quality of Manufactured Product*. D. Van Nostrand Company.

Shneiderman, B. 2020. "Bridging the Gap Between Ethics and Practice: Guidelines for Reliable, Safe, and Trustworthy Human-Centered AI Systems." *ACM Transactions on Interactive Intelligent Systems* 10 (4): 1–31. https://doi.org/10.1145/3419764.

Shneiderman, B. 2022. *Human-Centered AI*. Oxford University Press. https://doi.org/10.1093/oso/9780192845290.001.0001.

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." *arXiv Preprint arXiv:1909.08053*.

Shokri, Reza, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. "Membership Inference Attacks Against Machine Learning Models." *2017 IEEE Symposium on Security and Privacy (SP)*, 3–18. https://doi.org/10.1109/SP.2017.41.

Shumailov, Ilia, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. "AI Models Collapse When Trained on Recursively Generated Data." *Nature* 631 (8022): 755–59. https://doi.org/10.1038/s41586-024-07566-y.

Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. "The Hadoop Distributed File System." *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 1–10. https://doi.org/10.1109/msst.2010.5496972.

Sigelman, Benjamin H, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Dapper-2010-1. Google.

Sigler, Eric. 2021. *Scaling Kubernetes to 7,500 Nodes*. OpenAI Blog.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587): 484–89. https://doi.org/10.1038/nature16961.

Singh, Narendra, and Oladele A. Ogunseitan. 2022. "Disentangling the Worldwide Web of e-Waste and Climate Change Co-Benefits." *Circular Economy* 1 (2): 100011. https://doi.org/10.1016/j.cec.2022.100011.

Skorobogatov, S. 2009. "Local Heating Attacks on Flash Memory Devices." *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 1–6. https://doi.org/10.1109/hst.2009.5225028.

Skorobogatov, S. P., and R. J. Anderson. 2003. "Optical Fault Induction Attacks." In *Cryptographic Hardware and Embedded Systems - CHES 2002*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-36400-5_2.

Slade, G. 2007. *Made to Break: Technology and Obsolescence in America*. Harvard University Press. https://doi.org/10.4159/9780674043756.

Song, Liwei, Reza Shokri, and Prateek Mittal. 2019. "Privacy Risks of Securing Machine Learning Models Against Adversarial Examples." *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 241–57. https://doi.org/10.1145/3319535.3354211.

Sridharan, V., N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. 2015. "Memory Errors in Modern Systems: The Good, the Bad, and the Ugly." *ACM SIGPLAN Notices* 50 (4): 297–310. https://doi.org/10.1145/2775054.2694348.

Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (1): 1929–58.

Stahel, Walter R. 2016. "The Circular Economy." *Nature* 531 (7595): 435–38. https://doi.org/10.1038/531435a.

Statista. 2022. *Number of Internet of Things (IoT) Connected Devices Worldwide from 2019 to 2030*.

Steck, Harald, Linas Baltrunas, Ehtsham Elahi, Dawen Liang, Yves Raimond, and Justin Basilico. 2021. "Deep Learning for Recommender Systems: A Netflix Case Study." *AI Magazine* 42 (3): 7–18. https://doi.org/10.1609/aimag.v42i3.18140.

Stich, S. U., J.-B. Cordonnier, and M. Jaggi. 2018. "Sparsified SGD with Memory." *Advances in Neural Information Processing Systems* 31.

Stich, Sebastian U. 2019. "Local SGD Converges Fast and Communicates Little." *International Conference on Learning Representations*.

Stoica, Ion, Dawn Song, Raluca Ada Popa, David Patterson, Michael W. Mahoney, Randy Katz, Anthony D. Joseph, et al. 2017. "A Berkeley View of Systems Challenges for AI." *arXiv Preprint arXiv:1712.05855*.

Strubell, Emma, Ananya Ganesh, and Andrew McCallum. 2019. "Energy and Policy Considerations for Deep Learning in NLP." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 3645–50. https://doi.org/10.18653/v1/p19-1355.

Subramanya, Suhas Jayaram, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. "DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node." *Advances in Neural Information Processing Systems* 32.

Sudhakar, Soumya, Vivienne Sze, and Sertac Karaman. 2023. "Data Centers on Wheels: Emissions from Computing Onboard Autonomous Vehicles." *IEEE Micro* 43 (1): 29–39. https://doi.org/10.1109/mm.2022.3219803.

Sundararajan, Mukund, Ankur Taly, and Qiqi Yan. 2017. "Axiomatic Attribution for Deep Networks." *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 70: 3319–28.

Sutton, Richard S. 2019. "The Bitter Lesson." *Incompleteideas.net* 43.

Sweeney, Latanya. 2002. "K-ANONYMITY: A MODEL for PROTECTING PRIVACY." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10 (05): 557–70. https://doi.org/10.1142/s0218488502001648.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. "Intriguing Properties of Neural Networks." *arXiv Preprint arXiv:1312.6199*.

Tabassi, Elham. 2023. *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. National Institute of Standards; Technology. https://doi.org/10.6028/NIST.AI.100-1.

Tan, Mingxing, and Quoc V Le. 2019. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." *International Conference on Machine Learning (ICML)*, 6105–14.

Tang, Hanlin, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 2021. "1-Bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed." *Proceedings of the 38th International Conference on Machine Learning (ICML)*, Proceedings of machine learning research, vol. 139: 10118–29.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. 2005. "Optimization of Collective Communication Operations in MPICH." *The International Journal of High Performance Computing Applications* 19 (1): 49–66. https://doi.org/10.1177/1094342005051521.

The Green Grid. 2007. *Green Grid Data Center Power Efficiency Metrics: PUE and DCIE*. The Green Grid.

The Green Grid. 2011. *Water Usage Effectiveness (WUE): A Green Grid Data Center Sustainability Metric*. White Paper 35. The Green Grid.

The New York Times. 2017. *Google's Self-Driving Car Company Sues Uber over Trade Secrets*. The New York Times.

The U-Boot Project. 2026a. *Das U-Boot*.

The U-Boot Project. 2026b. *UEFI on U-Boot*.

Thompson, N. C., K. Greenewald, K. Lee, and G. F. Manso. 2021. "Deep Learning's Diminishing Returns: The Cost of Improvement Is Becoming Unsustainable." *IEEE Spectrum* 58 (10): 50–55. https://doi.org/10.1109/mspec.2021.9563954.

Thompson, Neil, Tobias Spanuth, and Hyrum Anderson Matthews. 2023. "The Computational Limits of Deep Learning and the Future of AI." *Communications of the ACM* 66 (3): 48–57.

Tillet, Philippe, H. T. Kung, and David Cox. 2019. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations." *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 10–19. https://doi.org/10.1145/3315508.3329973.

Tiwari, Devesh, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, et al. 2015. "Understanding GPU Errors on Large-Scale HPC Systems and the Implications for System Design and Operation." *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 331–42. https://doi.org/10.1109/hpca.2015.7056044.

Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, et al. 2023. "Llama 2: Open Foundation and Fine-Tuned Chat Models." *arXiv Preprint arXiv:2307.09288*.

Tramèr, Florian, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. 2019. "AdVersarial: Perceptual Ad Blocking Meets Adversarial Machine Learning." *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2005–21. https://doi.org/10.1145/3319535.3354222.

Tramèr, Florian, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2017. "Ensemble Adversarial Training: Attacks and Defenses." *International Conference on Learning Representations* abs/1705.07204.

Tramèr, Florian, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. "Stealing Machine Learning Models via Prediction APIs." *25th USENIX Security Symposium (USENIX Security 16)*, 601–18.

Tran, Brandon, Jerry Li, and Aleksander Madry. 2018. "Spectral Signatures in Backdoor Attacks." *Advances in Neural Information Processing Systems*.

Trusted Computing Group. 2018. *Hardware Requirements for a Device Identifier Composition Engine*. Family 2.0, Level 00, Revision 78. Trusted Computing Group.

Tsai, Min-Jen, Ping-Yi Lin, and Ming-En Lee. 2023. "Adversarial Attacks on Medical Image Classification." *Cancers* 15 (17): 4228. https://doi.org/10.3390/cancers15174228.

Tsai, Timothy, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. 2021. "NVBitFI: Dynamic Fault Injection for GPUs." *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 284–91. https://doi.org/10.1109/dsn48987.2021.00041.

Uddin, Mueen, and Azizah Abdul Rahman. 2012. "Energy Efficiency and Low Carbon Enabler Green IT Framework for Data Centers Considering Green Metrics." *Renewable and Sustainable Energy Reviews* 16 (6): 4078–94. https://doi.org/10.1016/j.rser.2012.03.014.

Ultra Ethernet Consortium. 2025. *Ultra Ethernet Specification V1.0*. Ultra Ethernet Consortium.

UNESCO. 2022. *Recommendation on the Ethics of Artificial Intelligence*. UNESCO.

United States Congress. 2022. *CHIPS and Science Act of 2022 (Public Law 117-167)*. Public Law 117-167, 136 Stat. 1366.

Uptime Institute. 2022. *Uptime Institute Global Data Center Survey 2022*. Uptime Institute.

Uptime Institute. 2026. *Tier Classification System*. Uptime Institute web page.

U.S. Department of Health and Human Services. 2005. "Summary of the HIPAA Security Rule."

U.S. Department of Health and Human Services. 2026. *The HIPAA Security Rule*. HHS.gov guidance.

Valiant, Leslie G. 1990. "A Bridging Model for Parallel Computation." *Communications of the ACM* 33 (8): 103–11. https://doi.org/10.1145/79173.79181.

Vassilev, Apostol, Alina Oprea, Alie Fordyce, Hyrum Anderson, Xander Davies, and Maia Hamin. 2025. *Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations*. NIST AI 100-2e2025. National Institute of Standards; Technology. https://doi.org/10.6028/NIST.AI.100-2e2025.

Velazco, Raoul, Gilles Foucard, and Paul Peronnard. 2010. "Combining Results of Accelerated Radiation Tests and Fault Injections to Predict the Error Rate of an Application Implemented in SRAM-Based FPGAs." *IEEE Transactions on Nuclear Science* 57 (6): 3500–3505. https://doi.org/10.1109/tns.2010.2087355.

Villalobos, Pablo, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. 2022. "Will We Run Out of Data? Limits of LLM Scaling Based on Human-Generated Data." *arXiv Preprint arXiv:2211.04325*.

Vinuesa, Ricardo, Hossein Azizpour, Iolanda Leite, Madeline Balaam, Virginia Dignum, Sami Domisch, Anna Felländer, Simone Daniela Langhans, Max Tegmark, and Francesco Fuso Nerini. 2020. "The Role of Artificial Intelligence in Achieving the Sustainable Development Goals." *Nature Communications* 11 (1): 233. https://doi.org/10.1038/s41467-019-14108-y.

Vogels, Thijs, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. "PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization." *Advances in Neural Information Processing Systems* 32.

Wachter, Sandra, Brent Mittelstadt, and Chris Russell. 2017. "Counterfactual Explanations Without Opening the Black Box: Automated Decisions and the GDPR." *SSRN Electronic Journal* 31: 841. https://doi.org/10.2139/ssrn.3063289.

Wan, Z., A. Anwar, Y.-S. Hsiao, T. Jia, V. J. Reddi, and A. Raychowdhury. 2021. "Analyzing and Improving Fault Tolerance of Learning-Based Navigation Systems." *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 841–46. https://doi.org/10.1109/dac18074.2021.9586116.

Wang, Jianyu, Zachary Charles, Zheng Xu, Gauri Joshi, H. Brendan McMahan, Blaise Aguera y Arcas, Maruan Al-Shedivat, et al. 2021. "A Field Guide to Federated Optimization." *arXiv Preprint arXiv:2107.06917*.

Wang, Y., Q. Yao, J. T. Kwok, and L. M. Ni. 2020. "Generalizing from a Few Examples: A Survey on Few-Shot Learning." *ACM Computing Surveys* 53 (3): 1–34. https://doi.org/10.1145/3386252.

Warden, Pete. 2018. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." *arXiv Preprint arXiv:1804.03209*, ahead of print. https://doi.org/10.48550/arXiv.1804.03209.

Warden, Pete, and Daniel Situnayake. 2020. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.

Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, et al. 2022. "Emergent Abilities of Large Language Models." *Transactions on Machine Learning Research* 4. https://doi.org/10.48550/arXiv.2206.07682.

Widmer, Gerhard, and Miroslav Kubat. 1996. "Learning in the Presence of Concept Drift and Hidden Contexts." *Machine Learning* 23 (1): 69–101. https://doi.org/10.1023/a:1018046501280.

Wiener, N. 1960. "Some Moral and Technical Consequences of Automation: As Machines Learn They May Develop Unforeseen Strategies at Rates That Baffle Their Programmers." *Science* 131 (3410): 1355–58. https://doi.org/10.1126/science.131.3410.1355.

Williams, Samuel, Andrew Waterman, and David Patterson. 2009. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." *Communications of the ACM* 52 (4): 65–76. https://doi.org/10.1145/1498765.1498785.

Wongpanich, A., T. Oguntebi, J. Baiocchi Paredes, Y. E. Wang, P. M. Phothilimthana, R. Mitra, Z. Zhou, N. Kumar, and V. J. Reddi. 2025. *Machine Learning Fleet Efficiency: Analyzing and Optimizing Large-Scale Google TPU Systems with ML Productivity Goodput*.

Work of the Future, MIT Task Force on the. 2020. *The Work of the Future: Building Better Jobs in an Age of Intelligent Machines*. Massachusetts Institute of Technology.

Wright, Peter, and Paul Greengrass. 1987. *Spycatcher: The Candid Autobiography of a Senior Intelligence Officer*. Viking Penguin.

Wu, Carole-Jean, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, et al. 2022. "Sustainable AI: Environmental Implications, Challenges and Opportunities." *Proceedings of Machine Learning and Systems* 4: 795–813.

Wulf, Wm. A., and Sally A. McKee. 1995. "Hitting the Memory Wall: Implications of the Obvious." *ACM SIGARCH Computer Architecture News* 23 (1): 20–24. https://doi.org/10.1145/216585.216588.

Wynsberghe, Aimee van. 2021. "Sustainable AI: AI for Sustainability and the Sustainability of AI." *AI and Ethics* 1: 213–18. https://doi.org/10.1007/s43681-021-00043-6.

Xiao, Guangxuan, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models." *Proceedings of the 40th International Conference on Machine Learning*, 38087–99.

Xiao, Wencong, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Prabal Patel, et al. 2018. "Gandiva: Introspective Cluster Scheduling for Deep Learning." *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 595–610.

Xu, Dianlei, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. 2020. "Edge Intelligence: Architectures, Challenges, and Applications." *arXiv Preprint arXiv:2003.12172*.

Xu, Weilin, David Evans, and Yanjun Qi. 2018. "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks." *Proceedings 2018 Network and Distributed System Security Symposium* 26. https://doi.org/10.14722/ndss.2018.23198.

Yao, Andrew C. 1982. "Protocols for Secure Computations." *23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982)*, 160–64. https://doi.org/10.1109/sfcs.1982.38.

Yeh, Y. C. 1996. "Triple-Triple Redundant 777 Primary Flight Computer." *1996 IEEE Aerospace Applications Conference. Proceedings* 1: 293–307. https://doi.org/10.1109/aero.1996.495891.

Yin, Dong, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. 2018. "Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates." *International Conference on Machine Learning*, 5650–59.

Yin, Shuang, Xiang Zhou, and Cedric Lam. 2022. *FEC Requirements for 800GbE/1.6TbE Optics: An Operator's Perspective*. IEEE 802.3df Task Force.

Yoo, A. B., M. A. Jette, and M. Grondona. 2003. "SLURM: Simple Linux Utility for Resource Management." In *Job Scheduling Strategies for Parallel Processing*. Springer Berlin Heidelberg. https://doi.org/10.1007/10968987_3.

You, Jie, Jae-Won Chung, and Mosharaf Chowdhury. 2023. "Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training." *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 119–39.

You, Yang, Igor Gitman, and Boris Ginsburg. 2017. "Large Batch Training of Convolutional Networks." *arXiv Preprint arXiv:1708.03888*.

You, Y., J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh. 2020. "Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes." *International Conference on Learning Representations*.

Young, John W. 1974. "A First Order Approximation to the Optimum Checkpoint Interval." *Communications of the ACM* 17 (9): 530–31. https://doi.org/10.1145/361147.361115.

Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. "Orca: A Distributed Serving System for Transformer-Based Generative Models." *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 521–38.

Yurdakul, Bilal, and Joshua Naranjo. 2020. "Statistical Properties of the Population Stability Index." *The Journal of Risk Model Validation* 52. https://doi.org/10.21314/jrmv.2020.227.

Zafrir, Ofir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. "Q8BERT: Quantized 8Bit BERT." *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, 36–39. https://doi.org/10.1109/emc2-nips53020.2019.00016.

Zaharia, Matei, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, et al. 2018. "Accelerating the Machine Learning Lifecycle with MLflow." *IEEE Data Engineering Bulletin* 41 (4): 39–45.

Zaharia, M., R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, et al. 2016. "Apache Spark: A Unified Engine for Big Data Processing." *Communications of the ACM* 59 (11): 56–65. https://doi.org/10.1145/2934664.

Zhang, Jeff Jun, Tianyu Gu, Kanad Basu, and Siddharth Garg. 2018. "Analyzing and Mitigating the Impact of Permanent Faults on a Systolic Array Based Neural Network Accelerator." *2018 IEEE 36th VLSI Test Symposium (VTS)*, 1–6. https://doi.org/10.1109/vts.2018.8368656.

Zhang, Jeff, Kartheek Rangineni, Zahra Ghodsi, and Siddharth Garg. 2018. "ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators." *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 1–6. https://doi.org/10.1109/dac.2018.8465918.

Zhang, Susan, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, et al. 2022. *OPT: Open Pre-Trained Transformer Language Models*.

Zhao, M., and G. E. Suh. 2018. "FPGA-Based Remote Power Side-Channel Attacks." *2018 IEEE Symposium on Security and Privacy (SP)*, 229–44. https://doi.org/10.1109/sp.2018.00049.

Zhao, Yanli, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, et al. 2023. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel." *Proceedings of the VLDB Endowment* 16 (12): 3848–60. https://doi.org/10.14778/3611540.3611569.

Zhao, Yue, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. 2018. "Federated Learning with Non-IID Data." *CoRR* abs/1806.00582.

Zhu, Ligeng, Zhijian Liu, and Song Han. 2019. "Deep Leakage from Gradients." *Advances in Neural Information Processing Systems* 32: 17–31. https://doi.org/10.1007/978-3-030-63076-8_2.

Zhu, Y., H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. 2015. "Congestion Control for Large-Scale RDMA Deployments." *ACM SIGCOMM Computer Communication Review* 45 (4): 523–36. https://doi.org/10.1145/2829988.2787484.

Ziegler, J. F., H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, et al. 1996. "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)." *IBM Journal of Research and Development* 40 (1): 3–18. https://doi.org/10.1147/rd.401.0003.

Zou, Andy, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. 2023. *Universal and Transferable Adversarial Attacks on Aligned Language Models*.

Zu, Y., A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, et al. 2024. "Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer." *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 761–74.

A

D·A·M Foundations

Purpose

When a single node fails, which axis is binding first: the data path, the algorithm, or the machine?

In production, "it is slow" and "it is wrong" are rarely informative symptoms. A serving stack can miss its latency objective because the accelerator is idle (data starvation), because the model is doing unnecessary work (algorithmic overhead), or because the accelerator is genuinely saturated (machine-bound). The C^3 taxonomy extends these diagnostics to the distributed fleet, but it relies on a firm foundation of single-node performance. Without understanding D·A·M, teams often optimize the wrong thing, buying faster accelerators to fix a slow input pipeline or rewriting kernels when the model is simply too large for the latency budget. This appendix provides a compact diagnostic framework, Data, Algorithm, and Machine, and maps single-machine symptoms and measurements to the term of the iron law that dominates. In C^3 terms, this single-machine refresher is the node-level base that fleet-scale compute, communication, and coordination build on.

Learning Objectives

- Classify single-machine bottlenecks by dominant Data, Algorithm, or Machine axis while recognizing mixed causes
- Map optimization techniques to their Data-Algorithm-Machine intersection zone to understand which axes they span
- Apply the iron law equation to quantitatively diagnose performance problems
- Distinguish between memory-bound and compute-bound workloads using Arithmetic Intensity
- Select appropriate profiling tools and optimization strategies for each Data-Algorithm-Machine axis
- Evaluate system health using Data-Algorithm-Machine scorecard metrics (I/O Overhead, Active Params, MFU)

How to Use This Appendix

This appendix is designed as a reference for single-node performance. Start with the scorecard-style metrics, form a hypothesis about which axis dominates, and then pick the tool that can confirm (or falsify) that hypothesis.

When training is slow on a single GPU, check utilization, data wait time, and MFU, then map each to its Data, Algorithm, or Machine axis. When serving misses a latency target, identify whether the regime is latency-bound (overhead), memory-bound (weight/KV movement), or compute-bound. When cost is exploding, use the D·A·M rubric to ensure that effort targets the dominant term, not a nonbottleneck.

The Data · Algorithm · Machine (D·A·M) taxonomy is the primary diagnostic framework for ML systems engineering. It formalizes the interdependence between information flow, mathematical logic, and physical execution. When performance stalls or behavior degrades, ask: *where is the flow blocked?* This taxonomy helps practitioners decompose bottlenecks across three diagnostic axes[1], while recognizing that real systems can involve mixed causes or interactions between axes.

[1] **MECE (Mutually Exclusive, Collectively Exhaustive)**: A classification principle from management consulting (popularized by McKinsey) requiring that categories do not overlap and together cover every possibility. Applied to systems engineering, it is useful as an idealized decomposition, but D·A·M bottlenecks often overlap in practice: a kernel can be simultaneously memory-bound, synchronization-heavy, and poorly matched to available hardware.

A.1 Diagnostic Summary

The taxonomy maps directly to the iron law of ML systems, introduced in section 1.5.2. Table A.1 summarizes the role, primary physical constraint, and core optimization pathway for each axis.

Table A.1: D·A·M Axis Reference: Each axis maps to a distinct physical constraint and a high-leverage optimization strategy. Start diagnosis here: identify which constraint is binding, then follow the optimization lever.

Axis	Role	Physical Constraint	High-Leverage Optimization
Data (D)	**Information** (The Fuel)	Bandwidth (BW)	I/O Pipeline Optimization
Algorithm (A)	**Logic** (The Blueprint)	Operations (O)	Model Compression
Machine (M)	**Physics** (The Engine)	Throughput (R_{peak})	Hardware Acceleration

A.2 Iron Law Mapping

With the three axes named, the next step is to connect them to time. The performance of any ML task is governed by the distribution of work across the D·A·M axes, and the iron law mapping reveals which component's variables dominate execution:

$$T = \underbrace{\frac{D_{\text{vol}}}{\text{BW}}}_{\text{Data (D)}} + \underbrace{\frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}}}_{\text{Algorithm (A)/Machine (M)}} + \underbrace{L_{\text{lat}}}_{\text{Overhead}}$$

Algorithm and Machine share the compute term and are separated by which variable the engineer controls. Reducing the total operations (O) is an **Algorithm** lever, while improving the hardware's peak throughput (R_{peak}) or utilization (η_{hw}) is a **Machine** lever.

A.2.1 D·A·M coordination: From sum to max

The additive iron law represents **sequential execution**—the worst case where Data, Algorithm, and Machine take turns. Skilled systems engineering transforms the sum into a max:

$$T_{\text{sequential}} = \frac{D_{\text{vol}}}{\text{BW}} + \frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}} + L_{\text{lat}} \xrightarrow{\text{overlap}} T_{\text{pipelined}} = \max\left(\frac{D_{\text{vol}}}{\text{BW}}, \frac{O}{R_{\text{peak}} \cdot \eta_{\text{hw}}}\right) + L_{\text{lat}}$$

The systems engineer's job is to make these components run in parallel, not in series. Table A.2 summarizes key D·A·M Coordination techniques:

Table A.2: D·A·M Overlap Techniques: Each technique allows one D·A·M axis to execute while another is in flight, converting the iron law's additive terms into overlapped terms.

Technique	D·A·M Axes Overlapped	Implementation
Prefetching	D overlaps M	DataLoader with `prefetch_factor`, `pin_memory=True`
CUDA Streams	D overlaps M	Separate streams for H2D transfer and compute
Async Gradient Sync	M (communication) overlaps A	Overlap gradient AllReduce with remaining backward computation as gradient buckets become ready
Double Buffering	D overlaps M	Fill buffer $i+1$ while computing on buffer i

A.3 Arithmetic Intensity Boundary

This overlap view still leaves one practical question: whether a workload is waiting on bytes or on FLOPs. The boundary between Data (memory bound) and Machine (compute bound) is not arbitrary; it is defined mathematically by **Arithmetic Intensity**[2] (I) of the workload.

The symptoms below are screening signals, not diagnoses. A workload whose arithmetic intensity sits left of $R_{\text{peak}}/\text{BW}$ cannot keep the accelerator fed with enough useful work per byte, but raw utilization must be read alongside memory-bandwidth utilization, data-loader wait, kernel occupancy, and trace gaps.

- **Low GPU utilization** (< 80 percent): Screen for data, CPU, or launch starvation. Confirm with data-loader wait, host CPU saturation, memory-bandwidth counters, and timeline gaps.
- **High GPU utilization** (> 95 percent): Treat this as machine bound only if compute units are busy and memory bandwidth is not saturated. If memory bandwidth is saturated, the bottleneck is still on the Data/Machine boundary.
- **If batch size is 1**: Treat this as a warning that latency or launch overhead may matter. For LLM decode, batch-1 execution is often memory-bandwidth bound unless traces show dispatch gaps.
- **If arithmetic intensity is below the ridge point** (about 295.2 FLOP/byte on H100): The workload is likely memory bound (Data/Machine boundary).

[2] **Arithmetic Intensity**: The ratio of floating-point operations to bytes transferred (FLOP/byte). It determines whether a workload is memory-bound or compute-bound by comparison against the hardware's *ridge point* ($R_{\text{peak}}/\text{BW}$).

A.3.1 Bottleneck diagnostic

Once the bottleneck is identified, table A.3 shows which optimizations help and which ones are wasted:

Table A.3: What Works vs. What Is Wasted: Optimizing the wrong term yields limited improvement and poor cost efficiency. A memory-bound model will not speed up from more peak FLOP/s alone; it needs higher arithmetic intensity, less data movement, or a faster memory subsystem.

If the workload is…	Dominant Term	Optimization That Works	Optimization That is Wasted
Memory-Bound	D_{vol}/BW	Quantization, pruning, batching, kernel fusion, higher memory bandwidth	More peak FLOP/s alone
Compute-Bound	$O/(R_{\text{peak}} \cdot \eta_{\text{hw}})$	Better kernels, Tensor Cores, faster GPU, lower precision	More memory bandwidth (already saturated)
Latency-Bound	L_{lat}	Batching requests, kernel fusion, async dispatch	Neither compute nor bandwidth (overhead dominates)

A.4 Tooling Map

After the action table narrows the likely fix, the tooling map identifies the measurements needed to verify it. Use table A.4 to select the right profiling tool when diagnosing a bottleneck along a particular D·A·M axis:

Table A.4: D·A·M Tooling Map: Profiling utilities for diagnosing bottlenecks along each D·A·M axis.

Axis	Key Metric	Primary Tool	Secondary Tool
Data	Batch Load Time	`tqdm` (iterations/sec)	`iotop`, `dstat` (Disk I/O)
Algorithm	FLOPs, Model Depth	PyTorch Profiler	DeepSpeed Flops Profiler
Machine	GPU Utilization, SM Occupancy	`nvidia-smi`	Nsight Compute, Nsight Systems

A.5 D·A·M Scorecard

The efficiency ratios in table A.5 grade a system's performance against its theoretical limit. This "Report Card" is anchored by **MFU**[3]—the ratio of achieved model FLOP/s to the hardware's theoretical peak FLOP/s.

[3] **MFU (Model FLOPs Utilization)**: Measures only *useful* model computation, excluding overhead like gradient synchronization and memory management.

Table A.5: The D·A·M Efficiency Rubric: Use these three numbers to characterize single-machine maturity.

Axis	Metric	Definition	Failing Grade	Passing Grade
Data	**I/O Overhead**	$\frac{\text{Data Wait Time}}{\text{Total Step Time}}$	$> 10\%$	$< 1\%$
Algorithm	**Compression/Sparsity Ratio**	$\frac{\text{Effective or active parameters}}{\text{Dense baseline parameters}}$	Workload-dependent	Workload-dependent
Machine	**MFU**	$\frac{\text{achieved model FLOP/s}}{\text{peak FLOP/s}}$	$< 30\%$	$> 50\%$

A.6 Scaling the Taxonomy: From Node to Fleet

These single-node grades become most useful when they are carried forward into distributed design. The D·A·M taxonomy is the diagnostic baseline for a single machine, but as we move from a single node to the **Machine Learning Fleet**, each axis undergoes a qualitative transformation. Understanding these "tie-ins" is essential for transitioning from local optimization to fleet-scale engineering.

The evolution of constraints

Table A.6 maps each axis from its node-level focus to its fleet-scale transformation:

Table A.6: D·A·M Constraints from Node to Fleet: How each axis of the D·A·M taxonomy shifts when scaling from a single node to a multi-node fleet. The node-level focus (left) gives way to a qualitatively different fleet-scale constraint (right)—the bottleneck migrates from local resource to global coordination cost.

Axis	Node-Level Focus (D·A·M)	Fleet-Scale Transformation
Data (D)	I/O Bandwidth (Disk/PCIe)	**The communication wall**: The bottleneck shifts from local storage to the **Bisection Bandwidth** of the network fabric.
Algorithm (A)	Model Depth/Ops Count	**The parallelism strategy**: The logic now includes *how* we partition the math across N devices (3D Parallelism).
Machine (M)	Peak TFLOP/s, HBM	**The power and reliability wall**: The constraint is no longer just silicon speed, but **Watts per rack** and cluster-wide **MTBF**.

Bridging to C^3

While D·A·M diagnoses the *components* of a single node, the C^3 Taxonomy (Appendix B) diagnoses the *interactions* of the fleet.

1. **Compute (C_1)** inherits the Algorithm and Machine axes, but adds the loss of scaling efficiency.
2. **Communication (C_2)** inherits the **Data** axis, but is governed by the speed of light and network topology rather than just local I/O.
3. **Coordination (C_3)** is the "at scale" tie-in that has no single-node equivalent. It represents the coordination tax—the time spent on synchronization, checkpoints, and failure recovery that only emerges at fleet scale.

This progression ensures that single-node efficiency (high MFU) is never traded off for fleet-scale inefficiency (low scaling efficiency). We optimize the node to serve the fleet.

A.7 Summary

The D·A·M taxonomy provides the diagnostic baseline for every ML systems analysis. By isolating the bottleneck to Data, Algorithm, or Machine, practitioners ensure that optimization efforts target the binding constraint. This single-node discipline is the prerequisite for the fleet-scale engineering addressed throughout this volume.

Key Takeaways: Single-machine diagnostic heuristics

- **Identify the dominant axis**: Decide whether Data, Algorithm, or Machine is the binding constraint before proposing any optimization.
- **Profile arithmetic intensity**: Arithmetic intensity quantitatively distinguishes between Data-bound and Machine-bound regimes.
- **Use the iron law**: The iron law transforms vague symptoms into specific term-based bottlenecks.
- **Grade with the scorecard**: The Data-Algorithm-Machine scorecard standardizes "good" performance before moving to fleet scale.

B

The C^3 Taxonomy

Purpose

When the fleet is slow, where do you look first: Compute, Communication, or Coordination?

In a distributed training cluster, "it is slow" is even less informative than on a single machine. A large distributed job can miss its throughput target because individual accelerators are underutilized, because gradient synchronization saturates the network fabric, or because checkpoint overhead and failure recovery consume too much wall-clock time. Without a taxonomy, teams buy more accelerators when they should be upgrading interconnects, or optimize kernels when the real problem is pipeline bubble overhead. This appendix provides a compact diagnostic framework, Compute, Communication, and Coordination, and maps fleet-scale symptoms and measurements to the term of the fleet law that dominates. C^3 projects the same diagnostic philosophy from a single machine to the distributed fleet before deeper fleet-scale optimization begins.

How to Use This Appendix

This appendix is designed as a reference. Start with the diagnostic summary table, form a hypothesis about which C^3 axis dominates, and then pick the tool that can confirm (or falsify) that hypothesis.

When training throughput is low, check MFU, communication fraction, and goodput ratio, then map each to its Compute, Communication, or Coordination axis. When scaling efficiency drops below expectations, use the fleet law decomposition to identify which term grew. When cost is exploding, use the C^3 scorecard to ensure that effort targets the dominant term, not a nonbottleneck.

The **C^3 Taxonomy** is the diagnostic framework for fleet-scale ML systems engineering. Where the Single-Machine Foundations (Appendix A) diagnose bottlenecks within a single node—data starvation, algorithmic overhead, or hardware saturation—the C^3 taxonomy diagnoses bottlenecks across the distributed fleet. Most fleet-scale performance problems can be diagnosed by identifying the dominant axis: Compute (are the accelerators doing useful math?), Communication (is the network moving data fast enough?), or Coordination (is the system spending too much time on synchronization, failure recovery, and scheduling?). Many production bottlenecks sit at intersections between these axes.

B.1 From D·A·M to C^3

The C^3 taxonomy does not replace D·A·M—it extends it. When a workload moves from one machine to a fleet, each D·A·M axis acquires new failure modes that the single-machine framework cannot capture. Table B.1 shows how the transition works.

Table B.1: D·A·M to C^3 Mapping: Each D·A·M axis maps to a C^3 counterpart, but Coordination (C_3) is genuinely new—it captures overhead that is negligible on a single machine but can consume 40 percent of wall-clock time at fleet scale.

D·A·M Axis	Single-Machine Concern	C^3 Extension	What Changes at Fleet Scale
Data (D)	I/O bandwidth, disk to GPU	**Communication (C_2)**	Data moves across network, not just memory hierarchy
Algorithm (A)	FLOPs, model depth, ops count	**Compute (C_1)**	Per-GPU utilization (MFU) still matters, but scaling efficiency erodes it

D·A·M Axis	Single-Machine Concern	C^3 Extension	What Changes at Fleet Scale
Machine (M)	Peak FLOP/s, hardware limits	**Compute (C_1)**	Fleet peak = $N\times$ single-GPU peak, but compound losses reduce effective FLOP/s
(no equivalent)	*(overhead term L_{lat})*	**Coordination (C_3)**	New axis: barriers, checkpoints, failure recovery, scheduling—negligible on one machine, dominant at 10K+ GPUs

The most important row in table B.1 is the last one. On a single machine, the overhead term (L_{lat}) in the iron law is typically small—kernel launch latency, Python dispatch, synchronization barriers. At fleet scale, Coordination becomes an axis in its own right: checkpoint writes, failure detection and recovery, pipeline bubble overhead, scheduler preemptions, and maintenance windows collectively consume a significant fraction of wall time. Coordination is the axis that this book exists to address.

B.2 Diagnostic Summary

With the mapping in place, diagnosis becomes a matter of matching symptoms to the axis that constrains the fleet. Table B.2 provides the main reference table for fleet-scale diagnosis. Each C^3 axis maps to a physical constraint, observable symptoms, measurable metrics, and engineering levers.

Table B.2: C^3 Diagnostic Summary: Each axis maps to a distinct physical constraint and a high-leverage optimization strategy. Start diagnosis here: identify which constraint binds, then follow the optimization pointer to the relevant chapter.

C^3 Axis	Physical Constraint	Symptoms	Key Metric	High-Leverage Optimization
Compute (C_1)	Arithmetic throughput ($R_{\text{peak}} \times \eta_{\text{hw}}$)	Low MFU, GPU utilization below 80%, poor per-GPU performance	MFU (Model FLOPs Utilization)	Kernel optimization, mixed precision, operator fusion (Chapter 9)
Communication (C_2)	Network bandwidth (BW_{net})	High AllReduce time, low scaling efficiency, communication > 30% of step	Scaling efficiency (η_{scaling}), communication fraction ($T_{\text{comm}}(N)/T_{\text{step}}(N)$)	Gradient compression, overlap compute/communication, topology optimization (Chapter 6)
Coordination (C_3)	Synchronization overhead and failure recovery	Low goodput ratio, frequent restarts, large pipeline bubbles, scheduler churn	Goodput ratio ($T_{\text{useful}}/T_{\text{wall}}$)	Async checkpointing, elastic training, faster failure detection (Chapter 7)

B.3 The Fleet Law

The same classification can be written as a time budget for each distributed step. The fleet law, introduced in section 1.5, decomposes every distributed training step into the distributed-step time budget:

$$T_{\text{step}}(N) = \frac{T_{\text{compute}}}{N} + T_{\text{comm}}(N) + T_{\text{sync}}(N) - T_{\text{overlap}}$$

This equation is the fleet-scale counterpart of the iron law. Where the iron law decomposes single-machine execution into data movement, compute, and overhead, the fleet law decomposes distributed execution into local arithmetic, network data transfer, and synchronization logic. The diagnostic strategy is identical: measure each term, identify which dominates, and direct engineering effort at the dominant term.

B.3.1 Component decomposition

Each fleet law term maps to specific measurable activities:

- $T_{\textbf{compute}}/N$: Forward pass, backward pass, optimizer step—all local arithmetic after distribution across N devices. Governed by MFU and per-GPU kernel efficiency. Improvements come from better kernels, mixed precision, and operator fusion.

- $T_{\textbf{comm}}(N)$: AllReduce of gradients, AllGather of parameters (in FSDP/ZeRO), activation transfers in tensor/pipeline parallelism. Governed by network bandwidth and collective algorithm choice. Improvements come from gradient compression, hierarchical collectives, and compute-communication overlap.
- $T_{\textbf{sync}}(N)$: Synchronization barriers, checkpoint writes, failure detection and recovery, pipeline bubble idle time, scheduler preemptions, and maintenance windows. Governed by cluster reliability and orchestration software. Improvements come from asynchronous checkpointing, elastic training, and faster failure detection.
- $T_{\textbf{overlap}}$: Communication or coordination time hidden behind useful arithmetic. Governed by scheduling and implementation overlap.

The fleet's efficiency follows directly:

$$f_{\text{compute}} = \frac{T_{\text{compute}}/N}{T_{\text{step}}(N)}$$

When f_{compute} drops below 0.5, the fleet spends more time on communication and coordination than on useful arithmetic. This compute-time fraction is not total fleet efficiency; useful fleet efficiency also depends on MFU, scaling efficiency, and goodput. The C^3 taxonomy identifies which noncompute term is responsible.

B.4 Intersection Landscape

Like D·A·M, the C^3 axes interact at their boundaries. Production bottlenecks often sit at an intersection where two axes compound.

B.4.1 Compute ∩ communication

This intersection governs whether the system can hide communication behind computation. The **communication-computation ratio** ($\rho = T_{\text{comm}}(N)/(T_{\text{compute}}/N)$) is the key metric (Appendix C). When $\rho < 1$, computation takes longer than communication and the network transfer can be overlapped—the system is compute-bound and healthy. When $\rho > 1$, GPUs finish their local work before the network delivers the next round of data, and the system is communication-bound.

Engineering at this intersection focuses on overlap strategies: launching AllReduce during the backward pass, using CUDA streams to pipeline local computation with network transfers, and increasing the computation per synchronization point (larger microbatches, gradient accumulation). Chapter 5 and Chapter 6 cover these techniques in depth.

B.4.2 Communication ∩ coordination

This intersection captures the synchronization cost embedded in communication. Every AllReduce is both a data transfer (Communication) and a synchronization barrier (Coordination)—all participants must reach the barrier before any can proceed. The cost of stragglers manifests here: if one GPU is 10 percent slower, every other GPU waits, converting a Communication operation into a Coordination bottleneck.

Engineering at this intersection focuses on reducing barrier sensitivity: asynchronous gradient methods that decouple communication from synchronization, hierarchical AllReduce that limits the blast radius of stragglers, and straggler detection with proactive mitigation. Chapter 7 addresses straggler management.

B.4.3 Compute ∩ coordination

This intersection captures the idle compute caused by coordination overhead. Pipeline bubbles are the canonical example: during warmup and cooldown phases of pipeline parallelism, some stages are idle while others compute. Checkpoint writes that block the training loop convert coordination overhead into wasted compute capacity. Failure recovery that requires rolling back and recomputing work transforms a coordination event into a computation penalty.

Engineering at this intersection focuses on minimizing idle time: increasing microbatches to shrink the pipeline bubble fraction, using asynchronous checkpointing to overlap writes with compute, and

reducing the blast radius of failures so that recomputation is bounded. Chapter 5 covers pipeline scheduling; Chapter 7 covers recovery strategies.

B.5 Rules of Thumb

In the middle of a production incident, fast heuristics narrow the search space before a profiler is needed. These thresholds provide that first line of defense.

B.5.1 The C^3 traffic light

Table B.3 provides threshold-based triage for each C^3 axis.

Table B.3: C^3 Traffic Light: Quick triage thresholds for fleet-scale diagnosis. Green means the axis is healthy; yellow means it deserves investigation; red means it is the likely bottleneck. These thresholds assume well-optimized large-model training on current-generation hardware.

C^3 Axis	Green (Healthy)	Yellow (Investigate)	Red (Bottleneck)
Compute	MFU $>$ 50%	MFU 30%–50%	MFU $<$ 30%
Communication	Comm fraction $<$ 20%	Comm fraction 20–40%	Comm fraction $>$ 40%
Coordination	Goodput ratio $>$ 90%	Goodput ratio 75–90%	Goodput ratio $<$ 75%

B.5.2 The bottleneck diagnostic table

Once the bottleneck axis is identified, table B.4 shows which optimizations help and which ones are wasted.

Table B.4: What Works vs. What Is Wasted at Fleet Scale: Optimizing a nondominant C^3 axis yields limited improvement and often worsens cost efficiency. A communication-bound fleet will gain little from faster GPUs until AllReduce and other communication bottlenecks are addressed.

If the fleet is…	Dominant Term	Optimization That Works	Optimization That is Wasted
Compute-bound	T_{compute}/N	Better kernels, mixed precision, operator fusion, next-gen accelerators	More network bandwidth (GPUs are not waiting on the network)
Communication-bound	$T_{\text{comm}}(N)$	Gradient compression, compute-comm overlap, hierarchical collectives, InfiniBand upgrade	Faster GPUs (they will just idle faster while waiting for the network)
Coordination-bound	$T_{\text{sync}}(N)$	Async checkpointing, elastic training, faster failure detection, fewer pipeline stages	Neither faster GPUs nor faster network (the time is lost to overhead, not to data movement or arithmetic)

B.6 C^3 Case Studies

Theoretical constraints manifest as confusing symptoms in production. These scenarios illustrate how to apply the C^3 taxonomy to fleet-scale performance problems. Each case isolates one dominant axis before showing which optimization levers follow from that diagnosis.

B.6.1 Case 1: The underutilized fleet (Compute)

Symptom

You provision 4,096 H100 GPUs for a large language model training run. The training loop runs without errors, but the PyTorch Profiler shows MFU of only 15 percent. The network profiler shows communication accounts for less than 10 percent of step time. The cluster is running but barely working.

Diagnosis

The **Compute** axis is the bottleneck. With 15 percent MFU, 85 percent of the fleet's arithmetic capacity sits idle on every step. This is not a communication or coordination problem—the network is fast enough and the system is stable. The system is not feeding work to the GPUs efficiently.

The fix

This is a per-GPU efficiency problem that happens to be multiplied across 4,096 accelerators. Target the Compute axis:

- **Mixed precision**: Ensure BF16/FP8 Tensor Cores are engaged. A common culprit is FP32 fallback in normalization layers or loss computation.
- **Operator fusion**: Use `torch.compile` or similar just-in-time (JIT) compilation to fuse element-wise operations and reduce kernel launch overhead.
- **Batch size tuning**: If per-GPU batch size is too small, the matrix multiplications have insufficient arithmetic intensity to saturate the Tensor Cores.

Raising MFU from 15 percent to 50 percent on the same hardware delivers 50 percent/15 percent = 3.3× more useful work—the equivalent of tripling the fleet without buying a single GPU.

B.6.2 Case 2: The communication wall (Communication)

Symptom

A 512-GPU data-parallel training run achieves 45 percent MFU during compute kernels on each GPU—good per-device kernel efficiency. Wall-clock MFU over the full step (including AllReduce) is only 20.2 percent because AllReduce consumes 55 percent of every training step. Scaling from 64 GPUs to 512 GPUs yields only 4× speedup instead of the expected 8×, or 50 percent scaling efficiency.

Diagnosis

The **Communication** axis dominates. Each GPU computes efficiently (MFU is healthy), but more than half the step time is spent synchronizing gradients across the network. The system is **communication-bound**: adding more GPUs will make it worse, not better, because AllReduce time grows with participant count while per-GPU computation stays constant.

The fix

Target the Communication axis without touching the per-GPU computation:

- **Compute-communication overlap**: Launch AllReduce during the backward pass rather than waiting until it completes. Modern frameworks (FSDP, DeepSpeed) support this natively.
- **Gradient compression**: Apply TopK sparsification or quantization to reduce the bytes crossing the network by 10–100×.
- **Hierarchical collectives**: Use intra-node NVLink for the first reduction stage, then inter-node InfiniBand only for cross-node aggregation, reducing cross-node traffic by 8×.

If communication were eliminated entirely, throughput would increase by 2.2× (Amdahl's Law applied to the 45 percent compute fraction). Realistically, reducing communication from 55 percent to 20 percent of step time would recover most of the lost scaling.

B.6.3 Case 3: The coordination tax (Coordination)

Symptom

A 10,000-GPU training run shows 40 percent MFU per device and communication accounts for only 15 percent of step time—both healthy. The job's **goodput ratio** (useful training steps/wall-clock time), however, is only 60 percent. The remaining 40 percent of wall time is consumed by checkpoint writes, failure recovery restarts, pipeline bubble idle time, scheduler preemptions (17 percent of wall time), and maintenance windows.

Diagnosis

The **Coordination** axis dominates. Per-GPU computation and inter-node communication are both efficient, but 40 percent of wall time is consumed by nonproductive overhead: 10 percent failure recovery (at 10,000 GPUs, GPU failures occur about every 5 hours), 5 percent pipeline bubbles, 3 percent checkpoint writes, and 5 percent maintenance windows. Neither faster GPUs nor faster networks will help—coordination, not computation or communication, consumes the time.

The fix

Target the Coordination axis:

- **Asynchronous checkpointing**: Overlap checkpoint writes with the next training step, reducing visible checkpoint overhead from 3 percent to near zero.
- **Elastic training**: When a node fails, shrink the job and continue rather than halting all 10,000 GPUs for recovery. This converts the 10 percent failure recovery cost into a smaller throughput reduction.
- **Pipeline schedule optimization**: Switch from GPipe to an interleaved 1F1B schedule to reduce bubble fraction, or increase microbatch count per pipeline flush.
- **Faster failure detection**: Reduce heartbeat timeout from 30 seconds to 5 seconds with hardware-level health monitoring, cutting the idle time between failure occurrence and recovery initiation.

B.7 Production Troubleshooting

Table B.5 provides a diagnostic matrix for common fleet-scale failure modes.

Table B.5: C^3 Troubleshooting Matrix: Root cause identification and remediation for common fleet-scale bottlenecks. Each row connects a user-visible symptom to the C^3 axis most likely responsible, reducing the search space before reaching for a profiler.

Symptom	C^3 Axis	Diagnostic Question	Measurement	Action
Low MFU despite fast network	**Compute**	Are Tensor Cores engaged? Is batch size sufficient for arithmetic intensity?	Per-GPU kernel trace (Nsight/PyTorch Profiler)	Enable mixed precision, increase per-GPU batch size
Throughput plateaus when adding GPUs	**Communication**	Does AllReduce time grow faster than computation shrinks?	NCCL trace, ρ ratio	Gradient compression, hierarchical collectives, overlap
Frequent job restarts	**Coordination**	What is the cluster MTBF? Is detection fast enough?	Failure logs, MTBF calculation	Elastic training, faster detection, smaller blast radius
High GPU-hours but slow progress	**Coordination**	What fraction of GPU-hours produce useful training steps?	Goodput ratio ($T_{\text{useful}}/T_{\text{wall}}$)	Async checkpointing, reduce pipeline stages, eliminate scheduler churn
Scaling efficiency drops with cluster size	**Comm/Coord**	Is the bottleneck network bandwidth or synchronization barriers?	Separate $T_{\text{comm}}(N)$ from $T_{\text{sync}}(N)$	If comm: compress or overlap. If coord: async methods
Stragglers slow entire job	**Comm ∩ Coord**	Is one node consistently last to reach the AllReduce barrier?	Per-node step time histogram	Straggler detection + replacement, bounded staleness, backup workers

B.8 Tooling Map

Engineers must measure abstract C^3 axes with concrete profiling tools. Table B.6 maps each axis to the utilities that confirm or falsify a hypothesis.

Table B.6: C^3 Tooling Map: Profiling utilities for diagnosing fleet-scale bottlenecks. Start with the primary tool for quick triage; use secondary tools for deep-dive analysis. Compute tools operate per-GPU; Communication tools operate at the network layer; Coordination tools operate at the cluster/job level.

C^3 Axis	Key Metric	Primary Tool	Secondary Tool
Compute	MFU, kernel utilization	PyTorch Profiler (TensorBoard plugin)	Nsight Compute (per-kernel roofline analysis)
Communication	AllReduce time, ρ ratio	NCCL debug logs (`NCCL_DEBUG=INFO`)	Nsight Systems (timeline), `ibstat`/`perfquery` (IB)

C^3 Axis	Key Metric	Primary Tool	Secondary Tool
Coordination	Goodput ratio, restart count	Cluster scheduler logs (Slurm, K8s event logs)	Custom goodput dashboards (for example, Google ML Goodput)

B.9 C^3 Scorecard

The C^3 Scorecard grades fleet efficiency against known thresholds, extending the Single-Machine Scorecard (Appendix A) to the distributed environment. Table B.7 defines the three metrics that characterize fleet health.

Table B.7: The C^3 Efficiency Rubric: Use these three numbers to characterize fleet health. The MFU denominator uses aggregate fleet peak, written as $NR_{\text{peak,device}}$ when R_{peak} is a per-device value. A fleet that passes all three thresholds has exhausted its easy optimizations; further gains require architectural changes, hardware upgrades, or larger problem sizes to improve the scaling regime.

C^3 Axis	Metric	Definition	Failing Grade	Passing Grade
Compute	**MFU**	$\frac{O_{\text{step}}}{NR_{\text{peak,device}} \times T_{\text{step}}}$	$< 30\%$	$> 50\%$
Communication	**Scaling Efficiency** (η_{scaling})	$\frac{T_1}{N \times T_N}$	$< 35\%$	$> 70\%$
Coordination	**Goodput Ratio**	$\frac{T_{\text{useful}}}{T_{\text{wall}}}$ or $\frac{\text{useful steps/sec}}{\text{ideal or allocated steps/sec}}$	$< 75\%$	$> 90\%$

B.10 Scaling Laws Through the C^3 Lens

Scaling laws are usually written in algorithmic FLOPs, but a training fleet delivers only the useful FLOPs that survive utilization, communication, and coordination losses. This section translates scaling-law targets into C^3 terms: first by naming the hidden perfect-systems assumption, then by defining effective FLOP/s as the quantity that connects model-quality forecasts to real cluster capacity.

B.10.1 Why scaling laws assume perfect C^3

Scaling laws—Kaplan, Chinchilla, and their successors—predict model quality as a function of algorithmic training compute. They usually abstract away systems efficiency: wall-clock time, accelerator peak FLOP/s, MFU, communication overhead, and scheduler losses enter later when teams provision hardware to deliver the target compute budget. In C^3 terms, scaling-law FLOPs must be converted into raw fleet capacity after MFU, communication, and goodput losses.

The gap between scaling-law predictions and observed training outcomes is, in large part, a C^3 gap. A team that budgets 10^{24} FLOPs for training will actually deliver far fewer effective FLOPs to the model, because each FLOP must survive three multiplicative losses: per-GPU utilization (MFU), inter-node scaling efficiency (η_{scaling}), and operational goodput.

B.10.2 The effective FLOP/s concept

A fleet's **Effective FLOP/s** is the usable throughput after compounding three independent C^3 losses:

$$\text{Effective} = \text{Peak} \times \underbrace{\text{MFU}}_{\text{Compute}} \times \underbrace{\eta_{\text{scaling}}}_{\text{Communication}} \times \underbrace{\text{Goodput Ratio}}_{\text{Coordination}}$$

Each factor maps to one C^3 axis. MFU captures per-GPU computation efficiency. Scaling efficiency captures communication overhead as GPUs are added. Goodput ratio captures coordination losses from checkpoints, failures, pipeline bubbles, and maintenance.

A concrete estimate makes the multiplicative loss visible at fleet scale.

Systems Perspective 19.1: The C^3 tax on a 100,000-GPU cluster

Consider a 100,000-GPU H100 cluster with 98,900 PFLOP/s of peak aggregate throughput. After the three C^3 losses:

$$\text{Effective} = 98,900 PFLOP/s \times 0.50 \times 0.35 \times 0.60 \approx 10384.5 PFLOP/s$$

The fleet delivers 10.5 percent of its peak capacity as useful training work. The **C^3 tax**—the ratio of peak to effective—is 9.5×: achieving a given effective compute budget requires 9.5× the raw hardware. Broken down by axis: Compute consumes a 50 percent factor (MFU), Communication consumes a 35 percent factor (using the 8,192-GPU scaling-efficiency reference as an illustrative proxy), and Coordination consumes a 60 percent factor (goodput ratio after pipeline bubbles, checkpoints, failures, scheduler preemptions, and maintenance).

This is not a failure of engineering—it is the physics of fleet-scale computation. The C^3 taxonomy quantifies where the losses occur so that optimization effort targets the dominant term.

B.11 Summary

The C^3 taxonomy provides a systematic framework for diagnosing fleet-scale bottlenecks. Each axis maps to a distinct physical constraint: arithmetic throughput and MFU bound Compute; network bandwidth and collective algorithm efficiency bound Communication; and synchronization overhead, failure recovery, and operational losses bound Coordination. The fleet law quantifies these constraints, enabling systematic diagnosis. Use the C^3 Traffic Light for quick triage, the Bottleneck Diagnostic Table to choose the right lever, and the C^3 Scorecard to grade fleet health.

Key Takeaways: Where to look first at fleet scale

- **Every fleet-scale bottleneck has a dominant C^3 axis**: Compute, Communication, or Coordination, with many real bottlenecks spanning intersections. Identify the dominant axis before optimizing.
- **Measure the C^3 Scorecard**: Use MFU $>$ 50 percent, Scaling Efficiency $>$ 70 percent, and Goodput Ratio $>$ 90 percent before investing in optimizations.
- **The C^3 tax is multiplicative**: Peak FLOP/s × MFU × Scaling Efficiency × Goodput Ratio = Effective FLOP/s. At 100,000 GPUs, expect only ~10.5 percent of peak.
- **Coordination is the new axis**: On a single machine, overhead is negligible. At fleet scale, checkpoints, failures, pipeline bubbles, and scheduling consume 40 percent or more of wall time.
- **Optimizing the wrong C^3 axis yields limited improvement**: Faster GPUs cannot fix a communication-bound fleet by themselves; faster networks cannot fix coordination overhead.

C

Fleet Foundations

Purpose

What reference numbers and physical laws should every fleet-scale ML engineer carry into distributed system design decisions?

Designing a system for a single machine requires knowledge of memory hierarchy latencies, roofline ridge points, and precision trade-offs. The moment a training job spans two machines, however, a new set of numbers takes over. Network latency replaces cache latency as the dominant concern. Component failure rates compound from negligible to inevitable. Communication overhead erodes the scaling efficiency that justifies buying more accelerators in the first place. This appendix collects the reference numbers and compact models for fleet-scale reasoning: the three system paradigms that underpin this book, the numbers every fleet engineer should know across Compute, Communication, and Coordination, and the scaling physics and thermal constraints that govern cluster design. In C^3 terms, these numbers define the physical scale on which compute capacity, communication bandwidth, and coordination overhead become measurable.

How to Use This Appendix

This appendix is designed as a reference. When a fleet-scale design question arises, use it to turn a vague symptom into a specific constraint and then choose the lever that can actually move.

- **"How fast can I communicate between nodes?"**: Start with the Communication Numbers in section C.2 for the bandwidth and latency hierarchy.
- **"How many GPUs do I actually need?"**: Use the Scaling Physics in section C.3 to understand why doubling GPUs does not halve training time.
- **"How often will my cluster fail?"**: Check the Coordination Numbers in section C.2 for MTBF tables and failure probability calculations.
- **"Is my cluster power-limited or compute-limited?"**: See Thermal and Power Physics in section C.4 for power density and cooling constraints.
- **"What does a typical overhead budget look like?"**: The Coordination Numbers include the four overhead categories that erode goodput.

C.1 The Three System Paradigms

Machine learning infrastructure at scale inherits design DNA from two distinct computing lineages—and then breaks the assumptions of both. Understanding where ML systems borrow from High-Performance Computing (HPC) and where they borrow from Warehouse-Scale Computing (WSC) is essential for choosing the right design trade-offs. A system architect who treats ML training as "just HPC" will build infrastructure that cannot tolerate failures. One who treats it as "just web services" will build infrastructure that cannot sustain the tight coupling that synchronous training demands.

High-performance computing

HPC systems descend from the supercomputer tradition. Their design philosophy is to maximize FLOP/s on tightly coupled simulations—weather modeling, molecular dynamics, nuclear physics.

Every node matters: they use specialized interconnects (InfiniBand), low-latency fabrics, and homogeneous hardware. Fault tolerance follows the **checkpoint/restart** model: if one node fails, the entire job stops, rolls back to the last checkpoint, and restarts from scratch. Scheduling is batch-oriented (Slurm), with jobs requesting rigid resource shapes ("512 nodes for 24 hours"). Nodes are *pets*—individually important, individually tracked.

Warehouse-scale computing

WSC systems descend from the web services tradition. Their design philosophy is to maximize queries per second across loosely coupled services—search, email, social media. Hardware is commodity Ethernet, varying generations coexist, and nodes are heterogeneous. Fault tolerance follows the **redundancy** model: if one node fails, the load balancer reroutes traffic to another replica. The user never notices. Scheduling is dynamic (Kubernetes, Borg), with elastic bin-packing. Nodes are *cattle*—interchangeable and expendable.

The ML fleet: A hybrid architecture

ML systems require the computational throughput of HPC (to train massive models with synchronous gradient updates) but must operate at the scale and unreliability of WSC (thousands of accelerators running for weeks). This creates a hybrid that borrows selectively from both traditions.

Training workloads are synchronous and bandwidth-hungry like HPC, but long-running and failure-tolerant like WSC. Inference workloads are latency-sensitive like WSC, but computationally heavy like HPC. The network is a fusion: TCP/IP for control planes, InfiniBand or NVLink for data planes. Fault tolerance uses **elastic training** strategies: training jobs can shrink, expand, pause, or resume without full restarts. Scheduling combines gang allocation (all-or-nothing for training) with dynamic preemption and replacement.

Table C.1 summarizes the design trade-offs across the three paradigms. The key insight is that ML fleets cannot simply adopt either HPC or WSC patterns wholesale; they must selectively combine elements from each based on the workload phase.

Table C.1: Three System Paradigms: ML fleets inherit design DNA from both HPC and WSC but break assumptions of each. Training resembles HPC (tight coupling), inference resembles WSC (elastic serving), and fault tolerance is a hybrid of both.

Dimension	HPC (Supercomputer)	WSC (Web Cloud)	ML Fleet (AI Cluster)
Philosophy	Maximize FLOP/s	Maximize QPS	Maximize Model Quality per Dollar/Watt
Coupling	Tight (MPI)	Loose (RPC/HTTP)	Hybrid (NCCL + RPC)
State	Stateful (RAM)	Stateless (DB-backed)	Semi-Stateful (Checkpoints + KV Cache)
Network	Latency-optimized	Bandwidth-optimized	Bisection-bandwidth critical
Bottleneck	Compute throughput (FLOP/s)	I/O (Disk/Net)	Memory bandwidth (HBM)
Fault model	Checkpoint/Restart	Redundancy/Replicas	Elastic shrink/expand
Scheduling	Batch (Slurm)	Orchestration (K8s)	Gang + preemption
Node model	Pets (tracked)	Cattle (expendable)	Pets during job, cattle between jobs

Foundations recap

The following provides a compact reference for the key foundational ideas that reappear throughout the distributed systems chapters.

- **The iron law** ($T \approx D_{\text{vol}}/\text{BW} + O/(R_{\text{peak}} \cdot \eta_{\text{hw}}) + L_{\text{lat}}$): Performance is bounded by data movement or compute. At fleet scale, the data movement term expands to include inter-node communication, not just memory bandwidth.
- **Roofline Model**: Distinguishes compute-bound from memory-bound workloads using arithmetic intensity. At fleet scale, a third ceiling appears: **network-bound** workloads whose performance is limited by inter-node bandwidth.

- **Amdahl's Law**: Caps strong-scaling speedup at $1/s$ (Amdahl 1967). At fleet scale, the "serial fraction" includes not just sequential code but also synchronization barriers, collective communication, and pipeline bubbles.
- **Training state rule**: about 14 bytes per parameter for common mixed-precision Adam checkpoint state (BF16 weights, FP32 master weights, and Adam moments), before framework metadata and implementation-specific extras. At fleet scale, this determines how model state is partitioned across nodes (ZeRO, tensor parallelism, pipeline parallelism).
- **Little's Law** ($Q_{\text{req}} = \lambda_{\text{arr}} T_{\text{lat}}$): Sizes inference infrastructure, where λ_{arr} is the request arrival rate and Q_{req} is the expected concurrent request count (Little 1961). At fleet scale, it determines how many serving replicas are needed behind a load balancer.

The transition from single-machine to fleet-scale reasoning requires extending these models with new dimensions: network topology, failure probability, and coordination overhead. The numbers in the next section provide the quantitative foundation for that extension.

C.2 Numbers Every Fleet Engineer Should Know

Just as single-machine analysis depends on a set of core numbers, fleet-scale engineering is governed by a set of predictable ratios and scaling behaviors. The single-machine numbers still apply within each node, but a new set of numbers governs the spaces *between* nodes. While absolute values evolve with hardware generations, the *ratios* between communication tiers and the *scaling behavior* of failure rates remain remarkably stable. Memorize the ratios and scaling trends; use the specific numbers as sanity checks.

> **Systems Perspective 20.1: Node-level numbers for fleet reasoning**
>
> Fleet reasoning depends on a few node-level numbers that directly affect fleet design: the training state rule of about 14 bytes per parameter for common Adam checkpoint state, with larger footprints when gradients, activations, metadata, or extra optimizer buffers are included; NVLink vs. HBM bandwidth for intra-node parallelism placement; and peak FLOP/s and HBM capacity for MFU, effective FLOP/s, batch sizing, and model sharding. Table C.2 and the communication numbers below give inter-node and current-generation values; the memory hierarchy and roofline ridge points within a node provide the necessary baseline.

Those node-level baselines make the fleet-level ratios easier to interpret.

> **Systems Perspective 20.2: Three fleet numbers that matter most**
>
> - **Bandwidth gap**: NVLink per-direction bandwidth within a node is ~9× faster than an InfiniBand NDR port between nodes. This ratio determines where parallelism boundaries belong—model parallelism within a node, data parallelism across nodes.
> - **MTBF scaling**: A cluster's mean time between failures is the single-component MTBF divided by the number of components ($1/N$ scaling). At 100,000 GPUs, expect a failure every 30 minutes.
> - **Effective utilization**: After MFU, scaling efficiency, and overhead losses compound, a cluster with 1,024 GPUs delivers roughly 19.2 percent of its peak FLOP/s as useful training work.

Quick reference—table C.2 condenses the fleet-scale numbers into one place. Use it for back-of-envelope checks; use the detailed tables in each subsection when designing or debugging.

Table C.2: Numbers Every Fleet Engineer Should Know (Quick Reference): One-page summary of the fleet-scale reference numbers in this section. See table C.3 and table C.4 for Communication, table C.5 for Compute, and table C.6 for Coordination.

Category	Number	Use
Communication	NVLink per-direction bandwidth ~9× IB NDR	Parallelism boundary (in-node vs. cross-node)
Communication	IB NDR ~50 GB/s, ~5 µs one-way	Inter-node bandwidth and latency
Compute	MFU 30–50%, η_{scaling} ~50% @ 1K, ~35% @ 8K	Effective FLOP/s and scaling sanity checks
Coordination	MTBF 8K: ~366.2 min; 100K: ~30 min	Failure expectation and checkpoint cadence
Coordination	175B checkpoint ~2450 GB (14 B/param common Adam state)	Recovery and storage sizing
Coordination	Goodput ~77% after overheads	Wall-clock utilization
Power & sustainability	AI rack ~70 kW; air limit ~30 kW	Cooling feasibility (liquid required above air limit)
Power & sustainability	PUE liquid ~1.06, typical ~1.40; H100 700 W -> 10,000 ≈ 7 MW IT × PUE	Facility load and carbon (see section C.4)

The invariants: Ratios that will not change

These relationships are governed by physics or architecture—they will still be true in 2035.

Network hierarchy ratio

The bandwidth gap between intra-node and inter-node communication is an architectural invariant. Chip-to-chip links (NVLink, ICI) connect through short, wide, dedicated paths on a shared substrate. Inter-node links (InfiniBand, Ethernet) must traverse cables, switches, and protocol stacks. This structural difference guarantees that intra-node bandwidth will always be an order of magnitude higher than inter-node bandwidth.

Currently, NVLink 4.0 provides 9× more per-direction bandwidth than an InfiniBand NDR port. Even as both technologies improve, the ratio persists because both are constrained by the same physics: signaling rates, lane counts, and connector density. This ratio is the single most important number for parallelism strategy: any operation requiring more bandwidth than the inter-node link can provide must be confined within a single node.

Failure scaling law

For N independent components, each with mean time between failures $\text{MTBF}_{\text{component}}$ under a constant-failure-rate assumption, equation C.1 gives the system MTBF:

$$\text{MTBF}_{\text{system}} = \frac{\text{MTBF}_{\text{component}}}{N} \tag{C.1}$$

This is pure arithmetic, not an approximation. Doubling the cluster size halves the time between failures. At 100,000 GPUs with a per-GPU MTTF of 50,000 hours, the cluster experiences a GPU failure every 30 minutes. No fault tolerance strategy can avoid this—the question is how quickly the system recovers.

AllReduce overhead scaling

The bandwidth-optimal Ring AllReduce algorithm transfers $2(N-1)M/N$ bytes per participant, where M is the message size and N is the number of participants (Patarasuk and Yuan 2009). As N grows large, this approaches $2M$ per GPU, so the per-GPU bandwidth term is nearly independent of the number of GPUs; aggregate cluster traffic still grows with the number of participants. This is why Ring AllReduce scales well in the bandwidth term. The latency term, however, grows as $2(N-1) \times \alpha$, making latency the bottleneck for small messages on large rings. This trade-off motivates hierarchical AllReduce strategies that use Ring AllReduce within nodes and Tree AllReduce across nodes.

Communication numbers

Communication defines the boundaries of parallelism. These tables quantify the bandwidth and latency at each tier of the network hierarchy, from the fastest intra-node links to the slowest wide-area connections. The key question for any distributed ML operation is: *which tier of the network hierarchy does this communication cross?*

Table C.3 shows the bandwidth available at each tier. Note the order-of-magnitude drops as communication crosses node boundaries.

Table C.3 lists the bandwidth tiers that determine where tensor, pipeline, and data-parallel traffic should stay local.

Table C.3: Communication Bandwidth Hierarchy: Per-direction bandwidth drops by roughly 9× crossing from intra-node (NVLink) to inter-node (InfiniBand). This ratio compares the H100 NVLink one-way rate against one InfiniBand NDR port and determines parallelism placement.

Interconnect	Bandwidth	Typical Role
NVLink 4.0 (H100)	450 GB/s	Tensor/pipeline parallelism within a node
TPU v5p ICI	1,200 GB/s	Intra-pod model parallelism (Google)
PCIe Gen5 x16	64 GB/s	CPU-GPU data transfer, NIC attachment
IB GXDR (1.6 Tbps)	200 GB/s	Next-gen inter-node (2026+)
IB XDR (800 Gbps)	100 GB/s	Inter-node standard (2025+)
IB NDR (400 Gbps)	50 GB/s	Current inter-node standard for AI clusters
IB HDR (200 Gbps)	25 GB/s	Previous-gen inter-node
RoCE v2 (100 GbE)	12.5 GB/s	Budget clusters, inference fleets

Table C.4 shows the one-way latency at each tier. For collective operations on small messages, latency—not bandwidth—is the bottleneck.

Table C.4: Communication Latency Hierarchy: Latency determines whether synchronous training is feasible. TCP/IP is roughly 10× slower than InfiniBand NDR, making it unsuitable for gradient synchronization in large clusters.

Interconnect	One-Way Latency	Implication
InfiniBand NDR	~5 µs	Low enough for synchronous AllReduce
InfiniBand HDR	~7 µs	Adequate for most training topologies
RoCE v2	~10 µs	Acceptable for data parallelism
TCP/IP (Ethernet)	~50 µs	Too slow for synchronous training
Cross-data-center	~40,000 μs (40 ms)	Physics floor; async training only

Compute numbers

Raw peak FLOP/s is a necessary but misleading metric for fleet capacity planning. Two multiplicative losses—Model FLOPs Utilization (MFU) and scaling efficiency—reduce effective throughput dramatically. Understanding these losses transforms fleet sizing from guesswork into engineering.

Model FLOPs Utilization (MFU) measures what fraction of peak FLOP/s a training workload actually achieves. Well-optimized large-model training on current hardware achieves 30–50 percent MFU. The gap comes from memory stalls, kernel launch overhead, pipeline bubbles, and suboptimal operator fusion. MFU below the low end of that range signals optimization opportunities; MFU above the high end indicates excellent hardware utilization.

Scaling efficiency (η_{scaling}) measures how much useful computation survives as accelerators are added. Table C.5 shows the empirical ranges for well-optimized distributed training.

Table C.5: Scaling Efficiency by Cluster Size: Efficiency degrades roughly as the logarithm of cluster size. These ranges assume well-optimized data parallelism with gradient compression. Poorly optimized systems can lose 2–3× more.

Cluster Size	Scaling Efficiency (η_{scaling})	Implication
32 GPUs	~90%	Near-linear scaling; communication is negligible
256 GPUs	~70%	Communication starts to erode throughput
1,024 GPUs	~50%	Significant overhead; optimization critical
8,192 GPUs	~35%	Fleet-scale regime; 65% of compute is overhead

Coordination numbers

At fleet scale, coordination—failure recovery, checkpointing, and maintenance—consumes a measurable fraction of wall-clock time. These numbers quantify the costs of keeping a large cluster running.

Failure rates by cluster size

Table C.6 shows how cluster MTBF shrinks with scale, using a per-GPU MTTF of 50,000 hours (~5.7 years). The failure probability column shows the likelihood of at least one GPU failure during a 24-hour training window.

Table C.6 translates per-GPU reliability into cluster-level failure cadence, which is the number checkpoint planning must absorb.

Table C.6: MTBF and Failure Probability by Cluster Size: GPU-only failure model with per-GPU MTTF of 50,000 hours. Real clusters also include NIC, PSU, cable, and switch failures, making these estimates conservative. The probability column uses $\Pr(\geq 1 \text{ failure}) = 1 - e^{-T/\text{MTBF}}$ for $T = 24$ hours.

Cluster Size	MTBF (GPU-only)	Minutes	Pr(**failure**) **in 24 hours**
256 GPUs	195.3 h	11,718.8 min	11.6%
2,048 GPUs	24.4 h	1,464.8 min	62.6%
8,192 GPUs	6.1 h	366.2 min	98%
100,000 GPUs	0.50 h	30 min	100%

The key takeaway: at 8,192 GPUs and above, failure is no longer an edge case. GPU-only MTBF is about 366.2 min at 8,192 GPUs, and the probability of at least one GPU failure within 24 hours is 98 percent. Fault tolerance is not optional at fleet scale; it is a prerequisite for completing any long-running training job. Chapter 7 covers the mechanisms in detail.

Checkpoint sizes

Checkpointing is the primary recovery mechanism, and its cost depends on the model size. Table C.7 shows checkpoint sizes for common mixed-precision Adam training checkpoints (14 bytes per parameter: 2B for BF16 weights and 12B for FP32 master weights + momentum + variance). Gradients are normally transient and recomputed after restore rather than serialized as durable checkpoint state.

Table C.7: Checkpoint Sizes by Model Scale: Uses 14 bytes/parameter for common mixed-precision Adam checkpoint state. Write time assumes 100 GB/s aggregate storage bandwidth. Asynchronous checkpointing (Chapter 7) can overlap writes with training, reducing the visible overhead.

Model Size	Checkpoint Size	Write Time @ 100 GB/s
7B parameters	98 GB	0.98 s
70B parameters	980 GB	9.8 s
175B parameters	2,450 GB	24.5 s

Model Size	Checkpoint Size	Write Time @ 100 GB/s
1T parameters	14 TB	2.3 min

Overhead budgets

Failure recovery and checkpointing are only part of the wall-clock budget. At fleet scale, table C.8 separates four recurring categories of overhead that consume time not spent on useful training:

Table C.8: Overhead Budgets for Fleet-Scale Training: These are fractions of wall-clock time. At 10,000+ GPUs, failure recovery dominates. The compound effect is additive: total goodput ratio $\approx 1.0 - (0.05 + 0.03 + 0.10 + 0.05) = 0.77 \approx 77\%$.

Overhead Category	Typical Budget	Lever
Pipeline bubbles	~5%	Increase microbatches per pipeline stage
Checkpointing	~3%	Async checkpointing, faster storage
Failure recovery	~10%	Faster detection, elastic rescheduling
Maintenance windows	~5%	Rolling upgrades, live migration

Power and sustainability numbers

Fleet-scale capacity planning and sustainability reporting require a few power numbers that every fleet engineer should know. At fleet scale, the critical numbers are **rack power density**, **PUE**, and the **air-cooling limit**—they determine where construction is feasible and what the facility load will be. Table C.9 collects these reference values:

Table C.9: Power and Cooling Reference Numbers: Rack power densities, air-cooling limits, and PUE values that govern fleet-scale capacity planning and sustainability reporting. Each row pairs a typical value with the engineering decision it informs.

Quantity	Typical value	Use
Traditional rack	12 kW	Baseline for non-AI data centers
AI rack (current gen)	70 kW	Liquid cooling required
AI rack (high-density)	100 kW	Direct-to-chip liquid
Air cooling limit	~30 kW per rack	Physics ceiling; above this, liquid is mandatory
PUE (liquid-cooled AI)	~1.06	Best case: facility load ≈ IT load
PUE (best air-cooled)	~1.12	Hyperscale best practice
PUE (industry average)	~1.40	Sanity check for cost/carbon
H100 TDP	700 W per GPU	IT load: 10,000 × 700 W = 7 MW

Rule of thumb: IT load (MW) = (number of GPUs × TDP per GPU)/10^6; facility load = IT load × PUE. A 10,000-GPU H100 cluster at 700 W each is 7 MW IT; at PUE 1.40 that is 9.8 MW facility draw. For carbon and cost, see section C.4 and Chapter 15.

Current hardware reference (c. 2025–2026)

These numbers (table C.10) reflect the current generation of fleet-scale hardware. Use them for back-of-envelope calculations, but expect them to improve ~2× every 2–3 years.

Table C.10: Fleet-Scale Hardware Reference (c. 2025–2026): Per-accelerator specifications for the 2025–2026 generations. The B200, MI300X, and Tensor Processing Unit (TPU) v6 (Trillium) represent the frontier of fleet-scale compute density and memory bandwidth.

Spec	NVIDIA B200	AMD MI300X	Google TPU v6
BF16/FP16 Peak	2,250 TFLOP/s	1,307 TFLOP/s	918 TFLOP/s
Memory Bandwidth	8 TB/s	5.30 TB/s	1.60 TB/s
HBM Capacity	192 GB	192 GB	32 GB
Intra-Node Link	1,800 GB/s (NVLink 5.0)	~890 GB/s (Infinity Fab)	~2,000 GB/s (estimated)
TDP	1000 W	750 W	~600 W (estimated)

For scale context, a DGX H100 SuperPOD contains 32 DGX H100 nodes (256 H100 GPUs). Meta announced two 24,576-H100 data-center-scale clusters built on the Grand Teton hardware platform. Google's TPU v5p pods scale to 8,960 chips. The largest announced clusters (as of 2025) exceed 100,000 accelerators.

C.3 Scaling Physics

The numbers in the previous section describe *what* the hardware can do. Scaling physics describes *what happens* when more of it is brought to bear. Reasoning starts from the ceiling on a single accelerator—the roofline—and then asks what survives as devices are added, governed by three models: Amdahl's Law extended to fleet overhead, the communication-computation ratio, and weak scaling behavior.

C.3.1 The single-accelerator roofline

Fleet performance rests on per-accelerator performance, and a single accelerator is bounded by one of two resources: the rate its arithmetic units sustain (peak FLOP/s) or the rate its memory delivers operands (HBM bandwidth). Which one binds a given kernel depends on its **arithmetic intensity** I, the number of useful operations performed per byte moved from memory, as equation C.2 defines:

$$I = \frac{\text{FLOPs}}{\text{bytes moved}} \tag{C.2}$$

Intensity is a property of the *workload*, not the hardware. A large matrix multiply has high intensity because each weight loaded from memory is reused across many multiply-accumulates; an element-wise operation, or the generation of a single token, has low intensity because each loaded value feeds only one or two operations before the next must be fetched.

The **roofline model** (Williams et al. 2009) turns intensity into a performance ceiling. Equation C.3 states that attainable throughput is the lesser of the compute ceiling and the bandwidth-limited slope:

$$R_{\text{attain}} = \min\left(R_{\text{peak}},\ I \times \text{BW}\right) \tag{C.3}$$

Plotted against intensity, equation C.3 traces a roof: a rising bandwidth-limited line on the left and a flat compute ceiling on the right. The two meet at the **ridge point**, the intensity at which a workload first saturates the arithmetic units, defined by equation C.4:

$$I_{\text{ridge}} = \frac{R_{\text{peak}}}{\text{BW}} \tag{C.4}$$

A workload with $I < I_{\text{ridge}}$ is memory bound: it sits on the slope, and faster arithmetic units change nothing because the operands cannot arrive quickly enough. A workload with $I > I_{\text{ridge}}$ is compute bound: it sits under the flat roof, limited by the arithmetic units themselves.

For an H100—BF16 peak 989 TFLOP/s, HBM bandwidth 3.35 TB/s—the ridge point is 295.2 FLOP/byte. A workload must perform hundreds of operations per byte fetched merely to keep the arithmetic units busy.

Autoregressive decoding falls far short of that line, which is why it is the defining serving bottleneck. Generating one token requires reading every model weight from HBM exactly once—16.1 GB for a 8B-parameter model in BF16—to perform about 16.1 GFLOP of matrix work, an intensity of 1 FLOP/byte. That sits far below the ridge point, so single-stream decode is purely bandwidth bound: throughput is the weight-read rate, about 208.6 tokens/s, no matter how much compute the accelerator advertises. Serving systems batch many requests precisely to raise this intensity, amortizing each weight read across every sequence in the batch and pushing decode back toward the compute roof.

The roofline is a single-accelerator law, but its logic is what scales. At fleet level the binding bandwidth is no longer HBM but the interconnect, and the bytes moved are the gradients and activations crossing the network. That ratio of network bytes to local arithmetic is the communication-computation ratio (section C.3.3): the roofline of the fleet, and the lens the rest of this section applies.

A training job provisioned with 4,096 GPUs achieves only 2.5× the throughput of a 1,024-GPU run. That observation implies about 31.2 percent scaling efficiency at the larger size, so scaling physics provides the diagnostic tools to determine whether the system is performing as physics allows or whether there is an engineering problem to fix.

C.3.2 Amdahl's Law at fleet scale

Amdahl's Law establishes that the maximum speedup of a parallel system is limited by its serial fraction s. At fleet scale, the "serial fraction" is not just sequential code—it includes every operation that forces all N GPUs to wait:

- **AllReduce synchronization**: All GPUs must complete their gradient computation before any can proceed.
- **Pipeline bubbles**: The warmup and cooldown phases of pipeline parallelism leave stages idle.
- **Checkpoint writes**: Even asynchronous checkpoints contend for storage bandwidth.
- **Python-level overhead**: Single-threaded operations in the training loop (data loading, metric logging).

To see the fleet-scale implications, consider a training workload where 10 percent of wall-clock time is spent in synchronization, communication, and other serial overhead. Amdahl's Law gives the following speedups:

- With 32 GPUs: 7.8× speedup (good efficiency)
- With 256 GPUs: 9.7× speedup (diminishing returns begin)
- With 1,024 GPUs: 9.9× speedup (approaching the ceiling)
- With 8,192 GPUs: 10× speedup (nearly at the Amdahl limit)
- With $N \to \infty$: capped at 10×

With just 10 percent serial overhead, no amount of hardware can deliver more than 10× speedup on this fixed workload. This is why fleet-scale training does not simply add more GPUs to the same problem—it scales the problem (weak scaling) to keep the serial fraction small relative to the total work.

> Systems Perspective 20.3: The compound overhead trap
>
> The 10 percent serial fraction above is optimistic. In practice, fleet-scale serial overhead accumulates from multiple sources: 5 percent pipeline bubbles + 3 percent checkpointing + 10 percent failure recovery + 5 percent maintenance. These are not all strictly serial in the Amdahl sense—some overlap with computation—but they illustrate how quickly small per-category overheads compound into significant throughput loss.

C.3.3 The communication-computation ratio

The fundamental question for any distributed training strategy is: *does the computation between synchronization points take long enough to hide the communication?* equation C.5 defines the **communication-computation ratio** (ρ) that answers this directly:

$$\rho = \frac{T_{\text{comm}}(N)}{T_{\text{compute}}/N} \tag{C.5}$$

When $\rho < 1$, computation dominates and communication can be overlapped. When $\rho > 1$, the system is **communication-bound**—GPUs spend more time waiting for data than computing on it.

Table C.11 shows the ratio for three representative scenarios. The contrast between them reveals why parallelism strategy must match the workload.

Table C.11: Communication-Computation Ratio (ρ): When $\rho \ll 1$, communication can be fully overlapped with computation. When $\rho \gg 1$, the system is communication-bound and GPUs sit idle waiting for data. Tensor parallelism within a node benefits from NVLink's high bandwidth, keeping ρ small.

Scenario	$T_{\text{comm}}(N)$	T_{compute}/N	ρ
Data parallel, 7B model, 256	560.4 ms (AllReduce over IB NDR)	~5 s (fwd+bwd)	0.11
Data parallel, 350M model, 256	29.6 ms (AllReduce over IB NDR)	~10 ms (fwd+bwd)	3
Tensor parallel, 8 (NVLink)	~0.04 ms (activation over NVLink)	~1 ms (one layer)	0.036

The 7B model on 256 achieves $\rho = 0.11$—communication takes about 11.2 percent as long as computation, which can be partially overlapped. The 350M model on the same cluster has $\rho = 3$—communication dominates, making this configuration communication-bound. The solution is either to use fewer GPUs (reduce N in the AllReduce) or to increase the computation per step (larger batch size, gradient accumulation).

Tensor parallelism within a node achieves $\rho = 0.036$, confirming that NVLink bandwidth is sufficient to keep intra-node parallelism compute-bound. This is the quantitative reason why tensor parallelism is confined within nodes while data parallelism spans across them.

C.3.4 Weak scaling at fleet scale

Amdahl's Law paints a pessimistic picture because it assumes a fixed problem size. In practice, fleet-scale training often follows **weak scaling**: the problem size (tokens, data, model parameters) grows proportionally with the number of GPUs. Gustafson's Law captures this more optimistic view (Gustafson 1988).

The key insight for fleet-scale ML is that weak scaling is not just a mathematical convenience—it reflects reality. Engineers do not use 8,192 GPUs to train a 7B model faster; they use them to train a 70B or 700B model in reasonable time. As models and training batches grow, engineers often increase the compute performed between synchronization points. For dense Transformers, compute is commonly estimated as about $6\times$ parameters $\times$ tokens, while gradient communication scales with parameter bytes, so the communication-computation ratio ρ depends on batch size, sequence length, accumulation, and parallelism layout rather than on parameter count alone.



Systems Perspective 20.4: The compound loss of fleet utilization

A 1,024-GPU H100 cluster has a peak aggregate throughput of 1,012,736 TFLOP/s. After the three multiplicative losses, the effective throughput is:

$$R_{\text{eff}} = N R_{\text{peak}} \times \text{MFU} \times \eta_{\text{scaling}} \times \eta_{\text{goodput}}$$

$$= 1,012,736 TFLOP/s \times 0.50 \times 0.50 \times 0.77 \approx 194,951.7 \text{ TFLOP/s}$$

The cluster delivers 19.2 percent of its peak FLOP/s as useful training work. The remaining 80.8 percent is explained by retained factors of 50 percent MFU, 50 percent scaling efficiency, and 77 percent goodput.

This is not a failure of engineering—it is the physics of fleet-scale computation. Every additional GPU adds less marginal useful work, but the total throughput still far exceeds what a smaller cluster could achieve. The goal is not to reach 100 percent utilization; the goal is to deliver trained models faster than any smaller configuration could.

C.4 Thermal and Power Physics

Compute performance ultimately converts to heat, and heat must be removed. At fleet scale, thermal and power constraints are not secondary concerns—they determine where construction is feasible, how densely accelerators can be packed, and what the operating costs will be. A cluster that is architecturally sound but thermally infeasible cannot be built.

A cluster design calling for 10,000 H100 GPUs at 700 W each—7 MW of IT load—and a PUE of 1.40 puts the facility load at 9.8 MW total. This is the electrical load of a small town. Before optimizing software, the physics of power delivery and heat removal must be feasible.

C.4.1 Power density wall

The shift from traditional data center workloads to AI training has created a **power density crisis**. Table C.12 quantifies the gap.

Table C.12: Power Density: Traditional vs. AI Workloads: AI racks consume roughly 5.8× the power of traditional racks. Air cooling physically cannot remove heat fast enough above ~30 kW per rack, making liquid cooling mandatory for modern AI clusters.

Configuration	Power per Rack	Cooling Implication
Traditional data center	12 kW	Standard air cooling sufficient
AI cluster (current gen)	70 kW	Liquid cooling required
AI cluster (high-density)	100 kW	Direct-to-chip liquid cooling required
Air cooling limit	~30 kW	Physics ceiling for forced-air convection

The 5.8× increase in rack power density between traditional and AI workloads is not merely an engineering challenge—it represents a fundamental constraint on facility design. Existing data centers built for 12 kW racks cannot simply be "retrofitted" with AI hardware. The electrical distribution, cooling infrastructure, and floor loading must all be redesigned. This is why new AI-focused data centers are being built from the ground up with liquid cooling as the baseline assumption.

C.4.2 The energy hierarchy at scale

Power Usage Effectiveness (PUE) measures facility energy overhead: total facility energy divided by IT equipment energy (The Green Grid 2007; Uptime Institute 2022). A PUE of 1.0 would mean all facility energy is delivered to IT equipment; values above 1.0 represent overhead for cooling, power conversion, lighting, and other facility systems.

Table C.13 compares PUE across data center generations and cooling technologies.

Table C.13: Power Usage Effectiveness (PUE): Lower is better. Liquid cooling achieves PUE near 1.0 because it removes heat directly from the chip without the intermediate step of heating air. The gap between legacy (1.58) and liquid-cooled (1.06) represents about a 32.9 percent reduction in total facility power at fixed IT load; the non-IT overhead drops from 0.58 MW to 0.06 MW per MW of IT load.

Data Center Type	PUE	Overhead per MW of IT Load
Liquid-cooled AI data center	1.06	60 kW cooling + infrastructure
Best-in-class air-cooled	1.12	120 kW overhead
Industry average	1.40	400 kW overhead
Legacy enterprise	1.58	580 kW overhead

For fleet-scale cost calculations, PUE directly multiplies the electricity bill. A 10 MW IT load in a legacy data center (PUE 1.58) requires 15.8 MW total, while the same load in a liquid-cooled facility (PUE 1.06) requires only 10.6 MW—a savings of 5.2 MW. At typical commercial electricity rates, this difference translates to millions of dollars per year. Chapter 15 covers the full sustainability implications.

Summary

Key Takeaways: Fleet scale changes every constraint

- **Three paradigms, one hybrid**: ML fleets combine HPC's tight coupling (for training) with WSC's elastic fault tolerance (for inference), creating a new architectural paradigm that cannot adopt either predecessor's patterns wholesale.
- **Bandwidth gap**: NVLink provides ~9× more per-direction bandwidth than InfiniBand NDR. This invariant ratio determines parallelism placement: tensor parallelism within nodes, data parallelism across nodes.
- **Failures become routine at scale**: Cluster MTBF scales as $1/N$. At 8,192 GPUs, expect a GPU failure about every 366.2 minutes, with 98.04 percent probability over 24 hours. Fault tolerance is not optional—it is a prerequisite for completing any fleet-scale training job.
- **Compound utilization loss**: After MFU (~50 percent), scaling efficiency (~50 percent), and goodput after operational overhead (~77 percent) compound, a 1,024-GPU cluster delivers ~19.2 percent of peak FLOP/s as useful work.
- **Power density demands liquid cooling**: AI racks consume 5.8× the power of traditional racks. Air cooling fails above ~30 kW per rack, making liquid cooling a physical requirement for modern AI clusters.
- **Communication-computation ratio (ρ) governs scaling strategy**: When $\rho > 1$, GPUs are idle waiting for data—reduce parallelism or increase computation per step. When $\rho \ll 1$, communication can be fully overlapped.
- **Weak scaling is the fleet-scale paradigm**: Engineers do not use more GPUs to solve the same problem faster (strong scaling); they use more GPUs to solve larger problems in reasonable time. This keeps the serial fraction small and utilization high.

D

Communication

Purpose

What communication primitives bind the fleet together, and how do we predict their cost before running a single experiment?

Distributed ML training spends a surprising fraction of wall time not computing, but communicating. A large distributed training run may dedicate a substantial fraction of every iteration to synchronizing gradients, redistributing activations, or shuffling expert tokens. When that communication overhead is the bottleneck, no amount of faster arithmetic will help. The question shifts from "how many FLOPs?" to "how many bytes, across what topology, at what latency?" This appendix collects the communication cost models that answer those questions quantitatively: startup-latency and bandwidth models, the cost of every collective operation used in distributed training, pipeline bubble overhead, and the economics of gradient compression. These models support the distributed strategies, collective algorithms, fault tolerance mechanisms, and performance analysis used throughout the volume. In C^3 terms, the appendix develops the communication axis, the cost of moving bytes across a fleet.

How to Use This Appendix

This appendix is designed as a reference. Reach for it when translating a distributed training symptom ("AllReduce is slow," "pipeline bubbles are killing throughput," "should we compress gradients?") into a quantitative diagnosis.

Conventions used here follow the book-wide notation. We use α for per-message startup latency, β for link bandwidth in bytes per second, n for point-to-point message size, n^* for the α-β crossover point, N for the number of participating GPUs, and M for collective payload size in bytes.

- **When AllReduce is slow**: Use section D.2 to compute the expected time and compare ring vs. tree costs.
- **When choosing ring vs. tree**: Use section D.3 and the crossover analysis to match algorithm to message size.
- **When pipeline bubbles dominate**: Use section D.4 to compute bubble fraction and determine the required microbatch count.
- **When gradient compression is proposed**: Use section D.5 to run the break-even calculation before committing engineering effort.

D.1 The α-β Communication Model

> Systems Perspective 21.1: Why this matters
>
> Estimating *how* long a gradient synchronization will take across 256 GPUs must happen before committing to a cluster topology. The α-β model is the "Roofline for networks": it gives a first-order prediction of transfer time from just two hardware parameters and the message size.

Every point-to-point transfer on a network follows the same fundamental pattern: the sender pays a fixed startup cost to initiate the message, then transmits the payload at a rate deter-

mined by the link's bandwidth. This two-parameter model captures the essential physics of network communication.

The α-β **model**, given by equation D.1, predicts the time to transfer a point-to-point message of n bytes:

$$T(n) = \alpha + \frac{n}{\beta} \tag{D.1}$$

where:

- α is the **startup latency** (seconds)—the fixed cost of initiating a message, including software overhead, NIC processing, and routing setup
- β is the **link bandwidth** (bytes/s)—the sustained data rate after startup
- n is the **message size** (bytes)

The model decomposes every transfer into exactly two costs: a per-message tax (α) paid regardless of size, and a per-byte tax (n/β) that scales linearly with the payload. This decomposition drives all the algorithm selection decisions in section D.3.

Table D.1 lists the α and β values for interconnects commonly found in ML training clusters. These numbers are the starting point for every communication cost estimate in this appendix.

Table D.1: α-β Parameters by Interconnect: Startup latency and sustained bandwidth for interconnects used in ML clusters. The β column uses one-way or per-direction bandwidth, so the H100 NVLink entry is half of the 900 GB/s bidirectional aggregate specification. NVLink and PCIe operate within a node; InfiniBand and RoCE operate between nodes. TCP latency is dominated by kernel software stack overhead.

Interconnect	α (Latency)	β (Bandwidth)	Typical Role
NVLink 4.0 (H100)	500 ns	450 GB/s	Intra-node GPU-to-GPU
PCIe Gen5	1000 ns	64 GB/s	CPU-GPU, NIC-GPU
InfiniBand NDR (400 Gbps)	5 µs	50 GB/s	Inter-node (high-end clusters)
InfiniBand HDR (200 Gbps)	7 µs	25 GB/s	Inter-node (previous generation)
RoCE v2 (100 GbE)	10 µs	12.5 GB/s	Inter-node (Ethernet clusters)
TCP/IP (Ethernet)	50 µs	Varies	Control plane, non-RDMA fallback

D.1.1 Latency-dominated vs. bandwidth-dominated regimes

The α-β model reveals two distinct operating regimes. For small messages, the startup latency α dominates—the link spends most of its time *starting* transfers, not *sustaining* them. For large messages, the bandwidth term n/β dominates, and the startup cost becomes negligible.

The **crossover message size** n^*, defined by equation D.2, is the point where both terms contribute equally:

$$n^* = \alpha \times \beta \tag{D.2}$$

Below n^*, communication is **latency-dominated**: sending more small messages wastes time on repeated startups. Above n^*, communication is **bandwidth-dominated**: the link is fully utilized, and the only way to go faster is higher bandwidth.

> Napkin Math 21.1: Worked example: Crossover on InfiniBand NDR
>
> **Setup**: InfiniBand NDR has α = 5 µs and β = 50 GB/s.
> **Question**: At what message size does the bandwidth term equal the latency term?
> **Math**:
> $n^* = \alpha \times \beta = 5\ \mu s \times 50\ GB/s = 250\ KB$

Analysis: Messages smaller than 250 KB are latency-dominated on IB NDR. Gradient tensors from a single layer of a large model are typically 10–100 MB—well above this threshold. Control messages, heartbeats, and barrier synchronizations, however, are well below it.

Systems insight: This is why NCCL and Gloo batch small gradient tensors into larger "buckets" before launching AllReduce. Sending each tensor individually would pay the 5 µs startup cost hundreds of times per iteration instead of a handful.

D.1.2 The LogGP model extension

The α-β model assumes a message is a single, atomic transfer. In practice, networks pipeline messages: a sender can inject a new packet before the previous one has been fully received. The **LogGP model** (Alexandrov et al. 1995) extends LogP with a long-message gap parameter:

- g **(gap)**: The minimum interval between consecutive message injections at the sender. This models the NIC's injection rate limit.
- o_{send}, o_{recv} **(overhead)**: The CPU time consumed by the processor to initiate or complete a message, during which it cannot do useful compute.

For a long message of n bytes, LogGP models transfer time as approximately:

$$T_{\text{long}}(n) \approx o_{\text{send}} + L_{\text{LogGP}} + (n-1)G + o_{\text{recv}}$$

where L_{LogGP} is the network latency, o_{send} and o_{recv} are sender and receiver overheads, G is the gap per byte for long messages, and g controls the minimum interval between consecutive message injections. Informal alpha-beta-style approximations can be useful for ML profiling, but the standard LogGP model keeps these gap parameters distinct from a simple bandwidth term.

In most ML training scenarios, the α-β model is sufficient because gradient tensors are large enough that pipelining effects are secondary to raw bandwidth. The LogGP model becomes important when analyzing the overhead of many small control messages or when modeling the interaction between communication and computation overlap—situations that arise in fine-grained pipeline parallelism and asynchronous gradient updates.

The α-β model describes point-to-point transfers. Collective operations—where all GPUs participate—compose multiple point-to-point transfers, and their cost formulas derive directly from the model above.

D.2 Collective Operation Complexity

Systems Perspective 21.2: Why this matters

Every distributed training step ends with a collective synchronization. The cost of that collective—not the cost of the matrix multiplies—often determines whether scaling from 8 GPUs to 256 is feasible. Understanding the exact complexity of each collective reveals which parallelism strategy (data, tensor, pipeline, expert) will dominate wall time at a given scale.

Collective operations move data between all N GPUs simultaneously. Each collective has a characteristic bandwidth term (how much data flows) and latency term (how many sequential message steps are required). The formulas below use the **bandwidth-optimal ring algorithm** unless otherwise noted; section D.3 discusses when other algorithms are preferable.

D.2.1 AllReduce

AllReduce is the workhorse of data-parallel training: every GPU starts with a local gradient tensor of size M bytes and ends with the globally reduced (summed) result. The ring algorithm decomposes AllReduce into a ReduceScatter phase followed by an AllGather phase. Ring AllReduce time is given by equation D.3:

$$T_{\text{ring}} = 2 \cdot \frac{N-1}{N} \cdot \frac{M}{\beta} + 2(N-1) \cdot \alpha \tag{D.3}$$

The factor $2(N-1)/N$ in the bandwidth term approaches 2 as N grows—each byte effectively traverses the ring twice (once for reduce-scatter, once for all-gather). The latency term grows linearly with N, which makes the ring algorithm expensive in latency for large GPU counts.

For comparison, tree AllReduce has the cost shown in equation D.4:

$$T_{\text{tree}} = 2\log_2 N \cdot \frac{M}{\beta} + 2\log_2 N \cdot \alpha \tag{D.4}$$

The tree algorithm trades bandwidth efficiency for latency efficiency: the latency term grows as $\log_2 N$ instead of N, but the bandwidth term sends the full message M at each tree level instead of M/N chunks.

Napkin Math 21.2: Worked example: Ring vs. tree AllReduce

Setup: 256 H100 GPUs connected via InfiniBand NDR (α = 5 µs, β = 50 GB/s). The gradient tensor is 1 GB (a 500M-parameter model in FP16).

Question: *How* long does AllReduce take with ring vs. tree?

Ring AllReduce:

$T_{\text{ring}} \approx 42.4$ ms

Tree AllReduce:

$T_{\text{tree}} \approx 320.1$ ms

Analysis: For this 1 GB message, ring AllReduce takes 42.4 ms, and tree takes 320.1 ms—the ring is significantly faster because it distributes the bandwidth load across all links. The tree's logarithmic latency advantage is negligible compared to its bandwidth penalty for large messages.

Systems insight: For the large gradient tensors typical of data-parallel training, ring (or ring-based) algorithms dominate. Tree algorithms are useful only for small messages or when latency—not bandwidth—is the bottleneck.

D.2.2 AllGather

AllGather is the complement of ReduceScatter: each GPU starts with a M/N-sized shard and ends with the complete M-byte tensor. It is the core communication primitive in Fully Sharded Data Parallelism (FSDP) for reconstructing parameters before each forward pass, and in tensor parallelism for reassembling split activations.

Ring AllGather, given by equation D.5:

$$T_{\text{allgather}} = \frac{N-1}{N} \cdot \frac{M}{\beta} + (N-1) \cdot \alpha \tag{D.5}$$

AllGather is exactly half the cost of AllReduce (one ring pass instead of two) because it does not include a reduction step.

D.2.3 ReduceScatter

ReduceScatter is the dual of AllGather: each GPU starts with a full M-byte tensor and ends with a M/N-sized shard that contains the globally reduced values for that shard. It is the gradient synchronization primitive in FSDP and ZeRO, replacing the full AllReduce with a cheaper operation when each GPU only needs its own parameter shard's gradient.

Ring ReduceScatter, given by equation D.6:

$$T_{\text{reducescatter}} = \frac{N-1}{N} \cdot \frac{M}{\beta} + (N-1) \cdot \alpha \tag{D.6}$$

The symmetry is not a coincidence: ring AllReduce decomposes into one ReduceScatter followed by one AllGather, and the costs add up exactly to equation D.3.

D.2.4 AllToAll

AllToAll is the most general collective: each GPU sends a distinct M/N-sized chunk to every other GPU, and receives a distinct chunk from each. It is the routing primitive for Mixture-of-Experts (MoE) architectures, where tokens must be dispatched to the correct expert and the results gathered back.

Ring AllToAll, given by equation D.7:

$$T_{\text{alltoall}} = \frac{N-1}{N} \cdot \frac{M}{\beta} + (N-1) \cdot \alpha \tag{D.7}$$

Although the formula matches AllGather and ReduceScatter in form, the communication pattern is fundamentally different: AllToAll is a *personalized* exchange (each GPU sends different data to each peer), which makes it harder to overlap with computation and more sensitive to network congestion.

D.2.5 Broadcast

Broadcast sends a single M-byte tensor from one root GPU to all $N-1$ others. It is used for distributing initial model weights, updated learning rates, and configuration changes during training.

Tree Broadcast, given by equation D.8:

$$T_{\text{broadcast}} = \log_2 N \left(\alpha + \frac{M}{\beta} \right) \tag{D.8}$$

Broadcast is inherently asymmetric (one sender, many receivers), which makes a tree topology natural. In an unsegmented tree broadcast, the full M-byte payload is forwarded at each tree level, so both the latency and bandwidth terms appear along the $\log_2 N$-level critical path. Segmented or scatter-allgather broadcast variants can reduce the bandwidth term, but they require a different algorithm-specific model.

Collective complexity summary

With the individual formulas in place, table D.2 provides a compact reference for comparing the cost structure of each collective operation.

Table D.2: Collective Operation Cost Summary: Bandwidth and latency terms for ring and tree algorithms. The bandwidth term determines cost for large messages (gradients, parameters); the latency term determines cost for small messages (scalars, barriers). N = GPU count, M = total message size in bytes.

Operation	Bandwidth Term	Latency Term	Primary Use Case
AllReduce (Ring)	$2 \cdot \frac{N-1}{N} \cdot \frac{M}{\beta}$	$2(N-1) \cdot \alpha$	Data-parallel gradient sync
AllReduce (Tree)	$2 \log_2 N \cdot \frac{M}{\beta}$	$2 \log_2 N \cdot \alpha$	Small-message sync
AllGather (Ring)	$\frac{N-1}{N} \cdot \frac{M}{\beta}$	$(N-1) \cdot \alpha$	FSDP parameter reconstruction
ReduceScatter (Ring)	$\frac{N-1}{N} \cdot \frac{M}{\beta}$	$(N-1) \cdot \alpha$	FSDP/ZeRO gradient sharding
AllToAll (Ring)	$\frac{N-1}{N} \cdot \frac{M}{\beta}$	$(N-1) \cdot \alpha$	MoE expert routing
Broadcast (Tree)	$\log_2 N \cdot \frac{M}{\beta}$	$\log_2 N \cdot \alpha$	Weight distribution, config sync

Two patterns emerge from this table. First, bandwidth-optimal ring algorithms all share the $(N-1)/N$ factor, which approaches 1 for large N. Per-rank communication therefore approaches a constant multiple of M: about $2M$ for AllReduce and about M for AllGather, ReduceScatter, or AllToAll. Aggregate traffic still scales with the number of ranks. Second, the latency term grows linearly with N for ring algorithms but logarithmically for tree algorithms, creating the algorithm selection trade-off analyzed next.

D.3 Algorithm Selection

> Systems Perspective 21.3: Why this matters
>
> NCCL automatically selects a collective algorithm based on message size and GPU topology, but understanding *why* it chooses ring for large messages and tree for small ones helps diagnose when the auto-selection is suboptimal—and how to structure communication to stay in the efficient regime.

D.3.1 Ring vs. tree vs. recursive halving-doubling

The ring algorithm minimizes bandwidth usage but pays a latency cost that grows linearly with N. The tree algorithm minimizes latency but wastes bandwidth because it sends the full message at each level. Recursive halving-doubling splits the difference: it uses halving (like a tree) for the reduce phase and doubling for the gather phase, achieving near-optimal bandwidth with $\mathcal{O}(\log N)$ latency steps.

The decision boundary between ring and tree depends on message size:

- $M \gg n^*$ (large gradients): Use **ring**. The bandwidth term dominates, and ring's $(N-1)/N$ factor is optimal.
- $M \ll n^*$ (barriers, scalars): Use **tree**. The latency term dominates, and tree's $\log_2 N$ steps are far fewer than ring's $2(N-1)$.
- $M \approx n^*$ (medium tensors): Use recursive halving-doubling for balanced performance, or test empirically.

Table D.3 provides guidance for choosing the right algorithm.

Table D.3: Algorithm Selection Guide: Strengths and weaknesses of ring, tree, and recursive halving-doubling for different message sizes and GPU counts. $n^* = \alpha \times \beta$ is the crossover message size from equation D.2.

Regime	Ring	Tree	Recursive Halving-Doubling
Large messages ($M > 10 \times n^*$)	Best—optimal bandwidth utilization	Poor—sends M at every tree level	Good—near-optimal bandwidth, $\mathcal{O}(\log N)$ latency
Small messages ($M < n^*/10$)	Poor—$2(N-1)$ startup costs dominate	Best—$\log_2 N$ startup costs	Good—$\mathcal{O}(\log N)$ startup costs
Medium messages ($M \approx n^*$)	Acceptable	Acceptable	Best—balances both terms
Power-of-2 N	Works for any N	Best when $N = 2^k$	Requires $N = 2^k$

D.3.2 Hierarchical AllReduce

Modern clusters have a two-level topology: GPUs within a node are connected by NVLink (450 GB/s), while nodes are connected by InfiniBand (50 GB/s). Hierarchical AllReduce exploits this asymmetry in three phases:

1. **Intra-node reduce** (NVLink): Each node reduces its 8 local GPUs to a single result using NVLink's 450 GB/s bandwidth. This is fast because NVLink's per-direction rate is 9× that of an InfiniBand NDR port.
2. **Inter-node AllReduce** (InfiniBand): The per-node results are all-reduced across nodes. Only one representative per 8 local GPUs participates, so the inter-node ring has one-eighth as many participants as a flat GPU-level ring.
3. **Intra-node broadcast** (NVLink): Each node broadcasts the global result to its 8 local GPUs.

The hierarchical approach reduces the inter-node ring size from all GPUs to one representative per 8 local GPUs, cutting the latency-step term by about 8× and confining the bandwidth-hungry phases to the fast intra-node links. NCCL selects among topology-aware algorithms such as Ring, Tree, CollNet variants, NVLS/NVLSTree, and PAT based on the collective, message size, topology,

architecture, and version; hierarchical communication is common on multi-node systems but is not a universal default.

Understanding the cost structure of collectives enables reasoning about data-parallel overhead. Pipeline parallelism, however, introduces a different kind of overhead: idle time caused by the sequential dependency between pipeline stages. The next section quantifies that cost.

D.4 Pipeline Arithmetic

> Systems Perspective 21.4: Why this matters
>
> Pipeline parallelism splits a model across multiple GPUs so that each GPU holds only a fraction of the layers. The trade-off is **bubble time**: idle cycles where GPUs wait for activations or gradients from upstream or downstream stages. Quantifying the bubble fraction reveals whether pipeline parallelism is worth the complexity at a given scale.

D.4.1 Bubble fraction

In a pipeline with p stages and m microbatches, equation D.9 defines the **bubble fraction** as the proportion of total GPU-time spent idle:

$$b_{\text{pipe}} = \frac{p-1}{p-1+m} \tag{D.9}$$

This formula applies to both GPipe (where all forward passes complete before any backward pass begins) and 1F1B (one-forward-one-backward interleaving). The 1F1B schedule has the same asymptotic bubble fraction but uses less peak memory because it retires microbatches earlier.

The key insight: the bubble fraction depends on the *ratio* of stages to microbatches. Adding more microbatches (for a fixed number of stages) amortizes the pipeline fill and drain across more useful work.

Rule of thumb: Keeping bubble overhead at or below 20 percent requires $m \geq 4(p-1)$; strictly below 20 percent requires $m > 4(p-1)$. The practical rule $m \geq 4p$ is a convenient conservative target.

Table D.4 shows how bubble fraction varies with pipeline depth and microbatch count.

Table D.4: Pipeline Bubble Fraction: Percentage of GPU-time lost to pipeline bubbles for various stage counts and microbatch counts. Values below 20 percent (the typical target) require $m \geq 4p$.

Pipeline Stages (p)	m = 8	m = 16	m = 32	m = 64
p = 4	27.3%	15.8%	8.6%	4.5%
p = 8	46.7%	30.4%	17.9%	9.9%

D.4.2 Interleaved scheduling

Interleaved scheduling (used in Megatron-LM) assigns V **virtual stages** to each physical GPU instead of one. Each GPU processes V nonconsecutive chunks of the model, which means the pipeline has $p \times V$ virtual stages but the bubble overhead uses the original p, as equation D.10 shows:

$$b_{\text{interleaved}} = \frac{p-1}{p-1+m \times V} \tag{D.10}$$

The microbatch term in the denominator is multiplied by V compared to equation D.9, so the bubble reduction approaches V when m dominates $p-1$. For $p = 8$, $m = 16$, and $V = 4$, the bubble fraction drops from 30.4 percent to 9.9 percent, a 3.1× reduction that comes at the cost of more frequent, smaller communication between stages.

Interleaved scheduling highlights a recurring theme in distributed ML: communication granularity is a tunable knob. Smaller, more frequent transfers improve overlap and reduce idle time, but increase the aggregate latency overhead. The same trade-off appears in gradient compression, which is the final cost model in this appendix.

D.5 Compression Economics

> Systems Perspective 21.5: Why this matters
>
> Gradient compression promises to reduce communication volume by 10–1000×, making AllReduce nearly free. Compression, however, has overhead (encoding and decoding time), and it only helps if that overhead is smaller than the communication time it saves. A quick break-even analysis prevents wasted effort on compression schemes that will not actually speed up training.

D.5.1 The compression equation

Gradient compression replaces the original M-byte gradient with a compressed representation of size M/r_{comp}, where r_{comp} is the compression ratio. Equation D.11 expresses the total time with compression:

$$T_{\text{compressed}} = T_{\text{encode}} + T_{\text{transfer}}\left(\frac{M}{r_{\text{comp}}}\right) + T_{\text{decode}} \tag{D.11}$$

Compression is beneficial when $T_{\text{compressed}} < T_{\text{transfer}}(M)$. Rearranging this inequality yields the break-even condition in equation D.12:

$$T_{\text{encode}} + T_{\text{decode}} < T_{\text{transfer}}(M) - T_{\text{transfer}}\left(\frac{M}{r_{\text{comp}}}\right) \tag{D.12}$$

In the bandwidth-dominated regime ($M \gg n^*$), the communication time scales linearly with message size, and the right-hand side simplifies to approximately $T_{\text{transfer}}(M) \times (1 - 1/r_{\text{comp}})$. For high compression ratios ($r_{\text{comp}} \gg 1$), this approaches $T_{\text{transfer}}(M)$—meaning compression is worthwhile as long as the codec overhead is less than the uncompressed communication time.

> Napkin Math 21.3: Worked example: Is top-k compression worth it?
>
> **Setup**: Ring AllReduce of a 1 GB gradient across 256 H100 GPUs on InfiniBand NDR, taking 42.4 ms uncompressed. A Top-k sparsification scheme achieves 64× compression with 2 ms total encode + decode overhead.
>
> **Question**: Does compression reduce the total AllReduce time?
>
> **Math**:
>
> - Uncompressed AllReduce: 42.4 ms
> - Compressed communication: 3.2 ms
> - Total with compression: 5.2 ms
> - Speedup: 8.2×
>
> **Result**: Compression delivers an 8.2× speedup in communication time. The 2 ms codec overhead is a small fraction of the 42.4 ms uncompressed baseline.
>
> **Systems insight**: At this scale (256 GPUs, 1 GB gradients), gradient compression is clearly profitable. However, the convergence impact must be validated: Top-k sparsification discards information, and the model may require more iterations to reach the same accuracy, potentially negating the per-iteration speedup.

Table D.5 shows how the break-even codec overhead varies with compression ratio and uncompressed AllReduce time. Higher compression ratios tolerate larger codec overheads.

Table D.5: Compression Break-Even Thresholds: Maximum tolerable codec overhead (encode + decode) for compression to reduce total time, computed as $T_{\text{transfer}}(M) \times (1 - 1/r_{\text{comp}})$. At high compression ratios, nearly the entire uncompressed AllReduce time is available as codec budget.

Compression Ratio (r_{comp})	AllReduce = 10 ms (Max Codec Overhead)	AllReduce = 40 ms (Max Codec Overhead)	AllReduce = 100 ms (Max Codec Overhead)
4×	7.5	30	75
16×	9.4	37.5	93.8
64×	9.8	39.4	98.4
256×	9.96	39.8	99.6

Summary

Key Takeaways: Communication costs are predictable

- **The α-β model**: The model ($T(n) = \alpha + n/\beta$) decomposes every network transfer into a fixed startup cost and a payload-proportional bandwidth cost. The crossover message size $n^* = \alpha \times \beta$ determines which cost dominates.
- **Ring-based collectives**: AllReduce, AllGather, ReduceScatter, and AllToAll achieve bandwidth-optimal communication with a $(N-1)/N$ factor, but their latency term grows linearly with GPU count. Tree-based algorithms trade bandwidth for logarithmic latency scaling.
- **Algorithm selection**: Message size relative to n^* determines the algorithm: ring for large messages (gradients), tree for small messages (barriers), and hierarchical two-level schemes for multi-node clusters.
- **Pipeline bubble fraction**: $b_{\text{pipe}} = (p-1)/(p-1+m)$ is the fundamental overhead of pipeline parallelism. The practical rule $m \geq 4p$ keeps bubbles below 20 percent, and interleaved scheduling with V virtual stages makes the bubble reduction approach V when the microbatch term dominates.
- **Gradient compression**: Gradient compression is profitable when the codec overhead (encode + decode) is less than the communication time saved. At high compression ratios in the bandwidth-dominated regime, nearly the entire uncompressed AllReduce time is available as overhead budget—but convergence impact must always be validated.
- **Four parameters predict communication cost**: The point-to-point and collective communication models in this appendix derive from two hardware parameters (α and β) and two workload parameters (message payload n or M, plus participant count N for collectives). Measuring these quantities for a cluster yields a complete first-order prediction of distributed training overhead, with pipeline and compression sections introducing additional parameters as noted earlier.

E

Reliability Foundations

Purpose

How does failure become a statistical certainty at fleet scale, and what does the math require for recovery?

Individual accelerators fail rarely, but fleets multiply component failure rates until failures become a continuous operating condition. At cluster scale, the expected time between failures drops from years to hours, and recovery becomes part of normal execution rather than an exceptional incident. This appendix collects the reference calculations for reasoning quantitatively about failure, recovery, and availability at scale, providing the mathematical tools behind fault tolerance strategies, fleet orchestration policies, and operational practice. In C^3 terms, it develops the coordination axis, where checkpointing, recovery, and availability determine how much of the fleet's compute and communication becomes useful work.

How to Use This Appendix

This appendix is designed as a *reference*, intended for moving from intuition ("failures happen more often at scale") to quantitative engineering decisions ("how often should one checkpoint?" or "how many spare nodes are needed?").

The worked examples in this appendix use a canonical 10,000-GPU training fleet unless noted otherwise.

- **"How often will something fail?"**—Start with section E.1 and the MTBF cascade in section E.1.2.
- **"How often should I checkpoint?"**—Use the Young-Daly model in section E.2.1 and the worked example in section E.2.3.
- **"How much time do I lose to recovery?"**—See the recovery anatomy in section E.3.1 and the goodput analysis in section E.3.2.
- **"Should I use redundancy or checkpointing?"**—Compare strategies in section E.4.1 and availability stacking in section E.4.2.

E.1 Failure Probability at Scale

Consider a training run on a 10,000-GPU cluster running for 30 days. What is the probability that at least one modeled node component fails during that time? The answer—effectively 100 percent—determines whether the system needs fault tolerance as a core design requirement or merely a nice-to-have. The calculations in this section make that determination precise.

Individual hardware components are remarkably reliable. A data center-grade GPU operates for tens of thousands of hours before failing. The physics of large-scale systems, however, works against reliability: every additional component is another opportunity for failure, and the aggregate failure rate scales linearly with component count. This section develops the arithmetic that transforms component-level reliability into system-level failure predictions.

E.1.1 Component failure rates

Reliability engineers characterize components using two complementary metrics. The **Failure in Time (FIT)** rate counts failures per 10^9 device-hours of operation—a unit chosen because individual components fail so rarely that failures-per-hour would produce inconveniently small numbers.

The reciprocal quantity, **Mean Time To Failure (MTTF)**, gives the average lifetime in hours as equation E.1:

$$\text{MTTF} = \frac{10^9}{\text{FIT}} \tag{E.1}$$

Table E.1 lists reference FIT rates and MTTF values for components found in a typical GPU training node, grounded in large-fleet and warehouse-scale experience (Kokolis et al. 2025; Zu et al. 2024; Barroso et al. 2019). These values assume the steady-state "useful life" phase of the bathtub curve, where the failure rate is approximately constant—neither dominated by infant mortality (early life) nor wear-out (end of life) (Klutke et al. 2003).

Table E.1: Component Failure Rates: Order-of-magnitude reference FIT/MTTF in the steady-state useful-life phase. Informed by large-GPU research-cluster analysis Kokolis et al. (2025), TPUv4 supercomputer resiliency and operations Zu et al. (2024), and warehouse-scale machine design Barroso et al. (2019).

Component	FIT Rate	MTTF	MTTF (years)	Typical Failure Mode
GPU	20,000	50,000 hours	5.7 years	Die defect, thermal fatigue
HBM	5,000	200,000 hours	22.8 years	Bit-flip accumulation, TSV
NIC	6,666.7	150,000 hours	17.1 years	Transceiver degradation
PSU	10,000	100,000 hours	11.4 years	Capacitor aging
PCIe Switch	5,000	200,000 hours	22.8 years	Solder joint, ESD damage
Optical Cable	20,000	50,000 hours	5.7 years	Fiber bend, connector wear
ToR Switch	3,333.3	300,000 hours	34.2 years	ASIC failure, fan bearing

Each component in isolation appears highly reliable—a GPU lasts 5.7 years on average. The trouble begins when we ask how a *node* behaves with many such components operating simultaneously.

E.1.2 The MTBF cascade

A compute node is a series system: if *any* component fails, the node fails. For independent components with constant failure rates, equation E.2 sums the individual rates into the node-level failure rate:

$$\frac{1}{\text{MTBF}_{\text{node}}} = \frac{n_{\text{gpu}}}{\text{MTTF}_{\text{gpu}}} + \frac{n_{\text{nic}}}{\text{MTTF}_{\text{nic}}} + \frac{n_{\text{psu}}}{\text{MTTF}_{\text{psu}}} + \cdots \tag{E.2}$$

Think of each component as a ticking clock counting down to failure. A node with 8 GPUs, 2 NICs, and 2 PSUs has 12 independent clocks—the node fails when the *first* clock reaches zero. More clocks mean a shorter expected wait.

For a cluster of N_{nodes} identical nodes, the same logic applies one level up, as equation E.3 shows:

$$\text{MTBF}_{\text{cluster}} = \frac{\text{MTBF}_{\text{node}}}{N_{\text{nodes}}} \tag{E.3}$$

This is the **MTBF cascade**: reliability degrades linearly with component count at each level, and the levels compound. A node with 5,172.4 hours MTBF sounds reliable. A cluster of 1,250 such nodes has an MTBF of just 4.14 hours—a failure every few hours is the expected steady state.

Table E.2 shows how cluster MTBF shrinks as fleet size grows.

Table E.2: Cluster MTBF by Scale: As cluster size grows, the aggregate MTBF shrinks proportionally. At 10,000 GPUs, failures occur every few hours; at 100,000 GPUs, they occur continuously. Node configuration: 8 GPUs, 2 NICs, 2 PSUs per node.

Cluster GPUs	Nodes	Cluster MTBF	Expected Failures/Day
256	32	161.6 hours	0.1
1,024	128	40.4 hours	0.6
2,048	256	20.2 hours	1.2
8,192	1,024	5.1 hours	4.8
10,000	1,250	4.1 hours	5.8
100,000	12,500	24.8 minutes	58

The table makes a visceral point: the transition from "hundreds of GPUs" to "tens of thousands" is not merely a quantitative change but a qualitative one. At 256 GPUs, a full day may pass between failures. At 10,000 GPUs, the cluster expects multiple failures per shift. At 100,000 GPUs, failures are a continuous background condition—the system is never fully healthy.

E.1.3 Probability of failure during a job

Knowing the MTBF tells us the *average* time between failures, but training jobs have fixed durations. The question practitioners ask is: *what is the probability that my job will be interrupted at least once?*

Under the exponential failure model (constant failure rate), equation E.4 gives the probability of at least one failure during a job of duration T_{job}:

$$\Pr(\geq 1 \text{ failure}) = 1 - e^{-T_{\text{job}}/\text{MTBF}} \tag{E.4}$$

When $T_{\text{job}} \gg \text{MTBF}$, this probability approaches 1 rapidly. Table E.3 shows the concrete numbers for various cluster sizes and job durations.

Table E.3: Probability of At Least One Failure: For large clusters and multi-day jobs, failure is a near-certainty. Any system operating in the bottom-right region of this table must treat fault tolerance as a core design requirement, not an optimization.

Cluster GPUs	1 Day (24 h)	1 Week (168 h)	30 Days (720 h)
256	13.8%	64.6%	98.8%
1,024	44.8%	98.4%	> 99.9%
2,048	69.5%	> 99.9%	> 99.9%
8,192	99.1%	> 99.9%	> 99.9%
10,000	99.7%	> 99.9%	> 99.9%
100,000	> 99.9%	> 99.9%	> 99.9%

The message is stark: for any cluster above a few thousand GPUs running jobs longer than a day, the probability of experiencing at least one failure is effectively 100 percent. This is why Chapter 7 treats fault tolerance not as a defensive measure but as a fundamental architectural requirement.

The exponential failure model assumes a constant failure rate, which holds during the steady-state useful-life phase. During burn-in (first few hundred hours) and wear-out (approaching end-of-life), failure rates are higher. In practice, fleet operators observe that newly deployed nodes exhibit 2–3$\times$ higher failure rates in their first week, making burn-in testing essential before admitting nodes to production clusters.

The inevitability of failure during long training jobs leads directly to the next question: if we *will* lose progress, how do we minimize how much?

E.2 Checkpoint Optimization

Every checkpoint saves progress but costs time. Checkpoint too rarely and a failure destroys hours of training. Checkpoint too frequently and the overhead of writing checkpoints itself becomes the bottleneck. The Young-Daly formula gives the mathematically optimal balance point, and it depends on just two measurable quantities: how long a checkpoint takes to write and how often failures occur (Young 1974; Daly 2006).

E.2.1 The Young-Daly model

The optimal checkpoint interval balances two competing costs. Writing a checkpoint takes time T_{write} (the **checkpoint cost**), during which no useful training occurs. The longer the interval between checkpoints, however, the more work is lost when a failure strikes—on average, half the interval. Equation E.5 states the **Young-Daly formula** that minimizes the expected total overhead:

$$\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}} \tag{E.5}$$

where T_{write} is the checkpoint write time in seconds and $\text{MTBF}_{\text{system}}$ is the cluster or job-level system MTBF in seconds.

The intuition behind equation E.5 is geometric-mean-like: when checkpoints are cheap relative to the MTBF ($T_{\text{write}} \ll \text{MTBF}_{\text{system}}$, the common case), the optimal interval sits between the two time scales. If checkpoints took zero time, the optimal cadence is every step. If the system never failed, no checkpoints would be required. The square root interpolates between these extremes.

The formula assumes that failures follow an exponential distribution (memoryless property) and that checkpoint cost T_{write} is small compared to $\text{MTBF}_{\text{system}}$. Both assumptions hold well for production training clusters: the exponential model fits observed failure data, and modern checkpointing systems write to fast parallel storage in tens of seconds, while MTBF is measured in hours.

E.2.2 Checkpoint sizing

Checkpoint size determines the write time T_{write} that feeds into the Young-Daly formula. For common mixed-precision training checkpoints with the Adaptive Moment Estimation (Adam) optimizer, each parameter requires approximately 14 bytes of persistent state:

- 2 bytes for BF16 model weights
- 4 bytes for FP32 master weights
- 4 bytes for FP32 first moment (Adam m)
- 4 bytes for FP32 second moment (Adam v)

Equation E.6 aggregates these per-parameter costs into the total checkpoint footprint:

$$\text{Checkpoint Size} = P \times 14 \text{ bytes/parameter} \tag{E.6}$$

Here, P is the number of trainable model parameters whose persistent optimizer and weight state must be serialized.

Table E.4 shows how checkpoint state moves from manageable gigabytes to frontier-scale terabytes as model size grows.

Table E.4: Checkpoint Sizes for Mixed-Precision Adam Training: Each parameter requires about 14 bytes of persistent checkpoint state (model weights + FP32 master weights + optimizer moments); gradients are normally transient and recomputed after restore rather than serialized as durable checkpoint state. Write times assume 100 GB/s aggregate storage bandwidth.

Model Size	Checkpoint Size	Write Time at 100 GB/s
7B	98 GB	0.98 s
13B	182 GB	2 s
70B	980 GB	10 s
175B	2450 GB	24 s

Model Size	Checkpoint Size	Write Time at 100 GB/s
1T	14000 GB	140 s

As table E.4 shows, at frontier scale (175B+ parameters), checkpoint sizes reach the terabyte range. This makes checkpoint write time a significant cost that directly affects the Young-Daly optimal interval. The checkpoint strategies in Chapter 7 discuss techniques for reducing T_{write}—asynchronous checkpointing, incremental deltas, and distributed storage—all of which improve the Young-Daly result by shrinking the numerator under the square root.

E.2.3 Worked example: Optimal checkpoint interval

Combining the checkpoint write time with the 10,000-GPU MTBF gives a concrete checkpoint cadence.

Example 22.1: Young-Daly: 175B model on a 10,000-GPU cluster

Setup: Consider training a 175B-parameter model on a 10,000-GPU cluster. The cluster MTBF is 4.14 hours (table E.2). The checkpoint size is 2,450 GB, and the parallel storage system writes at 100 GB/s.

Step 1: Checkpoint write time (T_{write}).

$$T_{\text{write}} = \frac{\text{Checkpoint Size}}{\text{Write Bandwidth}} = \frac{2,450 \text{ GB}}{100 \text{ GB/s}} = 24.5 \text{ s}$$

Step 2: Apply the Young-Daly formula (equation E.5).

$$\tau_{\text{opt}} = \sqrt{2 \times T_{\text{write}} \times \text{MTBF}_{\text{system}}} = \sqrt{2 \times 24.5 \text{ s} \times 4.14 \text{ h} \times 3,600 \text{ s/h}} = 14.2 \text{ min}$$

Interpretation. The optimal checkpoint interval is approximately 14.2 minutes. The overhead from checkpointing alone is $T_{\text{write}}/\tau_{\text{opt}} \approx 2.9$ percent of training time.

Systems insight: If the cluster were doubled to 20,000 GPUs, the MTBF would halve, and the optimal interval would shrink to 10.1 minutes—checkpointing more frequently because failures happen more often. This illustrates the fundamental tension at scale: larger clusters are faster but demand more frequent interruption to protect progress.

The boundary conditions of the Young-Daly formula merit attention. When T_{write} approaches $\text{MTBF}_{\text{system}}$ (checkpoint cost approaches MTBF), checkpoint and rework overheads become so large that checkpoint/restart alone may fail to maintain useful forward progress. In such cases, redundancy or elastic training becomes necessary, as discussed in section E.4.1.

E.3 Recovery Budgets

When a failure occurs, the system does not instantly resume training. Detection, rescheduling, reloading state, and replaying lost work each consume time. Understanding this recovery anatomy reveals which phase dominates and where to invest engineering effort.

E.3.1 The anatomy of recovery time

Recovery is not a single event but a pipeline of phases, each with its own time budget. Equation E.7 sums them into total recovery time:

$$T_{\text{recovery}} = T_{\text{detect}} + T_{\text{reschedule}} + T_{\text{reload}} + T_{\text{replay}} \tag{E.7}$$

The terms are all durations: failure detection, replacement scheduling, checkpoint reload, and replay of lost training work since the last checkpoint. Their sum is the wall-clock time before the job returns to productive training.

Table E.5 breaks recovery into the phases that determine how long the cluster remains below full productivity after a failure.

Table E.5: Recovery Time Breakdown: Each phase contributes to the total time between failure and full-speed resumption. For the 10K-GPU, 175B-model scenario, replay dominates because it recomputes work lost since the last checkpoint.

Phase	Typical Duration	What Happens
T_{detect}	30 s	Heartbeat timeout expires; worker declared failed (confirmation adds to the budget in Chapter 7)
$T_{\text{reschedule}}$	60 s	Replacement node allocated from spare pool
T_{reload}	24.5 s	Checkpoint read from storage into GPU memory
T_{replay}	~7.1 min	Recompute training steps since last checkpoint
Total T_{recovery}	~9 min	System fully productive again

As table E.5 illustrates, the key insight is that T_{replay} typically dominates, and it is directly controlled by the checkpoint interval: on average, half the interval must be replayed. This creates a reinforcing loop with the Young-Daly formula—shorter intervals mean less replay but more checkpoint overhead, and the formula finds the minimum of this sum.

The other phases offer engineering optimization targets. T_{detect} can be reduced with more aggressive heartbeat intervals (at the cost of false positives). $T_{\text{reschedule}}$ depends on having hot spare nodes preallocated, a fleet orchestration decision covered in Chapter 8. T_{reload} scales with checkpoint size and storage bandwidth, motivating the checkpoint compression and sharding techniques discussed in Chapter 7.

E.3.2 Goodput vs. rawput

Not all time spent on a training cluster produces useful progress. **Rawput** is the total number of training steps executed (including steps that will be discarded after a failure). **Goodput** is the number of training steps that actually contribute to the final model:

$$\text{Goodput Ratio} = \frac{\text{Useful Steps}}{\text{Total Executed Steps}} \approx \frac{T_{\text{useful}}}{T_{\text{wall}}}$$

The gap between rawput and goodput comes from three sources:

1. **Checkpoint overhead** ($\sim$ 2.9 percent): Training pauses during each checkpoint write.
2. **Recovery overhead** ($\sim$ 3.6 percent): Time lost to detection, rescheduling, reloading, and replay after each failure.
3. **Wasted work**: Training steps computed between the last checkpoint and the failure, which must be discarded and recomputed.

At a 10,000-GPU scale, published reports from Meta, Google, and others consistently show 10–25 percent total overhead from failures and checkpointing combined. A cluster nominally capable of completing a training run in 30 days therefore requires 33–38 days of wall-clock time. The fleet orchestration strategies in Chapter 8 and Chapter 12 focus on narrowing this gap—every percentage point of overhead recovered translates directly to dollars saved and training time shortened.

> **Systems Perspective 22.1: The hidden cost of scale**
>
> A common misconception is that doubling cluster size halves training time. In practice, doubling from 5,000 to 10,000 GPUs halves the MTBF, roughly doubling the failure-related overhead. The effective speedup is less than 2×, and at extreme scale, adding more GPUs can actually *increase* wall-clock time if the fault tolerance mechanisms cannot keep pace. This is the reliability analogue of Amdahl's Law: the serial overhead of recovery bounds the benefit of parallelism.

E.4 Strategy Selection

Checkpoint/restart is not the only fault tolerance strategy. For serving workloads where downtime is measured in lost revenue, redundancy provides a fundamentally different trade-off. For elastic training, the system can shrink around failures rather than stopping. Choosing the right strategy depends on the workload's tolerance for latency, cost, and complexity.

E.4.1 Checkpoint/restart vs. redundancy vs. elastic training

The three canonical strategies represent different points in the trade-off space between cost, complexity, and recovery speed.

Checkpoint/restart periodically saves full system state and rolls back to the last checkpoint after failure. It is the workhorse of large-scale training: conceptually simple, well-understood, and effective when MTBF is much larger than checkpoint cost. The weakness is that recovery requires stopping all workers and replaying lost computation.

Redundancy maintains duplicate copies of state or computation. If one replica fails, another immediately takes over. This is the dominant strategy for inference serving, where even seconds of downtime are unacceptable. The cost is 2–3$\times$ the compute resources, which is prohibitive for training but justified for revenue-critical serving.

Elastic training allows the training job to continue with fewer workers when a failure occurs, rather than stopping entirely. Workers are added back when replacement nodes become available. This minimizes wall-clock interruption but requires frameworks that support dynamic world-size changes (for example, TorchElastic), and it introduces complexity in learning rate adjustment and gradient normalization. Table E.6 summarizes the trade-offs.

Table E.6: Fault Tolerance Strategy Comparison: Each strategy excels in a different regime. Real-world systems often combine strategies: checkpoint/restart for training with redundancy for the metadata service and checkpoint storage layer.

Criterion	Checkpoint/Restart	Redundancy	Elastic Training
Recovery latency	Minutes (replay)	Milliseconds (failover)	Seconds (reconfigure)
Resource overhead	~3–13% (storage + IO)	100–200% (replicas)	~5–10% (spare capacity)
Workload fit	Training (batch)	Serving (online)	Training (long-running)
Implementation	Simple	Moderate	Complex
State management	Periodic snapshots	Continuous replication	Distributed with resharding
Failure mode	Job pauses, replays	Transparent to user	Throughput dip, continues

E.4.2 The availability stacking formula

For serving workloads, availability is typically expressed as a percentage: 99 percent ("two nines"), 99.9 percent ("three nines"), and so on. Redundancy improves availability by running k independent replicas. The system is unavailable only when *all* replicas are simultaneously down, which equation E.8 captures:

$$A_{\text{system}} = 1 - (1 - A)^k \tag{E.8}$$

where A is the availability of a single replica and k is the number of replicas.

Table E.7 quantifies the availability gain from adding independent serving replicas.

Table E.7: Availability Stacking with Independent Replicas: Starting from a single-replica availability of 99 percent, each additional replica dramatically reduces expected downtime. Assumes replica failures are independent.

Replicas k	System Availability	Nines	Downtime per Year
1	99.00%	2.0	87.6 hours
2	99.9900%	4.0	52.6 minutes
3	99.9999%	6.0	31.5 seconds

As table E.7 shows, the power of stacking is dramatic: two replicas of a 99 percent-available system yield 99.99 percent availability, reducing annual downtime from roughly 87 hours to under an hour. This is why inference serving systems almost universally deploy multiple replicas behind a load balancer—the cost of an extra replica is small compared to the business value of four-nines availability.

The independence assumption is critical, however. Correlated failures—power outages affecting an entire rack, software bugs triggered by a specific input, or network partitions isolating a failure domain—defeat availability stacking. This is why Chapter 7 emphasizes *failure domain isolation*: replicas must be placed in different racks, different power zones, and ideally different data centers to ensure that their failure modes are truly independent.

Summary

Key Takeaways: Failure as a physical constraint

- **Failure rate scales linearly with component count**: A single GPU fails once per 5.7 years; a 10,000-GPU cluster experiences a failure every 4.14 hours. At fleet scale, failure is a continuous background condition, not an exceptional event.
- **The MTBF cascade compounds through system levels**: Node MTBF is determined by the weakest component type; cluster MTBF divides by node count. Table E.2 provides the reference numbers for capacity planning.
- **Job failure probability approaches certainty quickly**: For clusters above a few thousand GPUs running multi-day jobs, the probability of at least one failure exceeds 99 percent. Fault tolerance is not optional at this scale—it is a prerequisite for completing any training run.
- **Young-Daly checkpoint interval**: The formula $\tau_{\text{opt}} = \sqrt{2T_{\text{write}} \text{MTBF}_{\text{system}}}$ optimizes checkpoint frequency by balancing the cost of writing checkpoints against the cost of lost work. It requires only two measurable inputs: checkpoint write time and system MTBF.
- **Recovery has four phases**: Detection, rescheduling, reloading, and replay. Replay typically dominates and is controlled by the checkpoint interval. Each phase offers distinct optimization opportunities.
- **Strategy selection depends on workload type**: Checkpoint/restart suits batch training. Redundancy suits latency-sensitive serving. Elastic training bridges the two but adds complexity.
- **Availability stacks exponentially with independent replicas**: Independent replicas improve availability exponentially, but correlated failures collapse that benefit. Failure domain isolation is the prerequisite that makes redundancy effective.

F

Inference Foundations

Purpose

How do queuing dynamics and memory capacity determine the throughput and latency of a production inference service?

Serving a model at scale is governed by the mathematics of waiting lines and the physics of the key-value cache. This appendix develops the quantitative tools that the inference and operations chapters draw on: a queuing-theoretic model of batched serving that yields practical batch sizes under stochastic arrivals, and the capacity arithmetic of the attention cache that frequently becomes the binding constraint on serving throughput. These are not background details but first-order constraints on how many requests a fleet can serve within a latency budget. In C^3 terms, inference turns compute capacity, communication with memory and clients, and coordination across queues into one serving constraint.

How to Use This Appendix

This appendix is a quantitative reference for serving systems. Use the queuing model when you need to choose a batch size or predict tail latency under a given arrival rate; use the key-value cache arithmetic when you need to size memory or find the point at which the cache, rather than compute, caps throughput.

Reach for it from the inference and operations chapters: when those chapters cite a serving result, the derivation lives here. Read it linearly for the full argument, or jump to the worked GPT-3 example and the batch-size decision framework when you need a template to apply to your own system.

F.1 Serving performance analysis

Serving performance analysis has two coupled layers. The queue determines how requests wait, batch, and shed under stochastic arrivals; the key-value cache determines how much live state the accelerator can hold while meeting the latency budget. The subsections that follow derive the queueing model first, then connect that model to memory-capacity limits in the worked serving examples.

F.1.1 Queuing theory for batched inference

The batching efficiency curve provides intuition about throughput-latency trade-offs, but production systems require rigorous analysis to determine optimal batch sizes under stochastic arrival patterns. Queuing theory provides the mathematical framework to derive these optimal operating points and to understand why certain batch sizes outperform others under specific conditions.

F.1.1.1 The M/G/c/K queue model for GPU serving

GPU inference systems can be modeled as **M/G/c/K queues**, a standard notation from queueing theory that captures the essential characteristics of production serving systems:

- **M (Markov arrivals)**: Requests arrive according to a Poisson process with rate λ_{arr}. This models the memoryless property of user requests, where arrival of one request does not predict the timing of the next.
- **G (General service distribution)**: Service times follow a general distribution, not restricted to exponential. GPU inference times depend on batch size and exhibit deterministic components (compute) mixed with stochastic variation (memory contention, kernel scheduling).

- **c (Number of servers)**: The system has c parallel GPU workers, each capable of serving requests independently.
- **K (Queue capacity)**: The system maintains a finite queue of capacity K requests. Requests arriving to a full queue are rejected (load shedding).

The notation comes from the same systems-performance lineage that sized telephone networks before it was applied to GPU clusters.

> Systems Perspective 23.1: Queuing theory and performance
>
> Queuing theory, developed by Agner Krarup Erlang in 1909 for telephone network analysis, remains foundational to systems performance engineering (Erlang 1909). The same mathematical framework that sized telephone exchanges now determines GPU cluster capacity. The M/G/c/K model is standard in systems textbooks, appearing in Jain's *The Art of Computer Systems Performance Analysis* (Jain 1991) and Kleinrock's *Queueing Systems* (Kleinrock 1975).

For a batched inference system with batch size B, service time becomes a function of batch size. Let $T_{\text{svc}}(B)$ denote the time to process a batch of B requests. From the physics of batching (section 10.3), equation F.1 models this as:

$$T_{\text{svc}}(B) = T_{\text{fixed}} + t_{\text{req}} \cdot B \tag{F.1}$$

where T_{fixed} represents fixed service overhead (kernel launch, weight loading from HBM) and t_{req} represents marginal per-request computation time. This linear model captures the first-order behavior observed in production systems, though actual service times may exhibit slight sublinearity due to memory bandwidth saturation at large batch sizes.

The **effective service rate** for batched processing is:

$$\mu_{\text{eff}}(B) = \frac{B}{T_{\text{svc}}(B)} = \frac{B}{T_{\text{fixed}} + t_{\text{req}} \cdot B}$$

The effective rate increases with batch size, approaching the asymptotic limit $1/t_{\text{req}}$ as $B \to \infty$.

F.1.1.2 Response time analysis

The total response time T for a request consists of three components:

$$E[T] = E[W] + E[T_{\text{batch}}] + E[T_{\text{svc}}]$$

where:

- $E[W]$ is the expected waiting time in queue before joining a batch
- $E[T_{\text{batch}}]$ is the expected time to form a complete batch (batch accumulation delay)
- $E[T_{\text{svc}}]$ is the expected inference time once the batch executes

For dynamic batching with maximum wait time T_{max} and maximum batch size B_{max}, the batch formation process is bounded. Under Poisson arrivals with rate λ_{arr}, the expected number of requests accumulated in time T_{max} is $\lambda_{\text{arr}} \cdot T_{\text{max}}$. The actual batch size B follows:

$$E[B] = \min(B_{\text{max}}, \lambda_{\text{arr}} \cdot T_{\text{max}})$$

The batch accumulation delay depends on whether the batch fills by reaching B_{max} or by timeout:

$$E[T_{\text{batch}}] = \begin{cases} \frac{B_{\text{max}}}{2\lambda_{\text{arr}}} & \text{if } \lambda_{\text{arr}} \cdot T_{\text{max}} \geq B_{\text{max}} \text{ (batch fills)} \\ \frac{T_{\text{max}}}{2} & \text{if } \lambda_{\text{arr}} \cdot T_{\text{max}} < B_{\text{max}} \text{ (timeout triggers)} \end{cases}$$

The factor of $1/2$ in both cases reflects the average wait for requests arriving uniformly throughout the batch window.

F.1.1.3 Optimal batch size derivation

The optimal batch size B minimizes expected response time subject to throughput requirements. We seek:

$$B = \arg\min_B E[T(B)] \quad \text{subject to} \quad \mu_{\text{eff}}(B) \geq \lambda_{\text{arr}}$$

The constraint ensures system stability (service rate exceeds arrival rate).

Substituting the response time components:

$$E[T(B)] = E[W(B)] + \frac{B}{2\lambda_{\text{arr}}} + T_{\text{svc}}(B)$$

For an M/G/1 queue with batch arrivals (treating each batch as a single "super-request"), the queue sees batches rather than individual requests. If the request arrival rate is λ_{arr} and the average batch size is B, the batch arrival rate is $\lambda_{\text{batch}} = \lambda_{\text{arr}}/B$. The Pollaczek-Khinchine formula gives the expected waiting time:

$$E[W] = \frac{\lambda_{\text{batch}} \cdot E[T_{\text{svc}}^2]}{2(1-\rho_{\text{serv}})}$$

where $\rho_{\text{serv}} = \lambda_{\text{batch}} \cdot E[T_{\text{svc}}] = \lambda_{\text{arr}} \cdot E[T_{\text{svc}}]/B$ is the server utilization. The second moment $E[T_{\text{svc}}^2]$ captures service time variability.

For our linear service time model with deterministic service (variance zero within a batch):

$$E[W(B)] = \frac{\lambda_{\text{arr}} \cdot T_{\text{svc}}(B)^2}{2B(1-\lambda_{\text{arr}} \cdot T_{\text{svc}}(B)/B)}$$

For production sizing, teams often first enforce a utilization headroom target rather than solving the full latency minimization problem. With $T_{\text{svc}}(B) = T_{\text{fixed}} + t_{\text{req}}B$, the constraint $\rho_{\text{serv}} = \lambda_{\text{arr}}T_{\text{svc}}(B)/B \leq \rho_{\text{target}}$ gives equation F.2:

$$B_{\text{min}}(\rho_{\text{target}}) = \frac{\lambda_{\text{arr}}T_{\text{fixed}}}{\rho_{\text{target}} - \lambda_{\text{arr}}t_{\text{req}}} \tag{F.2}$$

where ρ_{target} is the target utilization (typically 0.7–0.8 for production systems to maintain latency headroom) and $\rho_{\text{target}} > \lambda_{\text{arr}}t_{\text{req}}$.

> **Systems Perspective 23.2: Target-utilization batch sizing**
>
> Equation F.2 reveals a fundamental insight: *minimum stable batch size grows with fixed overhead and arrival rate*. This means:
>
> 1. **Higher traffic** (λ_{arr}): Required batch size increases
> 2. **Higher fixed overhead** (T_{fixed}): Required batch size increases to amortize overhead
> 3. **Higher target utilization** (ρ_{target}): Required batch size decreases, but with less latency headroom
>
> This utilization-constrained sizing explains why LLMs (high T_{fixed} from weight loading) benefit from larger batches than vision models (low T_{fixed}), even at the same arrival rate.

F.1.1.4 Worked example: GPT-3 serving at 100 QPS

Consider serving a GPT-3 class model (175B parameters) under the following system parameters:

- Arrival rate: $\lambda_{\text{arr}} = 100$ requests/second
- Hardware: 8× A100 GPUs with tensor parallelism
- Weight loading overhead: $T_{\text{fixed}} = 50$ ms (time to load attention matrices per forward pass)
- Per-token compute: $t_{\text{req}} = 0.5$ ms per request (amortized across batch)
- Average output length: 100 tokens per request

- Target utilization: $\rho_{\text{target}} = 0.75$

For the prefill phase (processing input prompt), service time follows equation F.1:

$$T_{\text{svc}}(B) = 50 + 0.5 \cdot B \text{ ms}$$

Applying equation F.2 to compute the minimum batch size that preserves 75 percent utilization headroom:

$$B_{\text{min}} \approx \frac{100 \times 0.05}{0.75 - 100 \times 0.0005} = \frac{5}{0.70} \approx 7.14$$

Rounding up gives $B = 8$ as the smallest batch size that meets the target utilization headroom. Evaluating the full response-time expression then shows how larger batches reduce utilization but add batch-accumulation delay:

Performance at different batch sizes.

Table F.1 quantifies the trade-offs across batch sizes from 1 to 32:

Table F.1: Batch Size Impact on GPT-3 Serving Performance: Batch sizes below 8 cannot sustain 100 QPS (utilization exceeds 100 percent). After including batch-accumulation delay, the latency minimum in this simple model occurs near $B = 8$; $B = 16$ has nearly the same mean latency with lower utilization, while $B = 32$ leaves more headroom but pays a large accumulation-delay penalty.

Size B	$T_{\text{svc}}(B)$ (ms)	req/s	ρ_{serv}	EW (ms)	$B/(2\lambda_{\text{arr}})$ (ms)	ET (ms)
1	50.5	19.8	505% (unstable)	∞	5.0	∞
4	52.0	76.9	130% (unstable)	∞	20.0	∞
8	54.0	148.1	67.5%	56.1	40.0	150.1
16	58.0	275.9	36.3%	16.5	80.0	154.5
32	66.0	484.8	20.6%	8.6	160.0	234.6

Analysis.

1. **Batch sizes 1–4 are unstable**: Utilization exceeds 100 percent, meaning the system cannot keep up with arrivals. Queues grow unboundedly.
2. **Batch size 8 achieves stability**: At 67.5 percent utilization, the system is stable but queuing delays contribute significantly to latency (56.1 ms), as figure F.1 illustrates.
3. **Batch sizes 8–16 form the latency knee**: Batch size 8 gives the lowest mean latency in this model. Batch size 16 trades a few milliseconds of additional total latency for lower utilization and queue wait time (16.5 ms vs. 56.1 ms), while $B = 32$ leaves still more utilization headroom but pays too much batch-accumulation delay.
4. **Diminishing returns** beyond $B = 32$: Further batch size increases would reduce utilization but memory constraints prevent exploration.

We can confirm these results by applying Little's Law as an internal consistency check.



Napkin Math 23.1: Applying Little's Law to verify

Verification using Little's Law (Little 1961): $Q_{\text{req}} = \lambda_{\text{arr}} \cdot T_{\text{lat}}$

At $B = 16$ with $\lambda_{\text{arr}} = 100$ req/s and $E[T] = 154.5$ ms:

$Q_{\text{req}} = 100 \times 0.1545$ s = 15.45 requests in system

With batch size 16 and utilization 36.3 percent:

- Expected requests in service: 16 × 0.363 = 5.8
- Expected requests in queue: 15.45 - 5.8 = 9.64

This matches the response-time decomposition: roughly 6 requests are in service and roughly 10 are waiting or accumulating into batches on average.

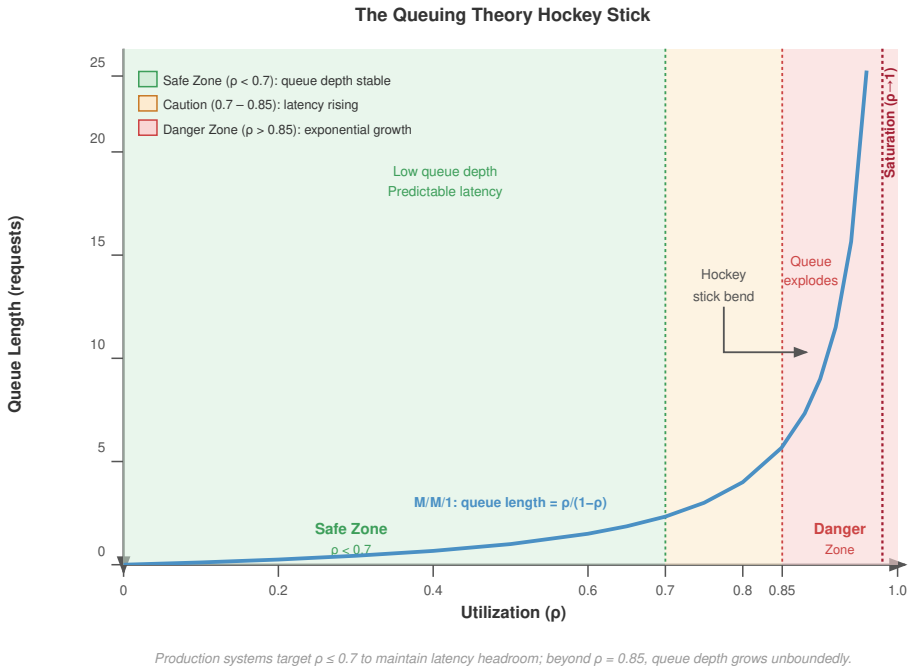


Figure F.1: The Queuing Hockey Stick: Relationship between system utilization ρ_{serv} and queue length (M/M/1 queue length $\rho_{\text{serv}}/(1-\rho_{\text{serv}})$). Three zones are shaded: a Safe Zone ($\rho_{\text{serv}} < 0.7$), a Caution band (0.7–0.85), and a Danger Zone ($\rho_{\text{serv}} > 0.85$) where queue depth grows unboundedly. Production systems typically target $\rho_{\text{serv}} \leq 0.7$ to maintain latency headroom.

F.1.1.5 Decision framework: Batch size selection given SLA

Production systems must select batch size to meet Service Level Objectives (SLOs), typically specified as latency percentiles (for example, P99 latency < 200 ms). The following framework systematizes this decision:

Step 1: Characterize service time Measure T_{fixed} and t_{req} empirically by profiling inference at batch sizes 1, 8, and 32. Fit the linear model $T_{\text{svc}}(B) = T_{\text{fixed}} + t_{\text{req}} B$.

Step 2: Compute stability threshold Find minimum batch size $B_{\min}$ such that $\mu_{\text{eff}}(B_{\min}) > \lambda_{\text{arr}}$:

$$B_{\min} = \frac{T_{\text{fixed}} \lambda_{\text{arr}}}{1 - t_{\text{req}} \lambda_{\text{arr}}}$$

Any batch size below $B_{\min}$ results in an unstable system.

Step 3: Compute latency at candidate batch sizes For each candidate $B \in \{B_{\min}, 2B_{\min}, \ldots, B_{\max}\}$, compute:

$$E[T(B)] = \frac{\lambda_{\text{arr}} \cdot T_{\text{svc}}(B)^2}{2B(1 - \lambda_{\text{arr}} T_{\text{svc}}(B)/B)} + \frac{B}{2\lambda_{\text{arr}}} + T_{\text{svc}}(B)$$

Step 4: Account for tail latency For P99 SLO compliance, use the heavy-traffic approximation for tail latency:

$$T_{\text{P99}} \approx E[T] + 2.33 \cdot \sigma_T$$

where σ_T is the standard deviation of response time. For exponential-like response times, P99 is closer to $-\ln(0.01)E[T] \approx 4.6 \cdot E[T]$; use measured percentiles for production sizing.

Step 5: Select optimal batch size Choose the largest B such that $T_{\text{P99}}(B) \leq \text{SLO}$:

$$B = \max\{B : T_{\text{P99}}(B) \leq \text{SLO}\}$$

Table F.2 summarizes the decision process:

Table F.2: Batch Size Decision Framework: Systematic approach to selecting batch size based on stability, latency SLO, and utilization targets.

Condition	Recommended Action	Rationale
$B < B_{\text{min}}$	Increase batch size or add replicas	System is unstable
$T_{\text{P99}} > \text{SLO}$	Reduce batch size or add replicas	Latency exceeds target
$\rho_{\text{serv}} < 0.5$	Consider reducing replicas to save cost	System is overprovisioned
$\rho_{\text{serv}} > 0.85$	Add replicas for headroom	Approaching instability

F.1.1.6 Trade-off curves: Visualizing the operating region

The relationship between batch size, throughput, and latency defines an operating region within which production systems must function. Figure F.2 illustrates this region:

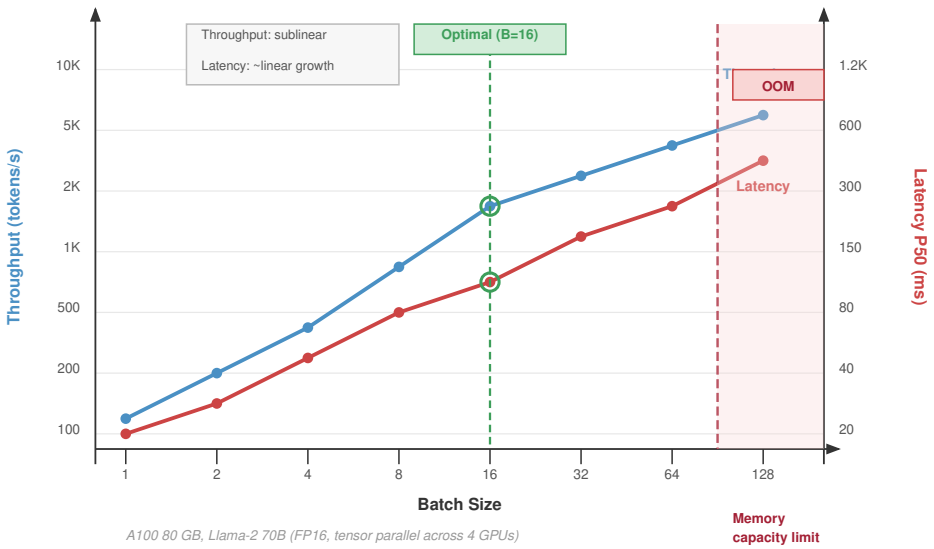


Figure F.2: Batch Size Trade-Off Curves: As batch size grows on the x-axis, throughput (left y-axis, blue) grows sublinearly while latency P50 (right y-axis, red) grows near-linearly. An optimal point is marked at B=16; a shaded OOM/memory-capacity region flags batch sizes that exceed device memory. Benchmark: Llama-2 70B in FP16 tensor-parallel across four A100 80 GB GPUs.

The trade-off curve demonstrates several key insights:

1. **Pareto frontier**: The curve represents efficient operating points; any point below the curve is dominated by a point on the curve with either higher throughput or lower latency.
2. **Knee of the curve**: The optimal batch size often lies at the "knee" where throughput gains diminish while latency continues to increase linearly.
3. **SLO-constrained optimum**: When an SLO bounds maximum latency, the optimal point is where the curve intersects the SLO boundary.

4. **Diminishing returns**: Beyond the knee, doubling batch size may increase throughput by only 10–20 percent while doubling latency.

The queuing-theoretic framework provides the mathematical foundation for the batching strategies examined in the following sections. Where intuition might suggest "larger batches are better for throughput," stability constraints, latency targets, and diminishing returns create a well-defined optimal operating region that varies by model type and deployment requirements.

F.1.2 KV cache fundamentals

The formal definition of the KV cache appears in the main inference memory-management section (section 10.2.2). This appendix expands the sizing math behind that definition: why the cache reduces per-token compute, why its memory grows with sequence length and batch size, and why unmanaged allocation can fragment HBM (Kwon et al. 2023).

Autoregressive generation without caching requires $\mathcal{O}(t^2)$ computation per token because each transformer layer must recompute attention keys and values for all previous tokens. The KV cache stores these computed key and value vectors, reducing generation to $\mathcal{O}(t)$ per token. For serving at scale, this compute saving creates a critical memory-management challenge since KV cache memory can exceed model weights for long contexts.

The cache size grows with context as calculated by equation F.3:

$$M_{\text{KV}} = 2 \times N_L \times d \times S \times B \times s_{\text{elem}} \tag{F.3}$$

where:

- N_L = number of layers
- d = hidden dimension
- S = sequence length
- B = batch size
- s_{elem} = storage size per element in bytes
- Factor of 2 accounts for both keys and values

Summary

Key Takeaways: Serving performance is bounded by queues and memory

- **Utilization has a hard ceiling**: As utilization ρ_{serv} approaches 1, queue depth and latency grow without bound. Production serving targets $\rho_{\text{serv}} \leq 0.7$ to preserve latency headroom; the remaining 30 percent is not waste but the buffer that keeps tail latency finite.
- **Batch size has an optimum, not a maximum**: Larger batches raise throughput but add batch-accumulation delay, producing a latency knee rather than monotonic improvement. Beyond the knee, doubling the batch can buy only 10–20 percent more throughput at the cost of doubled latency.
- **Little's Law ties the system together**: $Q_{\text{req}} = \lambda_{\text{arr}} T_{\text{lat}}$ relates the number of in-flight requests to the arrival rate and time-in-system, giving a one-line consistency check on any serving measurement.
- **The KV cache is the binding constraint**: Cache memory scales linearly with sequence length and batch size and routinely exceeds the model weights, capping concurrent capacity. Naive allocation wastes 60–80 percent to fragmentation, which is why production systems manage it with a virtual-memory scheme.

G

System Assumptions

Purpose

What assumptions sit underneath the distributed systems calculations throughout this book?

Every quantitative example in this volume, from cluster failure rates to collective-communication costs to carbon footprint estimates, uses a specific set of assumed numbers: canonical cluster sizes, fabric bandwidths, accelerator reliability, power usage effectiveness, regional carbon intensity, model utilization ranges, and dozens more. This appendix collects those fleet-scale assumptions in one place so the book's arithmetic can be audited, checked against chapter napkin math, or substituted with local values; the tables cover node-level accelerator specs, cluster tiers, network fabrics, reliability and recovery parameters, communication models, sustainability, cloud economics, capacity-planning utilization, unit conventions, and the provenance catalog that ties assumptions back to vendor datasheets, fleet studies, grid reports, and book conventions. In C^3 terms, the assumptions keep compute capacity, communication cost, and coordination overhead measured on the same scale across chapters.

How to Use This Appendix

Use this appendix when you want to verify a fleet-scale estimate, swap in an alternative assumption (for example, NDR vs. HDR fabric, Quebec vs. Poland carbon intensity, or a different MFU), or see which numbers a chapter's calculation depends on. Find the relevant section, read the `Value` and `Unit` columns, and plug them into your own arithmetic.

A short napkin-math callout at the start shows how several assumptions combine (failure rate, AllReduce time, carbon footprint). Hardware and link bandwidths are ceilings unless a chapter states an effective-bandwidth discount. Treat the tables as this edition's shared assumption sheet—not immutable physical laws.

Napkin Math 24.1: Quick calculations with these assumptions

These tables support quick distributed-systems napkin math. Three examples show how several assumptions combine.

Problem: Estimate how many GPU failures a cluster should expect per day.

Variables: Use a 8,192 GPUs cluster and an individual GPU MTTF of 50,000 hours.

Math: The GPU failure rate is 8,192 GPUs/50,000 hours $\approx$ 0.16 failures/hour, or roughly ~3.9 failures/day.

Result: This estimate excludes NIC, PSU, and cable failures. At 100,000 GPUs, the GPU-only rate is approximately ~48 failures/day; adding NIC, PSU, and cable failures can roughly double the operational incident stream.

Systems insight: At fleet scale, failure becomes a continuous background process rather than an exceptional event.

Problem: Estimate how long an AllReduce takes for a 70B model.

Variables: Use BF16 parameters and InfiniBand NDR with 50 GB/s effective bandwidth per port.

Math: The gradient payload is $70 \times 10^9 \times 2$ bytes = 140 GB. The ring AllReduce napkin estimate is 2×140 GB/50 GB/s ≈ 5.6 s.

Result: The collective can consume a significant fraction of a training step.

Systems insight: Pipeline and tensor parallelism exist partly to shrink the gradient volume each rank must communicate.

Problem: Estimate the carbon footprint of a 10,000 GPUs training run.

Variables: Each H100 draws 700 W at TDP. Use PUE 1.12, Quebec grid intensity 20 g/kWh, and Poland grid intensity 820 g/kWh.

Math: Total facility power is 10,000 GPUs $\times$ 700 W $\times$ 1.12 = 7.84 MW. In Quebec, 7.84 MW $\times$ 720 h/month $\times$ 20 g/kWh = 112.9 t CO_2 per month.

Result: The same run in Poland emits 4628.7 t, a 41× difference.

Systems insight: Grid carbon intensity can dominate the emissions difference for the same hardware, model, and training duration.

Foundational Hardware Recap

Distributed-systems reasoning in this book builds on single-node performance bounds. Table G.1 recaps the H100 assumptions used as the primary accelerator reference in this volume.

Table G.1: Single-node H100 assumptions: Peak specs from (Choquette 2023) and NVIDIA H100 documentation. Provide R_{peak} and BW baselines for iron-law calculations.

Category	Assumption	Value
Compute	H100 FP16 peak throughput	989 TFLOP/s
Compute	H100 FP8 peak throughput	1,979 TFLOP/s
Memory	HBM3 bandwidth	3.35 TB/s
Memory	HBM3 capacity	80 GB
Thermal	TDP	700 W

Cluster Reference Configurations

This book uses four **canonical cluster tiers** (256, 2,048, 8,192, and 100,000 GPUs) to illustrate how system behavior changes across scale. They are editorial reference points aligned with common research-lab, production, large-training, and hyperscale fleets—not measurements from a single deployment. Table G.2 defines GPU counts; chapters derive node counts, failure rates, and network requirements from these baselines and the assumptions in subsequent tables.

Table G.2: Canonical Cluster Sizes: Book-convention tier sizes for fleet-scale examples. At 8 per node: 32 nodes, 256 nodes, 1,024 nodes, and 12,500 nodes. Failure handling shifts from exception to steady state between the 256- and 8,192-GPU tiers (Kokolis et al. 2025).

Assumption	Value	Unit
Small cluster GPU count (256 tier)	256	GPUs
Medium cluster GPU count (2,048 tier)	2048	GPUs
Large cluster GPU count (8,192 tier)	8192	GPUs
Mega cluster GPU count (100,000 tier)	100000	GPUs

The relationship between cluster size and system behavior is not linear. A 256-GPU cluster might go days between failures; a 100,000-GPU cluster experiences multiple failures per hour. This

nonlinearity is why the chapters on fault tolerance (Chapter 7) and fleet orchestration (Chapter 8) treat failure as a continuous background process rather than an exceptional event.

With cluster sizes established, the next question is: what connects the machines? The network fabric determines whether gradient synchronization takes milliseconds or seconds, and whether pipeline parallelism is viable across node boundaries.

Network and Interconnect Specifications

Distributed training and inference are bottlenecked by data movement between machines as often as by computation within them. **Network assumptions** fix bandwidth and latency for every interconnect tier—from NVLink/PCIe within a node (NVIDIA Corporation 2017, 2020; Choquette 2023), through InfiniBand, RDMA, and Ethernet fabrics (InfiniBand Trade Association 2000; Gangidi et al. 2024; NVIDIA 2026c; Ethernet Alliance 2025), to the speed-of-light WAN floor (physics identity).

Intra-node interconnects

Within a single node, GPUs communicate over NVLink or PCIe. These bandwidths determine whether tensor parallelism (which requires high-bandwidth, low-latency communication) is viable within the node boundary. Table G.3 lists the intra-node interconnect bandwidths and latencies used throughout this book. These values are collected here for convenience and reused throughout the fleet-scale chapters.

Table G.3: Intra-Node Interconnect Bandwidth and Latency: NVLink/PCIe peaks from vendor documentation (NVIDIA Corporation 2017, 2020; Choquette 2023). NVLink is 7–10× faster than same-generation PCIe, which is why tensor parallelism stays intra-node.

Assumption	Value	Unit
NVLink bandwidth (V100)	300	GB/s
NVLink bandwidth (A100)	600	GB/s
NVLink bandwidth (H100)	900	GB/s
PCIe Gen3 bandwidth (V100 host)	15.75	GB/s
PCIe Gen4 bandwidth (A100 host)	32	GB/s
PCIe Gen5 bandwidth (H100 host)	64	GB/s
NVLink one-way latency	500	ns
PCIe Gen5 one-way latency	1000	ns

Inter-node interconnects

Once data crosses the node boundary, bandwidth drops by an order of magnitude and latency increases by 5–10×. This cliff shapes every decision about parallelism strategy: operations that require frequent, fine-grained communication (tensor parallelism) stay within the node, while operations with coarser communication patterns (data parallelism, pipeline parallelism) span nodes. Table G.4 lists both the bit-rate and byte-rate forms of each fabric, since different chapters use different conventions.

Table G.4: Inter-Node Network Bandwidth and Latency: Fabric link rates from (InfiniBand Trade Association 2000) and product-line specifications. The 9× NVLink-vs-NDR gap explains topology-aware parallelism (Chapter 6, Chapter 5).

Assumption	Value	Unit
InfiniBand HDR link bandwidth (bit rate)	200	Gb/s
InfiniBand HDR bandwidth (byte rate)	25	GB/s

Assumption	Value	Unit
InfiniBand NDR link bandwidth (bit rate)	400	Gb/s
InfiniBand NDR bandwidth (byte rate)	50	GB/s
InfiniBand XDR bandwidth (byte rate)	100	GB/s
400 GbE bandwidth (byte rate)	50	GB/s
800 GbE bandwidth (byte rate)	100	GB/s
100 GbE RoCE bandwidth (byte rate)	12.5	GB/s
InfiniBand one-way latency	5000	ns

Wide-area network

Cross-region communication introduces latency floors set by the speed of light in optical fiber. For a 5000 km link (roughly New York to London), the minimum one-way latency is 5000 km/200,000 km/s = 25 ms—orders of magnitude higher than intra-data-center latency. This physical constraint is why geo-distributed training uses asynchronous methods or careful placement of pipeline stages, and why federated learning (Chapter 11) tolerates stale gradients.

Network specifications define how fast data moves, but they say nothing about how often the machines sending that data fail. The next section quantifies the reliability of individual components—the building blocks from which cluster-level failure rates are derived.

Reliability Assumptions

At fleet scale, component failures are not exceptional events but statistical certainties. **Reliability assumptions** here fix Mean Time To Failure (MTTF) for each major component class in a data-center GPU node and the time to detect and recover from failures. These values drive analysis in Chapter 7 and scheduling in Chapter 8; deeper failure modeling appears in Appendix E.

Component MTTF

Table G.5 lists the MTTF for each component class. These are steady-state values that exclude the "infant mortality" period (first 30–90 days) and the "wear-out" period (beyond rated lifetime). Sources include large-GPU research-cluster analysis Kokolis et al. (2025), TPUv4 supercomputer resiliency and operations Zu et al. (2024), and warehouse-scale machine design Barroso et al. (2019).

Table G.5: Component Mean Time To Failure: Steady-state MTTF anchors from (Kokolis et al. 2025; Zu et al. 2024; Barroso et al. 2019). Combined node MTTF ≈ 3592.8 h (149.7 d) at 8 per node.

Assumption	Value	Unit
GPU MTTF	50000	hours
NIC MTTF	150000	hours
PSU MTTF	100000	hours
PCIe switch MTTF	200000	hours
HBM MTTF	200000	hours
Top-of-rack switch MTTF	300000	hours
Optical cable and transceiver MTTF	50000	hours

Recovery time assumptions

Failure detection and recovery introduce downtime that compounds with cluster size. Table G.6 lists the assumptions used in checkpoint interval optimization (the Young-Daly formula in section 7.1.2) and effective throughput calculations throughout this book.

Table G.6: Recovery Time Parameters: Design assumptions for heartbeat, reschedule, and checkpoint-write bandwidth (Young–Daly context (Young 1974; Daly 2006)). Total recovery ≈ detect + reschedule + checkpoint size divided by write bandwidth; 70B FP32 example ≈ 92.8 s per failure.

Assumption	Value	Unit
Failure detection timeout (heartbeat)	30	seconds
Job reschedule delay	60	seconds
Checkpoint write bandwidth (aggregate)	100	GB/s

Reliability constants quantify how often failures occur and how long recovery takes. The *cost* of each failure, however, depends on how much communication must be repeated—which brings us to the parameters that model communication overhead.

Communication Model Parameters

This book models communication cost using the α-β framework: $T(n) = \alpha + n/\beta$, where α is startup latency and β is sustained bandwidth (Hockney 1994). Table G.7 lists values from fabric specifications, vendor product documentation, and large-cluster training studies (InfiniBand Trade Association 2000; Gangidi et al. 2024; NVIDIA 2026c; Ethernet Alliance 2025; Z. Jiang et al. 2024). They feed collective-communication models in Chapter 6 and parallelism comparisons in Chapter 5.

Table G.7: Communication Model Parameters (α-β): α (startup) and β (bandwidth) from fabric specifications and measured cluster studies (InfiniBand Trade Association 2000; Z. Jiang et al. 2024). The 10× NDR-vs-TCP gap explains why RDMA fabrics dominate synchronous training.

Assumption	Value	Unit
InfiniBand NDR startup latency (α)	5	us
InfiniBand NDR sustained bandwidth (β)	50	GB/s
InfiniBand HDR startup latency (α)	7	us
InfiniBand HDR sustained bandwidth (β)	25	GB/s
RoCE startup latency (α)	10	us
RoCE sustained bandwidth (β)	12.5	GB/s
TCP startup latency (α)	50	us

AllReduce cost model

The ring AllReduce—the most common collective for gradient synchronization—transmits $2(N_{\text{rank}} - 1)M/N_{\text{rank}}$ bytes per rank, where N_{rank} is the number of ranks and M is the message size. For large N_{rank}, this approaches $2M$ bytes per rank. Table G.8 lists the constants used in AllReduce cost estimation.

Table G.8: AllReduce Cost Parameters: Ring AllReduce factor of 2 from the standard ring algorithm (Gibiansky 2017). `Systems.Nodes.DGX_H100` (8 GPUs per node) sets the intra-node versus inter-node boundary in hierarchical collectives.

Assumption	Value	Unit
Ring AllReduce bandwidth factor	2	(dimensionless)
GPUs per node	8	GPUs/node

Communication models quantify the overhead of coordination. The sustainability cost of that coordination—the power drawn, the water consumed, the carbon emitted—requires separate assumptions for the physical infrastructure surrounding the compute.

Sustainability Assumptions

The environmental impact of fleet-scale ML depends on three factors: how much power the facility draws beyond the IT equipment (PUE), how much water the cooling system consumes (WUE), and how much carbon the local grid emits per kilowatt-hour. **Sustainability assumptions** here support Chapter 15 and total-cost-of-ownership estimates in Chapter 12.

Power usage effectiveness

Power Usage Effectiveness (PUE) is the ratio of total facility power to IT equipment power. A PUE of 1.0 would mean zero cooling overhead; real data centers range from 1.06 (liquid-cooled) to 1.58 (legacy air-cooled). Table G.9 lists the reference PUE values used throughout this book.

Table G.9: Power Usage Effectiveness (PUE): Illustrative PUE tiers from hyperscale surveys and Green Grid definitions (Uptime Institute 2022; The Green Grid 2007). Legacy vs. liquid-cooled gap: 49.1 percent more facility power for the same IT load.

Assumption	Value	Unit
PUE (liquid-cooled, best in class)	1.06	(ratio)
PUE (best air-cooled)	1.12	(ratio)
PUE (industry average)	1.4	(ratio)
PUE (legacy air-cooled)	1.58	(ratio)

Water usage effectiveness

Water Usage Effectiveness (WUE) measures liters of water consumed per kilowatt-hour of IT energy (The Green Grid 2011). Evaporative cooling towers achieve excellent PUE but consume significant water; closed-loop liquid cooling uses near-zero water but requires higher capital investment. Table G.10 lists the reference WUE values.

Table G.10: Water Usage Effectiveness (WUE): Illustrative WUE tiers for cooling-class comparisons (The Green Grid 2011; Uptime Institute 2022). Evaporative example at 10 MW IT load: 432,000 L/day.

Assumption	Value	Unit
WUE (air-cooled)	1.8	L/kWh
WUE (evaporative cooling)	1.8	L/kWh
WUE (closed-loop liquid)	0	L/kWh

Regional carbon intensity

The same training run emits vastly different amounts of CO_2 depending on the grid that supplies its electricity. Table G.11 lists regional grid carbon intensities from IEA and Canadian government emissions-factor data (International Energy Agency 2024b; Environment and Climate Change

Canada 2026), rounded for napkin math. These values are central to location-aware scheduling in Chapter 15.

Table G.11: Regional Grid Carbon Intensity: Grid carbon intensity in gCO_2/kWh, anchored to emissions-factor data (International Energy Agency 2024b; Environment and Climate Change Canada 2026) and ordered from low (Norway) to high (Poland). Location spans a 82× carbon spread—often larger than algorithmic tweaks.

Assumption	Value	Unit
Grid carbon intensity (Norway)	10	gCO2/kWh
Grid carbon intensity (Quebec)	20	gCO2/kWh
Grid carbon intensity (France)	50	gCO2/kWh
Grid carbon intensity (EU average)	270	gCO2/kWh
Grid carbon intensity (US average)	429	gCO2/kWh
Grid carbon intensity (Poland)	820	gCO2/kWh

Power density

AI accelerator racks draw 6–8× more power than traditional data center racks, creating thermal density challenges that force the transition from air cooling to liquid cooling. Table G.12 lists the reference rack power levels used in Chapter 2 and Chapter 15, anchored to current AI power-and-cooling product disclosures (Dell Technologies 2026; Schneider Electric 2024).

Table G.12: Rack Power Density: Illustrative rack-power tiers and air-cooling limit (Uptime Institute 2022; Dell Technologies 2026; Schneider Electric 2024). AI racks (70 kW–100 kW) exceed the 30 kW air-cooling practical limit.

Assumption	Value	Unit
Rack power (traditional enterprise)	12	kW
Rack power (AI, typical)	70	kW
Rack power (AI, high density)	100	kW
Air-cooling practical limit per rack	30	kW

Sustainability constants capture the physical and environmental costs of running a fleet. The economic constants in the next section translate those physical costs into dollar values, completing the total-cost-of-ownership picture.

Economic Assumptions

Fleet-scale cost estimates depend on GPU rental rates, electricity pricing, and data transfer charges. These are illustrative hyperscaler-order rates (2024–2025 magnitude) for ratio analysis, not quotes for a specific contract. Table G.13 supports Chapter 12 and Chapter 10 TCO napkin math.

Table G.13: Fleet-Scale Economic Parameters: Illustrative cloud GPU, electricity, and egress rates for TCO examples (substitute contract pricing for absolute budgets).

Assumption	Value	Unit
Cloud GPU training rental rate	4	dollar/h

Assumption	Value	Unit
Cloud GPU inference rental rate	2.5	dollar/h
Cloud electricity price	0.12	dollar/kWh
Cloud egress price per GB	0.09	dollar/GB

Economic constants set the price per unit of resource. The next question is: what fraction of each resource unit does useful work? Capacity planning constants quantify the gap between peak capability and achievable throughput.

Capacity Planning Assumptions

Capacity-planning assumptions quantify two related phenomena: (1) the fraction of peak hardware performance that real workloads achieve (Model FLOPs Utilization), and (2) how efficiently that performance scales as more machines are added (scaling efficiency). Together they determine the *effective* compute available for a given cluster size and budget—the number that matters for project planning.

The planning equation is multiplicative:

$$R_{\text{eff}} = N R_{\text{peak}} \times \text{MFU} \times \eta_{\text{scaling}} \times (1 - f_{\text{overhead}})$$

Here R_{eff} is effective fleet throughput, $N R_{\text{peak}}$ is nominal cluster peak throughput, MFU discounts peak hardware to realized model throughput, η_{scaling} discounts added machines for communication and coordination losses, and f_{overhead} removes wall time spent on pipeline bubbles, checkpoints, failure recovery, and maintenance. The three subsections that follow fix those factors in order.

Model FLOPs utilization

Model FLOPs Utilization (MFU) measures the ratio of actual compute throughput to the hardware's peak theoretical throughput. An MFU of 0.50 means the workload achieves half the peak FLOP/s. Table G.14 lists the reference MFU ranges used in training time and cost estimates throughout this book.

Table G.14: Model FLOPs Utilization (MFU) Ranges: Training MFU 0.3–0.5 from large-model training reports (Chowdhery et al. 2022; Narayanan et al. 2021); inference batch-1 MFU 5 percent from (Pope et al. 2023) (memory-bound decode).

Assumption	Value	Unit
Training MFU (low end)	0.3	(fraction)
Training MFU (high end)	0.5	(fraction)
Inference MFU (batch size 1)	0.05	(fraction)
Inference MFU (batched)	0.4	(fraction)

Scaling efficiency

Scaling efficiency $\eta_{\text{scaling}} = T_1/(N \times T_N)$ measures how much of the added compute actually reduces training time. Table G.15 lists **illustrative efficiency tiers** aligned with published LLM training at scale (Chowdhery et al. 2022; Z. Jiang et al. 2024)—workload and fabric dependent.

Table G.15: Scaling Efficiency by Cluster Size: Illustrative η_{scaling} tiers from (Chowdhery et al. 2022; Z. Jiang et al. 2024). At 8,192 GPUs, 2,867.2 effective GPUs of 0.35 nominal.

Assumption	Value	Unit
Scaling efficiency (32 GPUs)	0.9	(fraction)
Scaling efficiency (256 GPUs)	0.7	(fraction)
Scaling efficiency (1,024 GPUs)	0.5	(fraction)
Scaling efficiency (8,192 GPUs)	0.35	(fraction)

Overhead budgets

Real training jobs spend a fraction of wall time on noncompute activities: pipeline bubble idle time, checkpoint writes, failure recovery, and scheduled maintenance. Table G.16 lists the overhead budgets assumed throughout this book. These fractions are additive: total overhead at fleet scale is approximately 5 percent + 3 percent + 10 percent + 5 percent = 23 percent, meaning only 77 percent of wall time produces useful training progress.

Table G.16: Overhead Budgets (Fraction of Wall Time): Book-convention wall-time fractions for pipeline bubbles, checkpointing, failure recovery, and maintenance—engineering targets, not physical constants. Combined 23 percent overhead ⇒ 23.1 days of effective work in a 30-day run.

Assumption	Value	Unit
Wall-time overhead (pipeline bubbles)	0.05	(fraction)
Wall-time overhead (checkpointing)	0.03	(fraction)
Wall-time overhead (failure recovery)	0.1	(fraction)
Wall-time overhead (scheduled maintenance)	0.05	(fraction)

The tables above use the unit conventions in table G.17—decimal data prefixes (KB $= 10^3$ bytes), separate work (FLOPs) from throughput (FLOP/s), and the aliases below.

Unit Conventions

Table G.17 fixes the unit conventions used in every quantitative example in this volume. Each row gives the multiplier k in `1 alias` $= k$ `base`. Data prefixes use decimal SI (KB $= 10^3$ bytes, not 1024). Throughput quantities (FLOP/s, GB/s) combine these aliases with time; adding incompatible dimensions (bytes to FLOP/s) is a category error in napkin math, not a unit conversion.

Table G.17: Unit conventions: Decimal SI aliases and scale factors (book convention). Throughput forms (FLOP/s, GB/s) divide work or data by time using the same conventions.

Alias	Multiplier	Base unit
`byte`	1	byte
`KB`	1000	byte
`MB`	1e+06	byte
`GB`	1e+09	byte
`TB`	1e+12	byte
`PB`	1e+15	byte

Alias	Multiplier	Base unit
`flop`	1	flop
`GFLOPs`	1e+09	flop
`TFLOPs`	1e+12	flop
`ZFLOPs`	1e+21	flop
`param`	1	param
`Mparam`	1e+06	param
`Gbps`	1e+09	bit/s
`NS`	1e-09	second
`US`	1e-06	second
`MS`	0.001	second
`second`	1	second
`hour`	3600	second
`day`	86400	second
`joule`	1	joule
`watt`	1	watt
`meter`	1	meter

Two fleet-scale conventions deserve emphasis beyond the table:

- **Network bandwidth** appears in both bit-rate (Gbps) and byte-rate (GB/s). Divide by 8 to convert Gbps to GB/s (for example, 400 Gbps = 50 GB/s for InfiniBand NDR). Chapters use whichever form fits the calculation; byte-rate is usually easier for transfer-time estimates.
- **Carbon intensity** uses gCO_2/kWh (grams of CO_2 per kilowatt-hour). Multiply total energy (kWh) by regional carbon intensity to get emissions in grams, then divide by 10^6 for tonnes.

G.1 Assumption Provenance

The catalog below maps each appendix section to source class and primary references. The book cites with `@citekey`; `mlsysim` mirrors the same provenance as `Provenance` records on `Sourced` registry scalars and `metadata` (e.g. `Systems.Reliability.Gpu.mttf_hours.provenance`, `Hardware.Cloud.H100.metadata.provenance`) without shipping a `.bib` file. See table G.18 for the full table.

Table G.18: Fleet assumption provenance catalog: Quick map from appendix section to source class and bibliography.

Appendix section	Source type	Primary references
H100 recap	Vendor datasheet peak	(Choquette 2023)
Cluster tiers	Book convention	(Kokolis et al. 2025) (scale context)
Network fabrics	Vendor specs; InfiniBand standard	(NVIDIA Corporation 2017, 2020; Choquette 2023; InfiniBand Trade Association 2000)
Reliability (MTTF)	Fleet studies	(Kokolis et al. 2025; Zu et al. 2024; Barroso et al. 2019)
Recovery times	Design assumptions	(Young 1974; Daly 2006)
Communication (α–β)	Fabric specs; training at scale	(InfiniBand Trade Association 2000; Z. Jiang et al. 2024)
AllReduce model	Algorithm identity	(Gibiansky 2017)
Sustainability (PUE/WUE/rack power)	Industry surveys	(Uptime Institute 2022; The Green Grid 2007)
Carbon intensity	Grid statistics	(International Energy Agency 2023)
Cloud economics	Illustrative rates	Editorial (2024–2025 order of magnitude)

Appendix section	Source type	Primary references
MFU and scaling efficiency	Published LLM training	(Chowdhery et al. 2022; Narayanan et al. 2021; Pope et al. 2023; Z. Jiang et al. 2024)
Overhead budgets	Book convention	Editorial engineering targets
Unit conventions	Decimal SI; book notation	Editorial

Summary

This appendix is the book's shared fleet-scale assumption sheet: every napkin-math estimate in the volume should be traceable to a row in the tables above—and, where it matters, to section G.1 and table G.18.

Key Takeaways: Using the fleet assumption tables

- **One place for shared fleet numbers**: Cluster tiers, fabric bandwidths, MTTF and recovery times, α–β communication parameters, facility efficiency, water use, carbon intensity, cloud pricing, MFU, scaling efficiency, and overhead budgets used in worked examples are listed here so you can audit or replace them without re-deriving from prose.
- **Reliability assumptions set the failure clock**: MTTF and recovery-time rows determine how often failures interrupt training and how much wall time is lost. At 10,000+ GPUs, failure is steady state, not an exception; see Appendix E for deeper modeling.
- **Communication assumptions determine viability**: α–β values determine whether synchronous distributed training is viable for a given model size and fabric. The 10× latency gap between RDMA and TCP explains why training clusters favor InfiniBand.
- **Sustainability assumptions expose location leverage**: Facility efficiency, water use, and carbon intensity show that data-center location and cooling can outweigh many algorithmic tweaks. The 82× spread between Norway and Poland grid carbon intensity is often the largest lever.
- **Capacity assumptions quantify usable throughput**: MFU, scaling efficiency, and overhead budgets bridge peak hardware to useful throughput. At 8,192 GPUs, only about 35 percent of ideal scaling is realized, and combined overheads consume roughly 23 percent of wall time.
- **Substitute your own assumptions**: When your deployment differs—different fabric, region, MFU, or cluster size—swap the `Value` column and rerun the same formulas the chapters use.
- **Check provenance before debating a number**: Datasheet peaks, fleet studies, illustrative rates, and book conventions are sourced differently; table G.18 and section captions say which is which.

Glossary

This glossary defines key terms used throughout this book. Terms are organized alphabetically.

A

accountability The mechanisms by which individuals or organizations are held responsible for the outcomes of AI systems, involving traceability, documentation, auditing, and the ability to remedy harms.

activation checkpointing A memory-saving training technique that stores only selected activations and recomputes others during backpropagation, trading extra compute for lower memory use.

adapter modules Small trainable neural network components inserted between frozen layers of a pretrained model to enable lightweight adaptation without modifying the base architecture.

adaptive resource pattern A design pattern that enables systems to dynamically adjust their operations in response to varying resource availability, ensuring efficiency and resilience by scaling up or down based on computational load, network bandwidth, and storage capacity.

adversarial attack A type of attack where carefully crafted inputs are designed to cause machine learning models to make incorrect predictions while remaining nearly indistinguishable from legitimate data to humans

adversarial example A maliciously modified input that is designed to fool a machine learning model into making an incorrect prediction, often created by adding small, imperceptible perturbations to legitimate data

adversarial training A defense technique that involves training models on adversarial examples to improve their robustness and ability to correctly classify adversarial inputs

agi Artificial General Intelligence - computational systems that match or exceed human cognitive capabilities across all domains of knowledge and reasoning, capable of generalizing across diverse problem domains without task-specific training.

ai for good The design, development, and deployment of machine learning systems aimed at addressing important societal and environmental challenges to enhance human welfare, promote sustainability, and contribute to global development goals.

algorithmic fairness The principle that automated systems should not disproportionately disadvantage individuals or groups based on protected attributes such as race, gender, or age.

AllGather A collective communication operation that gathers each rank's shard and makes the combined result available to every rank.

AllReduce A collective communication operation that reduces values across ranks and returns the reduced result to every rank, commonly used for gradient synchronization.

AllToAll A collective communication operation where each rank sends distinct data to every other rank and receives distinct data from each rank.

alpha-beta model A communication cost model that separates fixed startup latency from message-size-dependent transfer time to estimate the cost of moving data.

anomaly detection The identification of patterns in data that do not conform to expected behavior, often used to detect outliers, faults, or malicious activities in systems.

anonymization The process of removing or modifying personally identifiable information from datasets to protect individual privacy, though often insufficient against sophisticated re-identification attacks.

arithmetic intensity The amount of computation performed per byte of data moved, usually measured in FLOP/byte.

artificial general intelligence A hypothetical form of AI that matches or exceeds human cognitive abilities across all domains, representing the ultimate goal of AI research beyond current narrow AI systems.

artificial intelligence A broad field of computer science focused on creating systems that can perform tasks typically requiring human intelligence, including learning, reasoning, and decision-making

attack taxonomy Systematic classification of cybersecurity threats and adversarial attacks against ML systems, organizing threats by method, target, and impact to guide defense strategies.

autoencoder A neural network architecture that learns compressed data representations by minimizing reconstruction error, commonly used for anomaly detection and dimensionality reduction.

automation bias The tendency for humans to over-rely on automated system outputs even when clear errors are present, potentially compromising human oversight.

automl Automated machine learning that uses algorithms to automate the process of applying machine learning to real-world problems, including feature engineering, model selection, and hyperparameter tuning.

availability attack A type of data poisoning attack that aims to degrade the overall performance of a machine learning model by introducing noise or corrupting training data across multiple classes.

B

backdoor attack A type of data poisoning where hidden triggers are embedded in training data, causing models to behave maliciously when specific patterns are encountered during inference

backfill scheduling A scheduling policy that runs smaller jobs in idle gaps when doing so will not delay a reserved larger job.

backpropagation The algorithm for computing gradients in neural networks by propagating error signals backward through layers, essential for training but computationally expensive on resource-constrained devices

bayesian neural networks Neural networks that incorporate probability distributions over their weights, enabling uncertainty quantification in predictions and more robust decision-making.

bias detection Systematic methods for identifying unfair discrimination or disparate treatment across different demographic groups in machine learning system outputs.

bias mitigation Techniques and interventions designed to reduce unfair discrimination in machine learning systems, applied during data collection, model training, or postprocessing stages.

bias-only adaptation A lightweight training strategy that freezes all model weights and updates only scalar bias terms, drastically reducing memory requirements and computational overhead for on-device learning.

bin packing The assignment of jobs with resource demands into fixed-capacity nodes or clusters; in ML fleets this often spans GPUs, memory, CPUs, network locality, and topology.

biodiversity monitoring The systematic observation and measurement of biological diversity using technology such as camera traps and sensor networks to track species populations, habitat

changes, and conservation effectiveness.

bisection bandwidth The minimum aggregate link capacity crossing any cut that splits a network into two equal halves, bounding worst-case throughput for global communication.

bit flip A hardware fault where a single bit in memory or a register unexpectedly changes its value from 0 to 1 or vice versa, potentially corrupting data or computations.

black-box attack An adversarial attack where the attacker has no knowledge of the model's internal architecture, parameters, or training data, and must rely solely on querying the model and observing outputs.

brain-computer interface A direct communication pathway between the brain and an external device, enabling control of computers or prosthetics through neural signals and representing a convergence of ML with neurotechnology.

built-in self-test (bist) Hardware testing mechanisms that allow components to test themselves for faults using dedicated circuitry and predefined test patterns.

bulk synchronous parallel (BSP) A parallel execution model in which workers perform local computation, communicate, and then wait at a global barrier before the next step.

C

cache timing attack A type of side-channel attack that exploits variations in memory cache access patterns to infer sensitive information about program execution or data.

canary deployment A rollout pattern that sends a small slice of traffic to a new model or service version, monitors behavior, and then expands or rolls back.

CAP theorem A distributed-systems result stating that, under a network partition, a system must choose between strict consistency and availability.

carbon-aware scheduling A computational approach that schedules AI workloads based on the carbon intensity of the electricity grid, prioritizing execution when renewable energy sources are most available.

carbon footprint The total amount of greenhouse gas emissions produced directly and indirectly by an individual, organization, event, or product, typically measured in CO_2-equivalent.

carbon intensity The greenhouse gas emissions associated with a unit of electricity, commonly measured in grams of CO_2-equivalent per kilowatt-hour.

carbon usage effectiveness (CUE) A data-center sustainability metric that relates carbon emissions to IT energy use, analogous to power usage effectiveness for facility energy overhead.

catastrophic forgetting The phenomenon where neural networks lose previously learned knowledge when adapting to new tasks, a critical challenge in continual on-device learning scenarios.

certified robustness A formal guarantee that a model prediction remains unchanged for all allowed perturbations within a specified radius or norm bound.

checkpoint and restart mechanisms Techniques that periodically save a program's state so it can resume from the last saved state after a failure, improving system resilience.

chunked prefill An LLM serving technique that splits long prompt processing into smaller chunks so prefill work can interleave with decode work.

client scheduling The process of selecting which devices participate in federated learning rounds based on availability, data quality, and resource constraints to ensure representative model updates.

cloud ml Machine learning systems that use cloud computing infrastructure to provide scalable computational resources for training and inference, typically offering high-bandwidth connectivity and substantial processing power.

collective operation A group communication pattern in which all ranks participate to aggregate, broadcast, or redistribute data.

combinational logic Digital logic circuits where the output depends only on the current input states, not any past states or memory elements.

compound ai systems AI architectures that combine multiple specialized models, tools, and components to achieve complex capabilities through systematic integration rather than relying on a single monolithic model.

Compute Express Link (CXL) A cache-coherent interconnect standard that lets CPUs, accelerators, and memory devices share or pool memory over PCIe-derived links.

concept bottleneck models Neural network architectures that first predict interpretable intermediate concepts before making final predictions, combining deep learning power with transparency.

concept drift A change in the relationship between input features and target outputs over time, requiring model adaptation to maintain performance.

conservation technology Technological solutions designed to protect and monitor wildlife and ecosystems, including camera traps, sensor networks, and satellite monitoring systems for tracking animal behavior and detecting threats.

consistent hashing A hash-based routing method that keeps keys or sessions mapped to stable replicas while minimizing remapping when replicas are added or removed.

constitutional ai A training method where models learn to improve their own outputs by critiquing responses against a set of principles, enabling iterative self-refinement and reducing harmful content while maintaining helpfulness

continual learning The ability of machine learning systems to learn continuously from a stream of data while retaining previously acquired knowledge, addressing the challenge of catastrophic forgetting in neural networks

continuous batching An LLM serving scheduler that removes completed sequences and admits new requests at iteration boundaries rather than waiting for an entire batch to finish.

cooling effectiveness The efficiency with which a data center cooling system removes heat from computing equipment, typically measured as the ratio of heat removed to energy consumed for cooling.

counterfactual explanations Explanations that describe how a model's output would change if specific input features were modified, particularly useful for understanding decision boundaries.

covariate shift A type of distribution shift where the input distribution changes while the conditional relationship between inputs and outputs remains stable.

D

data center A facility housing computer systems and associated components such as telecommunications and storage systems, typically including redundant power supplies, cooling systems, and network connections.

data compression Techniques for reducing the size and complexity of training data through encoding, quantization, or feature extraction to enable efficient storage and processing on memory-constrained devices.

data parallelism A training strategy where each worker holds a full model replica, processes a different data shard, and synchronizes gradients before applying the same update.

data poisoning An attack method where adversaries inject carefully crafted malicious data points into the training dataset to manipulate model behavior in targeted or systematic ways

data sanitization The process of deliberately and permanently removing or destroying data stored on memory devices to make it unrecoverable, ensuring data security.

decode phase The autoregressive generation phase in LLM inference that emits tokens one step at a time using the KV cache, often limited by memory bandwidth.

deep learning A subset of machine learning using neural networks with multiple hidden layers to automatically learn hierarchical representations from data.

defensive distillation A technique that trains a student model to mimic a teacher model's behavior using soft labels, reducing sensitivity to adversarial perturbations.

demographic parity A fairness criterion requiring that the probability of receiving a positive prediction is independent of group membership across protected attributes.

dennard scaling The historical observation that as transistors become smaller, their power density remains approximately constant, allowing for more transistors without proportional increases in power consumption.

depthwise separable convolutions A computational technique that decomposes standard convolutions into depthwise and pointwise operations, reducing parameters and computation by roughly 8–9× for typical 3×3 high-channel mobile layers.

differential privacy A mathematical framework that provides formal pri-

vacy guarantees by adding calibrated noise to computations, ensuring that the inclusion or exclusion of any individual's data has a provably limited effect on the output

differentially private stochastic gradient descent (DP-SGD) A training algorithm that clips per-example gradients and adds calibrated noise to provide differential privacy guarantees.

digital divide The gap between those who have access to modern information and communication technology and those who do not, particularly affecting underserved communities' ability to benefit from digital solutions.

digital twin A virtual representation of a physical system that uses real-time data and machine learning to mirror, predict, and optimize the behavior of its physical counterpart.

disaggregated serving A serving architecture that separates prefill and decode work onto different pools or hardware optimized for their distinct bottlenecks.

disaster response systems Automated systems that use machine learning to detect, predict, and respond to natural disasters through satellite imagery analysis, sensor networks, and resource allocation optimization.

distributed checkpointing A checkpointing pattern where workers write their own portions of training state in parallel under a coordinated protocol.

distributed knowledge pattern A design pattern that addresses collective learning and inference across decentralized nodes, emphasizing peer-to-peer knowledge sharing and collaborative model improvement while maintaining operational independence.

distributed tracing An observability technique that follows a request across multiple services or components using propagated trace context.

distributed training The practice of training machine learning models across multiple computing nodes or devices to reduce training time and enable larger model architectures.

distribution shift The phenomenon where data encountered during model deployment differs from the training distribution, potentially degrading model performance

distribution shift types Formal categorization of changes in data distributions including covariate shift, label shift, concept drift, and domain shift, each requiring specific adaptation techniques.

domain adaptation Machine learning techniques that enable models trained on one domain to perform well on a different but related domain, addressing distribution mismatch challenges.

domain-specific ai applications Machine learning solutions tailored to specific sectors like healthcare, agriculture, education, or disaster response, designed to address unique challenges and constraints.

double modular redundancy (dmr) A fault-tolerance technique where computations are duplicated across two independent systems to identify and correct errors through comparison.

dual-use dilemma The challenge of mitigating misuse of technology that has both positive and negative potential applications, particularly relevant in AI security.

dynamic batching A serving technique that briefly aggregates incoming requests into batches to improve throughput while controlling added latency.

E

edge ai The deployment of artificial intelligence algorithms directly on edge devices like smartphones, IoT sensors, and embedded systems, enabling real-time processing without cloud connectivity.

edge computing A distributed computing paradigm that brings computation and data storage closer to data sources, enabling real-time processing with reduced latency and lower bandwidth requirements

edge ml Machine learning systems that perform inference and sometimes training at the edge of networks, typically on resource-constrained devices like smartphones or embedded systems with limited computational power.

edge training The process of training or fine-tuning machine learning models directly on edge devices, enabling personalization and adaptation without requiring data transmission to cloud servers.

elastic training Training that can continue as workers join, leave, fail, or are replaced by resizing process groups and redistributing state or data.

electromigration The movement of metal atoms in a conductor under the influence of an electric field, potentially causing permanent hardware faults over time.

embodied carbon The total greenhouse gas emissions generated during the manufacturing, transportation, and installation of a product before it begins operation.

emergent capabilities Abilities that appear suddenly in neural networks at specific parameter thresholds, such as reasoning and arithmetic skills that emerge discontinuously rather than gradually improving with scale.

energy efficiency The ratio of useful output to energy input, measuring how effectively a system converts energy into desired computational work.

ensemble methods Machine learning approaches that combine predictions from multiple models to improve accuracy, robustness, and reliability compared to individual models.

environmental impact measurement Systematic tracking and quantification of the ecological effects of AI systems, including energy consumption, carbon emissions, and resource depletion across the complete system lifecycle.

environmental monitoring The systematic collection and analysis of environmental data using sensor networks and machine learning to track ecosystem health, pollution levels, and climate change impacts.

equality of opportunity A fairness criterion focused on ensuring equal true positive rates across groups, guaranteeing that qualified individuals are treated equally regardless of group membership.

equalized odds A fairness definition requiring that true positive and false positive rates are equal across different demographic groups.

error-correcting codes Methods used in data storage and transmission to detect and correct errors, improving system reliability and data integrity.

error feedback A gradient-compression technique that stores compression residuals and reinjects them into later updates to reduce convergence error.

esp32 A low-cost microcontroller unit widely used in IoT applications, featuring a 240 MHz processor and 520 KB SRAM, commonly deployed in resource-constrained social impact applications.

exact model theft An attack that aims to extract the precise internal structure, parameters, and architecture of a machine learning model, allowing complete reproduction of the original model.

experience replay A memory-based technique that stores past training examples in a buffer to prevent catastrophic forgetting and stabilize learning in streaming or continual adaptation scenarios.

expert collapse A training pathology in mixture of experts models where only a few experts receive significant training signal, causing other experts to become underutilized and reducing the model's effective capacity.

expert parallelism A strategy for mixture of experts models in which experts are distributed across workers and tokens are routed to selected experts.

explainability The ability of stakeholders to understand how a machine learning model produces its outputs through post-hoc explanation techniques.

explainable ai AI systems designed to provide clear, interpretable explanations for their decisions and predictions, addressing the "black box" problem of complex machine learning models.

external memory Mechanisms that allow neural networks to access and manipulate external storage systems, extending their working memory beyond parameter storage to enable more complex reasoning and information retrieval.

F

f1 score A measure of model accuracy that combines precision and recall into a single metric, calculated as their harmonic mean.

fairness constraints Technical and policy restrictions designed to ensure equitable treatment across demographic groups in machine learning systems.

fast gradient sign method (fgsm) A gradient-based adversarial attack that generates adversarial examples by adding small perturbations in the direction of the gradient.

fat-tree topology A Clos-style hierarchical network topology whose aggregate capacity increases toward the spine to provide high bisection bandwidth and multipath routing.

fault injection attack A physical attack that deliberately disrupts hardware operations through techniques like volt-

age manipulation or electromagnetic interference to induce computational errors and compromise system integrity.

fault tolerance The ability of a system to continue operating correctly even when some of its components fail or encounter errors.

feature store A system for defining, computing, storing, and serving reusable machine learning features consistently across training and inference.

federated averaging The standard algorithm for federated learning where client model updates are aggregated using weighted averaging based on local dataset sizes to produce a global model.

federated learning A machine learning approach that trains algorithms across decentralized data sources without requiring data to be centralized, improving privacy and reducing data transmission energy costs

few-shot learning A machine learning paradigm that enables models to adapt to new tasks using only a small number of labeled examples, critical for data-sparse on-device scenarios.

FlashAttention A tiled attention algorithm that avoids materializing the full attention matrix, using online softmax and memory-aware blocking to reduce memory traffic.

flop/s Floating-point operations per second, a measure of computer performance indicating how many mathematical calculations involving decimal numbers a system can perform per second.

foundation models Large-scale pretrained models like GPT and BERT that can be adapted for a wide variety of downstream tasks, serving as a foundation for multiple applications.

Fully Sharded Data Parallel (FSDP) A sharded data-parallel approach that partitions model state across workers and gathers parameters only when needed for computation.

G

gang scheduling A scheduling policy that allocates all workers or resources for a distributed job together so the job can start and make progress as a unit.

gdpr The General Data Protection Regulation, a European Union law that imposes strict requirements on personal data processing and significantly influences privacy-preserving machine learning design

generative adversarial networks A class of machine learning systems where two neural networks compete against each other, with one generating fake data and the other trying to detect it, leading to highly realistic synthetic data generation.

glitches Momentary deviations in voltage, current, or signal that can cause incorrect operation in digital systems and circuits.

governance frameworks Structured approaches for managing responsible AI development including policies, procedures, oversight mechanisms, and accountability structures.

gpu Graphics Processing Unit, a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images and parallel processing tasks like AI training.

GPUDirect RDMA A direct data path that lets RDMA-capable network adapters transfer data to or from GPU memory without staging through host memory.

GPUDirect Storage (GDS) A storage data path that enables direct DMA between storage devices and GPU memory, bypassing CPU and host DRAM bottlenecks.

gradient accumulation A technique that accumulates gradients over multiple microbatches before an optimizer step or synchronization.

gradient descent An optimization algorithm that iteratively updates model parameters by moving in the direction opposite to the gradient of the loss function, fundamental to neural network training.

gradient inversion attack An attack that reconstructs sensitive training examples or features from shared gradients or model updates.

gradient staleness The delay, measured in updates or steps, between when a gradient is computed and when it is applied.

green ai metrics Specialized performance indicators that measure the environmental impact of AI systems, including carbon footprint, energy efficiency, and resource utilization throughout the ML lifecycle.

green computing The practice of designing, manufacturing, using, and disposing of computers and computer systems in an environmentally responsible manner.

grey-box attack An adversarial attack where the attacker has partial knowledge about the model, such as knowing the architecture but not the specific parameters or training data.

H

hallucination A generative model failure where output is fluent and confident but unsupported, fabricated, or factually wrong.

hardware constraint optimization Techniques for adapting ML algorithms and models to work within the memory, compute, and power limitations of mobile and embedded devices.

hardware redundancy The duplication of critical hardware components to provide backup functionality and improve system reliability through voting mechanisms.

hardware trojan A malicious modification embedded in hardware components during manufacturing that can remain dormant under normal conditions but trigger harmful behavior when specific conditions are met.

heartbeat mechanisms Periodic signals sent between system components to monitor health and detect failures, enabling timely fault detection and recovery.

hierarchical AllReduce An AllReduce decomposition that reduces within fast local domains before communicating across slower inter-node or rack links.

hierarchical processing pattern A design pattern that organizes systems into tiers (edge, regional, cloud) that share responsibilities based on available resources and capabilities, optimizing resource usage across the computing spectrum.

high bandwidth memory (HBM) A 3D-stacked DRAM architecture connected through very wide, short interconnects to provide high memory bandwidth near accelerators.

homomorphic encryption A cryptographic technique that allows computations to be performed directly on encrypted data without decrypting it first, enabling privacy-preserving machine learning inference.

hot spares Backup components kept ready to instantaneously replace failing components without disrupting system operation, providing redundancy.

huber loss A robust loss function used in regression that is less sensitive to outliers compared to squared error loss, improving training stability.

human-ai collaboration The synergistic partnership between humans and AI systems where each contributes their unique strengths to solve complex problems more effectively than either could alone.

human oversight The principle that human judgment should supervise, correct, or halt automated decisions, maintaining meaningful human control over AI systems.

hyperparameter optimization The process of finding the optimal configuration of hyperparameters (learning rate, batch size, network architecture parameters) that control the machine learning training process.

I

impact assessment frameworks Structured methodologies for evaluating the potential social, economic, and environmental effects of AI deployments in humanitarian and development contexts.

inference The phase in machine learning where a trained model is used to make predictions or generate outputs on new, previously unseen data.

InfiniBand A high-performance, low-latency interconnect widely used for HPC and ML training clusters, with native RDMA and credit-based flow control.

intermittent faults Hardware faults that occur sporadically and unpredictably, appearing and disappearing without consistent patterns, making diagnosis challenging.

interpretability The degree to which humans can understand the reasoning behind a machine learning model's predictions, often referring to inherently transparent models.

iot sensors Internet of Things devices that collect and transmit environmental or behavioral data, often operating on limited power budgets and using low-bandwidth communication protocols.

K

k-anonymity A privacy guarantee where each released record is indistinguishable from at least $k-1$ others af-

ter generalizing or suppressing quasi-identifiers.

knowledge distillation A technique where a large, complex "teacher" model transfers its learned knowledge to a smaller, more efficient "student" model, maintaining performance while reducing computational requirements

KV cache A memory buffer that stores previously computed transformer key and value tensors so autoregressive generation does not recompute attention history for every token.

L

label shift A type of distribution shift where the distribution of target labels changes while the conditional relationship between features and labels remains constant.

large language models Neural networks with billions or trillions of parameters trained on vast text corpora, capable of understanding and generating human-like text across diverse domains and tasks.

lifecycle assessment A systematic approach to evaluating the environmental impacts of a product or system throughout its entire life cycle, from raw material extraction to disposal.

load balancing Techniques in mixture of experts models to ensure that computational load and training signal are distributed evenly across experts, preventing expert collapse and maintaining model efficiency.

LogP model A communication model that represents latency, overhead, gap, and processor count to reason about message passing and overlap.

lookup table A data structure that replaces runtime computation with simpler array indexing operations, commonly used for performance optimization.

lora technology Long Range wireless communication protocol that enables IoT devices to communicate over 15 km with minimal power consumption, ideal for agricultural and environmental monitoring applications.

low-rank adaptation A parameter-efficient fine-tuning method that approximates weight updates using low-rank matrices, reducing trainable parameters while maintaining adaptation capability.

M

machine consciousness The hypothetical emergence of conscious awareness in artificial systems, representing a frontier research area exploring whether machines can develop subjective experiences.

machine learning A subset of artificial intelligence that enables computers to learn and make decisions from data without being explicitly programmed for each specific task.

machine learning lifecycle The complete process of developing, deploying, and maintaining ML systems, from data collection through model retirement.

machine learning operations (mlops) The practice of deploying and maintaining machine learning models in production reliably and efficiently through automated pipelines.

machine learning security The protection of data, models, and infrastructure from unauthorized access, manipulation, or disruption throughout the entire machine learning lifecycle.

machine unlearning Techniques for removing the influence of specific data points from trained models without complete retraining, supporting data deletion rights.

marginal emissions The emissions caused by an incremental increase in electricity demand, often from the generator dispatched next on the grid.

mean time between failures (MTBF) The expected operating time between successive failures in a repairable component or system.

megawatt-hour A unit of energy equal to one megawatt of power used for one hour, commonly used to measure electricity consumption in large facilities like data centers.

membership inference attack A privacy attack that attempts to determine whether a specific data point was included in a model's training dataset by analyzing the model's behavior and outputs.

meta-learning The process of learning how to learn, where models are trained to quickly adapt to new tasks with minimal data, particularly useful for personalization in on-device systems

microcontroller A compact integrated circuit designed to govern specific operations in embedded systems, typically featuring limited processing power and memory but optimized for low power consumption and real-time applications.

minimax A decision-making strategy used in game theory that attempts to minimize the maximum possible loss in adversarial scenarios.

mixed precision training A technique that uses different numerical precisions for different parts of neural network training, typically combining 16-bit and 32-bit floating-point arithmetic to reduce memory usage and increase training speed.

mixture of experts An architectural approach that uses multiple specialized sub-models (experts) with a gating mechanism to route inputs to the most relevant experts, enabling efficient scaling while maintaining sparsity.

mobile ml Machine learning systems optimized for mobile devices like smartphones and tablets, balancing computational efficiency with inference accuracy for on-device processing.

mobile-optimized architectures Neural network designs specifically created for mobile deployment, emphasizing parameter efficiency, computational speed, and energy conservation.

mobilenet A family of efficient neural network architectures designed for mobile devices using depthwise separable convolutions to achieve significant reductions in model size and computation.

mode collapse A failure mode in generative models where the model produces only a limited variety of outputs, ignoring the diversity present in the training data and failing to capture the full distribution.

model cards Documentation framework that provides structured information about machine learning models, including intended use, performance characteristics, and limitations.

model compression Techniques to reduce the size and computational requirements of machine learning models through methods like quantization, pruning, and knowledge distillation to enable deployment on resource-constrained devices

model extraction The process of stealing or recreating a machine learning model by observing its input-output behavior, often through systematic querying of model APIs.

model FLOPs utilization (MFU) The fraction of hardware peak FLOP/s spent on useful model computation over wall-clock time, used to measure sustained training or serving efficiency.

model inversion attack An attack that attempts to reconstruct training data or infer sensitive information about the dataset by analyzing a model's outputs and confidence scores.

model pruning The process of removing unnecessary weights, neurons, or connections from a trained neural network to reduce its size and computational requirements.

model quantization A technique that reduces the precision of model parameters (typically from 32-bit to 8-bit or lower) to decrease model size and computational requirements while maintaining acceptable accuracy.

model registry A central catalog for model artifacts, versions, metadata, lineage, dependencies, and lifecycle state.

model uncertainty The inadequacy of a machine learning model to capture the full complexity of the underlying data-generating process, leading to prediction uncertainty.

model watermarking A technique for embedding verifiable ownership signatures into machine learning models that can be used to detect unauthorized use or prove intellectual property theft.

monte carlo dropout A technique that uses multiple forward passes with different dropout masks at inference time to estimate prediction uncertainty.

Moore's Law The observation that the number of transistors on a microchip doubles approximately every two years, reducing cost per transistor under historical scaling assumptions.

multi-agent approach Systems architecture where multiple AI agents collaborate, negotiate, or compete to solve complex problems, enabling division of labor and specialized expertise across different components.

Multi-Instance GPU (MIG) An NVIDIA mechanism for partitioning one GPU into isolated hardware slices with separate compute and memory resources.

multicalibration A fairness technique ensuring that model predictions remain calibrated across intersecting subgroups, addressing complex demographic interactions.

multimodal ai AI systems that can process and understand multiple types of data simultaneously, such as text, images, audio, and video, enabling more com-

prehensive understanding and interaction.

N

narrow ai AI systems designed to excel at specific, well-defined tasks but lacking the ability to generalize across diverse problem domains, in contrast to artificial general intelligence.

NCCL NVIDIA Collective Communications Library, a GPU communication library for topology-aware collective operations.

neural architecture search Automated methods for designing optimal neural network architectures, using algorithms to explore the space of possible network designs and find the best performing structures

neural engine Specialized hardware accelerators designed for machine learning inference and training, such as Apple's Neural Engine or Google's Edge TPU, optimized for on-device AI workloads.

neural network A computational model inspired by biological neural networks, consisting of interconnected nodes (neurons) organized in layers that can learn complex patterns from data.

neural processing unit (NPU) A specialized accelerator for neural network workloads, often optimized for low-power matrix and tensor operations on edge devices.

neuromorphic computing Computing architectures inspired by the structure and function of biological neural networks, designed to process information more efficiently than traditional digital computers.

non-iid data Non-independent and identically distributed data where samples are not uniformly distributed across devices or time, creating challenges for federated learning convergence and generalization.

Non-Volatile Memory Express (NVMe) A high-performance protocol for SSDs attached over PCIe, often used as a local cache or checkpoint staging tier in ML storage hierarchies.

NVLink A high-bandwidth NVIDIA interconnect for GPU-to-GPU communication within a node.

O

object storage A storage model that manages data as objects with metadata and network API access, commonly used for durable lower-cost capacity.

on-device learning The local adaptation or training of machine learning models directly on deployed hardware devices without reliance on continuous connectivity to centralized servers.

operational carbon Emissions produced while a system runs, primarily from electricity consumed during training, inference, cooling, and facility operation.

operator fusion An optimization that combines adjacent operations into fewer kernels so intermediate values stay on chip and memory traffic or launch overhead falls.

orchestration The coordination and management of multiple AI systems or agents working together, ensuring proper sequencing, communication, and resource allocation across distributed intelligence systems.

oxide breakdown The failure of an oxide layer in transistors due to excessive electric field stress, causing permanent hardware faults.

P

PagedAttention A KV-cache memory management technique that maps logical token blocks to noncontiguous physical GPU memory pages to reduce fragmentation.

parallel file system (PFS) A distributed file system that stripes data across many storage servers to provide aggregate throughput beyond a single device.

parameter A learnable variable in a machine learning model, such as weights and biases in neural networks, that are adjusted during training to optimize performance.

parameter-efficient fine-tuning (PEFT) Adaptation methods that update only a small subset of parameters or added modules while keeping most pretrained weights frozen.

permanent faults Hardware defects that persist irreversibly until repair or component replacement, consistently affecting system behavior.

personalization layers Model components, typically the final classification layers, that are adapted locally to user-specific data while keeping shared backbone layers frozen.

physical attack Direct manipulation or tampering with computing hardware to compromise the security and integrity of machine learning systems, bypassing traditional software defenses.

pipeline parallelism A distributed training strategy that partitions layers into sequential stages and uses micro-batches to overlap work across devices.

point-in-time join A join that retrieves feature values as they existed at an event timestamp, preventing future data leakage in training sets.

post-hoc explanations Explanation methods applied after model training that treat the model as a black box and infer reasoning patterns from input-output behavior.

power usage effectiveness A metric used to determine the energy efficiency of a data center, calculated as the ratio of total facility energy consumption to IT equipment energy consumption.

precision agriculture The use of technology including GPS, sensors, and machine learning to optimize farming practices by precisely monitoring and managing crop inputs like water, fertilizer, and pesticides.

prefill phase The initial LLM inference phase that processes the prompt and populates the KV cache, often dominated by compute.

principle of least privilege A security concept where users are given the minimum access levels necessary to complete their job functions, reducing security risks.

priority flow control (PFC) An Ethernet link-layer mechanism that pauses traffic by priority class to prevent packet drops, enabling lossless RoCE at operational cost.

privacy budget A concept in differential privacy that represents the total amount of privacy loss allowed across all queries or computations, with each operation consuming part of this finite budget.

privacy-preserving machine learning Techniques and approaches that enable machine learning while protecting the privacy of individuals whose data is used for training or inference.

privacy-preserving techniques Methods designed to protect individual privacy in machine learning, including differential privacy, federated learning, and local processing.

privacy-utility trade-off The fundamental tension between preserving individual privacy and maintaining the utility of data for machine learning, requiring careful balance through techniques like differential privacy.

progressive enhancement pattern A design pattern that establishes baseline functionality under minimal resource conditions and incrementally incorporates advanced features as additional resources become available.

projected gradient descent (PGD) An iterative gradient-based adversarial attack that repeatedly perturbs an input and projects it back into an allowed norm ball.

prompt engineering The practice of designing and optimizing text prompts to effectively communicate with large language models and achieve desired outputs from AI systems.

prompt injection An attack in which adversarial instructions embedded in user or retrieved content cause an LLM to ignore intended constraints.

pruning A model compression technique that removes unnecessary connections or neurons from neural networks to reduce model size and computational requirements without significantly impacting performance.

Q

quantization The process of reducing the precision of model weights and activations from floating-point to lower-bit representations to decrease memory usage and accelerate computation on edge devices

quantization-aware training (QAT) Training that simulates low-precision arithmetic so a model remains accurate after deployment with quantized weights or activations.

quantum machine learning The intersection of quantum computing and machine learning, exploring how quantum algorithms and quantum computers can enhance or transform machine learning tasks.

R

randomized smoothing A certified-defense method that classifies noisy versions of an input to obtain probabilistic robustness guarantees.

RDMA over Converged Ethernet (RoCE) A protocol that carries remote direct

memory access semantics over Ethernet, relying on carefully managed low-loss or lossless behavior.

ReduceScatter A collective communication operation that reduces inputs across ranks and scatters distinct shards of the reduced result back to ranks.

regularization Methods used in machine learning to prevent overfitting by adding penalty terms to the loss function, constraining model complexity.

remote direct memory access (RDMA) A networking mechanism that lets one machine read or write another machine's memory without kernel-mediated copies on the data path.

renewable energy Energy collected from renewable resources that are naturally replenished, including solar, wind, hydroelectric, geothermal, and biomass sources.

resource-constrained environments Deployment contexts with limited computational power, network bandwidth, or power availability, typically requiring specialized system design and optimization techniques.

resource paradox The challenge in social impact applications where areas with the greatest needs often lack the basic infrastructure required for traditional technology deployments, requiring innovative engineering solutions.

responsible ai The practice of developing and deploying AI systems in ways that are ethical, fair, transparent, and beneficial to society while minimizing potential harms and biases

reward hacking The phenomenon where AI systems exploit unintended aspects of reward functions to maximize scores while violating the intended objectives.

Ring AllReduce A bandwidth-efficient AllReduce algorithm that arranges ranks in a logical ring and uses reduce-scatter plus all-gather phases.

rlhf Reinforcement Learning from Human Feedback - a training method that uses human preferences to guide model behavior, enabling AI systems to better align with human values and intentions.

robust ai The ability of artificial intelligence systems to maintain performance and reliability despite internal errors, external perturbations, and environmental changes.

robustness A model's ability to maintain stable and consistent performance under input variations, environmental changes, or adversarial conditions.

robustness metrics Quantitative measures for evaluating model stability under various perturbations, including adversarial accuracy, certified robustness bounds, and performance under distribution shift.

roofline model A performance model that compares arithmetic intensity with peak compute and memory bandwidth to identify compute-bound and bandwidth-bound regimes.

S

scaling laws Mathematical relationships demonstrating power-law scaling between model size, dataset size, compute budget, and performance, suggesting predictable improvements with increased resources.

scan chains Dedicated test paths in processors that provide access to internal registers and logic for comprehensive hardware testing and fault detection.

scope 1 emissions Direct greenhouse gas emissions from sources owned or controlled by an organization, such as on-site fuel combustion and company vehicles.

scope 2 emissions Indirect greenhouse gas emissions from the generation of purchased electricity, steam, heating, or cooling consumed by an organization.

scope 3 emissions All other indirect greenhouse gas emissions that occur in an organization's value chain, including manufacturing, transportation, and end-of-life disposal.

secure aggregation A cryptographic protocol that enables federated learning servers to compute aggregate model updates without accessing individual client contributions, enhancing privacy protection

secure boot A hardware-backed startup process that cryptographically verifies firmware and software before execution.

secure computation Cryptographic protocols that enable multiple parties to jointly compute functions over private inputs without revealing those inputs to each other.

secure multi-party computation A cryptographic method that allows multiple parties to jointly compute a function over their private inputs without revealing those inputs to each other.

selective computation Computational strategies that dynamically allocate processing resources based on input complexity or current needs, improving efficiency by avoiding unnecessary computation.

self-refinement A training approach where models iteratively improve their own outputs by critiquing and refining their initial responses, enabling continuous improvement and better alignment with desired behaviors.

self-supervised learning A machine learning paradigm where models learn representations from unlabeled data by predicting parts of the input from other parts, reducing dependence on manually labeled datasets.

service level objective (SLO) A measurable target for service behavior such as latency, freshness, availability, or model quality.

SHAP SHapley Additive exPlanations, a feature-attribution framework based on Shapley values from cooperative game theory.

sharded checkpointing A checkpointing approach where each worker saves only its assigned model, gradient, or optimizer shard.

side-channel attack An attack that exploits information leaked through the physical implementation of computing systems, such as power consumption, electromagnetic emissions, or timing variations.

silent data corruption (sdc) Undetected errors during computation or data transfer that propagate through system layers without triggering alerts, potentially compromising results.

smallholder farmers Farmers operating on plots smaller than 2 hectares who produce a significant portion of global food supply but often lack access to modern agricultural technology and credit.

social impact measurement Systematic evaluation of how AI applications affect communities and individuals, including metrics for accessibility, equity, effectiveness, and unintended consequences.

software fault Unintended behavior in software systems resulting from defects, bugs, or design oversights that can impair performance or compromise security.

sparse training A training approach that maintains sparsity in neural network weights throughout the training process, reducing computational requirements and memory usage.

sparse updates A training strategy that selectively updates only a subset of model parameters based on their importance or contribution to performance, reducing computational and memory overhead.

specification gaming When AI systems find unexpected ways to achieve high rewards that technically satisfy the objective function but violate the intended purpose.

speculative decoding A serving optimization where a smaller draft model proposes tokens that a larger target model verifies in parallel.

speculative execution A performance optimization in processors that executes instructions before confirming they are needed, which can inadvertently expose sensitive data through microarchitectural side channels.

stale synchronous parallel (SSP) A synchronization model that lets faster workers advance a bounded number of steps ahead of the slowest worker.

state space models Neural architectures that process sequences by maintaining compressed memory representations that update incrementally, offering linear scaling advantages over transformer attention mechanisms.

stochastic computing Computing techniques that use random bits and probabilistic operations to perform arithmetic, potentially offering better fault tolerance than traditional methods.

straggler A correct but slow worker that delays synchronous progress for the rest of a distributed job.

stuck-at fault A permanent hardware fault where a signal line becomes fixed at a logical 0 or 1 regardless of input, causing incorrect computations.

supply chain attack An attack that compromises hardware or software components during the manufacturing, distribution, or integration process, potentially affecting multiple downstream systems.

sustainable ai The practice of developing and deploying artificial intelligence systems that minimize environmental impact while maintaining effectiveness and accessibility.

sustainable development goals A collection of 17 global goals adopted by the United Nations to address pressing social, economic, and environmental challenges by 2030, providing a framework for AI applications in social good.

swarm intelligence Collective intelligence emerging from decentralized, self-organized systems, often inspired by biological swarms and applied to distributed ML systems and robotics.

synthetic data generation The creation of artificial datasets that approximate the statistical properties of real data while reducing privacy risks and avoiding direct exposure of sensitive information.

system-wide sustainability Holistic approach to environmental responsibility that considers the entire AI infrastructure ecosystem, from data centers to edge devices, rather than optimizing individual components in isolation.

T

tail latency High-percentile request latency, such as P99, that captures rare slow responses affecting distributed service reliability.

targeted attack A type of data poisoning attack that aims to cause misclassification of specific inputs or classes while leaving the model's general performance largely intact.

Tensor Core A specialized matrix-multiply-accumulate hardware unit optimized for mixed-precision tensor operations.

tensor parallelism A model-parallel strategy that splits tensor operations or layer weights across devices and combines partial results with collectives.

thermal stress Hardware degradation caused by repeated cycling through high and low temperatures, leading to material fatigue and potential failures.

threat modeling A structured security analysis that identifies assets, adversary capabilities, attack paths, and defensive priorities for a system.

time per output token (TPOT) The per-token latency after the first generated token, controlling the pace of streamed LLM output.

time to first token (TTFT) The latency from request arrival until the first generated token appears.

tinyml A field focused on deploying machine learning models on microcontrollers and extremely resource-constrained devices with kilobytes of memory and milliwatts of power consumption.

topology-aware scheduling Placement that accounts for GPU and interconnect locality such as NVLink domains, racks, rails, or switch hierarchy.

tpu Tensor Processing Unit, a specialized AI accelerator chip developed by Google specifically designed for machine learning workloads, particularly neural network computations.

training The process of teaching a machine learning model to make predictions by showing it examples and adjusting its parameters based on performance feedback.

training-serving skew A mismatch between training-time and serving-time feature or data computation that causes production inputs to differ from what the model learned.

transfer learning A machine learning technique that uses knowledge from a pre-trained model on one task to improve learning on a related task, enabling efficient adaptation with limited data

transformer A neural network architecture that uses self-attention mechanisms to process sequential data, forming the foundation for many modern language models.

transformer architecture A neural network architecture based on attention mechanisms that has revolutionized natural language processing and is increasingly applied to other domains like computer vision.

transient faults Temporary hardware faults that do not persist or cause permanent damage but can lead to incorrect computations if not handled properly.

transparency Openness about how AI systems are built, trained, validated, and deployed, including disclosure of data sources, design assumptions, and limitations.

Tree AllReduce An AllReduce algorithm that uses tree reduce and broadcast phases to reduce latency, usually at the cost of bandwidth efficiency.

triple modular redundancy (tmr) A fault-tolerance technique where three instances of a computation are performed, with majority voting determining the correct result.

trusted execution environment A secure area within a processor that provides hardware-based protection for code and data, ensuring confidentiality and integrity even from privileged system software.

two-phase commit (2PC) A coordination protocol with prepare and commit phases that ensures participants commit atomically or abort.

V

value alignment The principle that AI systems should pursue goals consistent with human intent and ethical norms, addressing the challenge of encoding human values in machine objectives.

value-sensitive design A methodology for incorporating human values into technology design through systematic stakeholder engagement and ethical consideration of system impacts.

vector-borne diseases Diseases transmitted by insects or other vectors, such as malaria carried by mosquitoes, which can be monitored and controlled using machine learning-powered detection systems.

vision-language models AI systems that can understand and reason about both visual and textual information simultaneously, enabling tasks like image captioning, visual question answering, and multimodal understanding.

W

warehouse-scale computer (WSC) A building-scale computing system operated as one coherent machine, with the network fabric and storage hierarchy treated as system-level resources.

watchdog timer A hardware component that monitors system execution and triggers recovery actions if the system becomes unresponsive or stuck.

water usage effectiveness A metric that measures the efficiency of water use in data centers, calculated as the ratio of total water consumed to IT equipment energy consumption.

weight freezing A technique that fixes most model parameters during training while allowing only specific layers or components to be updated, reducing computational requirements for on-device adaptation.

white-box attack An adversarial attack where the attacker has complete knowledge of the model's architecture, parameters, training data, and internal workings, enabling highly effective attack strategies.

Z

ZeRO (Zero Redundancy Optimizer) A memory-reduction technique that shards optimizer state, gradients, and optionally parameters across data-parallel workers.

zero-day vulnerability A previously unknown security flaw in software or hardware that can be exploited by attackers before developers have had a chance to create and distribute a patch.

zero-shot learning The ability of machine learning models to perform tasks or classify objects they have never seen during training, often achieved through sophisticated representation learning or large-scale pretraining.

1-bit Adam, 286
1-bit Quantization
  edge quantization, 452
1-bit SGD, 283
1F1B Scheduling
  memory efficiency, 228, 229
2.5D interposers, 96
3D Parallelism, 20, 121
  definition, 238
  etymology, 238
3D stacking, 96
48V DC busbar, 90

A/B Testing
  power, 689
  statistical power, 689
A11 Bionic
  on-device learning, 582
Abstention
  safety mechanism, 957
Abstraction Gap, 332
Accelerator
  selection, 68
  spectrum, 41
Access Pattern, 142
Accountability, 936
Action Selector, 344
Activation Caching
  memory overhead, 639
Activation Checkpointing, 84
  memory reduction, 201
Adam Optimizer, 286
  checkpoint memory, 333
  state, 333
  state size, 177
Adaptation Strategy
  trade-offs, 606
Adapter
  per-context efficiency, 599
  storage efficiency, 599
Adapter Switching, 603
Adaptive Routing, 127
Admission Control, 357
Advanced Composition, 813
Advanced Packaging, 96
Adversarial Attack
  categories, 844
  definition, 844
Adversarial Examples
  discovery, 759
Adversarial Inputs
  robustness, 951
Adversarial Perturbation, 844, 845, 970
  gradient-directed, 844
Adversarial Prompt Evasion, 793
Adversarial Training, 853
  accuracy cost, 832
  implementation, 854
Adversarial Transferability
  black-box, 769
AES
  AES-256, 797
  side-channel vulnerability, 777
Aggregation
  small-file mitigation, 156
Aggressive Sharing, 415
AI Compute Growth
  doubling time, 868
  growth rate, 868
AI Safety
  research field, 996
AI Triad, 25
Air-cooling limit, 1067
Air-Gapped
  isolation, 753
Alert Fatigue, 699
Algorithm
  D·A·M taxonomy, 22
Algorithm Aversion
  human-AI interaction, 981
Algorithmic Fairness
  definition, 938
Algorithmic Stability, 195
Alignment Bypass, 771
Alignment Regression Testing, 979
Alignment Tax, 978
All-Pairs Bandwidth Matrix, 132
All-to-All
  embedding communication, 50
  expert parallelism, 233
AllGather, 265
AllReduce, 104, 212, 265, 461
  algorithm comparison, 274
  etymology, 255
  forward reference, 104
  inter-node, 279
  ring algorithm, 212
AllToAll, 265
  communication, 537
Alpha-Beta Model, 112
  definition, 259
Amdahl's Law, 97
  distributed, 24
AOC, 110
AOTAutograd, 455
API Keys
  credential leakage, 791
API Security
  ML serving, 956
API Security for ML, 956
Apple Neural Engine, 594
Application Monitoring, 131
Approximate Behavioral Theft, 761
Archetype
  A (GPT-4), compute infrastructure, 50
  A (GPT-4/Llama-3), 10
  B (DLRM at Scale), compute infrastructure, 50
Architecture
  disaggregated, 135
Archive Tier, 175
Arduino
  memory constraint, 591
Arduino Nano 33 BLE Sense, 591
Arithmetic Intensity, 21, 57, 1068
  performance engineering, 439
  serving, 525
ARM Cortex
  device heterogeneity, 636
ARM Cortex Spectrum, 636
ARM TrustZone
  adoption, 798
Assumptions
  capacity planning, 1106
  cluster tiers, 1100
  network, 1101
  reliability, 1102
  sustainability, 1104
Asynchronous Coordination
  federated learning, 626
Asynchronous SGD
  staleness penalty, 203
Atomic Allocation, 380
Atomic Unit of Failure, 366
Attained Service Scheduling, 407
  two-dimensional, 407
Attention
  maps, security monitoring, 793
Auditability
  fleet governance, 22
Auditable Autonomy, 629
Autoencoder
  anomaly detection, 857
Automated Pre-Flight Checks, 698
Automation
  automation cost, 656
Automation Bias, 981
AutoML, 732
Automotive Recalls
  cybersecurity, 754
Automotive Safety Integrity Level
  ML certification, 827
Autopilot
  perception failure, 825
Auxiliary Load Balancing Loss, 234
Auxiliary Loss, 533
Availability Attack, 767
Average Odds Difference
  definition, 945
AWQ, 451, 568
AWS S3 Outage
  cascade failure, 824
AWS Trainium, 43

Backdoor Attack, 767
  BadNets, 767
  detection asymmetry, 849
Backfill, 726
Backfill Scheduling, 382
Backup Worker, 345
Bandwidth
  bound, 260
  effective, 278
  lower bound, 268
  network, 212
  optimality, 268
  per-node PFS, 178
  reduction, 613
  register file, 49
  storage hierarchy, 146
  survival strategy, 595
  theoretical, 337
Bandwidth Hierarchy, 76
  parallelism placement, 101
Bandwidth-Bound, 525
Bandwidth-dominated communication, 1074
Barrier Synchronization, 10
  vs. independent tasks, 10
Batch Size, 212
  8 achieves stability, 1094
  latency knee, 1094
  maximum, 476
Batch/Research, 418
Batching, 493
  batch, accumulation window, 508
  batch, sizes 1-4 are unstable, 1094
  differentiated, 506
  diminishing returns, 1094
  efficiency curve, 495
  max batch size, 511
  preferred batch size, 511
  timeout parameter, 511
Bathtub Curve, 315
Battery
  on-device training impact, 596
Bayesian Neural Networks
  uncertainty estimation, 854
BF16, 69, 570
Bias
  amplification feedback loop, 955
  automation, human-AI interaction, 981
Bias-Only Adaptation, 599
Bin Packing
  latency-aware, 416
  scheduling, 379
Binary Neural Network, 920
  edge quantization, 452
Biofilm, 904
Bisection Bandwidth
  definition, 116
  dominance, 10
  etymology, 8
  wall, 8
Bit-Flip Error, 320
Black-Box Access, 757
Block Quantization, 283
Block-wise Quantization
  definition, 450
Blue-Green Deployment
  blue-green exposure, 682
  ML serving, 675
Bottom-Up Sizing, 668
Bounded Staleness, 345
Brain Power Efficiency, 627
Broadcast, 265
BSP

synchronization, 202
Bucket Fusion, 293
Budget
burn rate, 707
Build vs. Buy, 662
storage, 175
Bulk Synchronous Parallel
definition, 124
origin, 107
Bulkhead Pattern
isolation, 553
Burn-in Testing
reliability screening, 315
Butterfly Algorithm, 273
Byte-Pair Encoding, 13
Byzantine Failures, 310
Byzantine Fault Tolerance
federated learning, 642
ML overhead, 313
Byzantine-Resilient ML, 313

Calibration, 940
group-specific, 939
score, 945
Canary Deployment
canary, exposure, 682
gradual rollout, 682
ML rollout, 795
safety validation, 1000
CAP Theorem, 21
availability, 21
consistency, 21
ML trade-off, 21
partition tolerance, 21
scheduling, 376
Capacity Factor, 533
mixture of experts, 234
Capacity Planning, 497
Capacity Wall, 225
CapEx, 661
facility capex, 664
hardware, 664
network, 664
Capital Expenditure, 661
Carbon Border Adjustment Mechanism, 926
Carbon Budget, 917
Carbon Emissions
grid intensity, 18
Carbon Footprint
benchmarks, 891
model, 914
operational, large-scale ML, 892
tracking, 875
training emissions, 865
Carbon Intensity, 899
by energy source, 878
variance, 875
Carbon-Aware Scheduling, 922
at scale, 922
optimal region, 866
results, 922
Carlini-Wagner Attack
defense evaluation, 846
Catastrophic Forgetting
definition, 608
CCPA
data deletion, 951
Centralized Checkpointing, 339
Certified Defenses
robustness, 971
Certified Robustness, 853
Channel Pipelining, 289
Chaos Engineering, 365
Chargeback, 406
Checkpoint
storage architecture, 177
Checkpoint Consistency
in-flight gradients, 368
partial writes, 368
rank desynchronization, 368
Checkpoint Format Optimization, 180
Checkpoint Storm, 333
definition, 143
storage traffic, 178
Checkpoint Tax, 667
Checkpoint-Restart
training completion, 325
Checkpoint/restart, 1062
Checkpointing
async PFS copy, 178
asynchronous, pipeline overhead, 338
bounded asynchronous, 340
definition, 333
frequency tuning, 713
incremental, 349
local checkpoint staging, 153
overhead, 178
tiered, 349
training pause, 178
zero-redundancy, 180
Chi-Square Goodness-of-Fit Test, 840
Chiplet, 45
Chunked Prefill
chunk sizing, 527
interleaving schedule, 527
latency, 526
CI/CD Patterns by Model Type, 696
Circuit Breaker
circuit breaker pattern, 548
half-open state, 357
open state, 357
safety mechanism, 1000
serving, 548
serving resilience, 357
Circular Economy, 916
Clean-Label Poisoning, 860
Client Drift
federated learning, 617
Client Scheduling
federated learning, 619
Clock Gating
definition, 66
Clos Network, 118
Closed-Loop Governance, 746, 980
Cloud Computing
ML infrastructure, 667
Cloud ML, 958
Cloud-Lite Fallacy, 29
Cloudlet
hybrid training, 640
Cluster
geometry, 110
MTBF scaling, 307
Cluster Design, 129
Clustered Federated Learning, 622
CMOS Power Equation, 875
Co-Packaged Optics
power savings, 134
Code Generation, 564
CodeCarbon, 927
Cold Restart, 343
Cold Start, 731
GPU inference, 556
warmup, 561
Cold Start Latency, 412
Cold Tier, 175
Cold-Start Problem, 634
personalization, 634
Collective Operation
definition, 263
Collective Primitives
forward reference, 104
Columnar Format, 161
Communication
event timeline, 467
Communication Intensity
definition, 20
Communication Overhead, 195
Communication Tax
scaling efficiency, 215
Communication Wall, 105
Communication-Bound
optimization path, 473
Communication-bound workload, 1057, 1070
Communication-Computation Overlap, 129
Communication-Computation Ratio
definition, 194
COMPAS
algorithmic bias, 938
Compliance
legal, 983
Computational Storage Drive, 176
Compute Express Link, 176
Compute to the Data, 613
Compute-Bound, 58, 439, 495
optimization path, 473
Compute-Limited Regime, 15
Compute-Optimal Training
resource allocation, 14
Concept Bottleneck Models
interpretability, 974
Concept Drift, 702
definition, 836
ground-truth feedback, 837
Confidence Calibration
drift detection, 631
Confidence Rounding, 765
Congestion Control
PFC and ECN counters, 132
Congestion Spreading, 125
Consistent Hashing
determinism, 542
failure handling, 543
KV cache, 542
minimal disruption, 542
virtual nodes, 542
Constitutional AI
alignment, 978
Container Isolation
ML serving, 794
Contestability, 985
Contestability Stack, 986
appeal routing, 986
decision provenance, 986
explanation generation, 986
outcome tracking, 986
Context Filtering
safety mechanism, 978
Context Propagation, 360
Continual Learning
edge devices, 628
Continuous Batching, 73, 413, 474
definition, 499
iteration-level scheduling, 499
Continuous Integration, 677
Continuous Validation, 728
Contrastive Learning, 859
edge self-supervised, 628
Contribution Analysis
layer selection, 605
Control Plane, 21, 648
Controller Pattern
Kubernetes, 387
Cooldown Period, 413
Cooling, 88
energy density, 922
immersion, 900, 923
liquid flow rate, 66
system failure, 313
Coordination Tax
scaling efficiency, 215
Corporate Sustainability Reporting Directive, 926
Correlated Failure, 313
shared power domain, 367
shared software, 367
shared switch, 367
thermal correlation, 367
Correlation Analysis, 698
Coscheduling
Kubernetes scheduling, 388
Cost
annual cost, 664
anomaly root cause categories, 706
attribution schema, 706
per active user, 707
quality trade-off analysis, 714
Cost Efficiency, 378
Cost Per Inference, 713
Cost Per Token, 72
Cost Policy, 405
Cost-Aware Routing, 566
Counterfactual Explanations, 974

recourse, 974
Covariate Shift, 702
CPU Offloading, 84
CPU-GPU Synchronization, 467
CRC
gradient validation, 328
Critical Batch Size
convergence limit, 248
definition, 218
Critical Materials, 911
AI hardware, 911
Critical Message Size, 259
Critical Path, 670
Crossbar Switch, 80
NVSwitch, 80
Crossover Arithmetic Intensity, 881
Crowdsourcing
poisoning risk, 767
Cryptographic Hash
validation cost, 169
CUDA
graphs, overhead reduction, 447
streams, 462
CUDA Time-Slicing, 414
Curriculum Learning
data locality, 143
Custom Accelerator, 667
CXL
coherent interconnect, 96
definition, 96
memory pooling, 96
Cybersecurity Regulations
ML compliance, 780
C^3 Taxonomy, 22
communication, 22, 1057
computation, 1056
compute, 22
coordination, 22, 1057
tax, 1060

DAC, 109
DALI, 151
Dark Launching, 731
Data
data freshness, 727
limited regime, 15
mobile, sensor scale, 627
non-iid, convergence impact, 614
sensitivity distribution, 814
Data Augmentation
techniques, 858
workspace memory, 151
Data Center
emissions scale, 890
energy projections, 868
environmental justice, 948
hyperscale footprint, 890
industrial electricity demand, 867
industrial scale, 867
Data Contract, 728
Data Debt, 716
Data Drift, 734, 836
Data Efficiency, 18
edge constraints, 598
Data Engineering, 25
Data Format
compression integration, 162
index, 161
Data Gravity
placement economics, 169
Data Harmonization, 991
Data Leakage
feature store risk, 724
Data Loader
pipeline architecture, 150
Data Loading
batching, 87
host-to-GPU transfer, 87
pipeline, 86
preprocessing, 87
reading, 87
Data Locality, 613, 1014
Data Movement
energy cost, 70
Data Nutrition Project, 991
Data Parallelism, 78, 120, 129, 258
definition, 198
pure, 84
Data Pipeline
throughput equation, 162
Data Poisoning, 767
attack, 850
concept poisoning, 851
definition, 848
efficiency, 759
examples, 849
impact, 850
preview, 22
training attack, 848
Data Processing Unit, 81
Data Provenance
ML pipelines, 858
Data Quality, 727
Data Sanitization
poisoning defense, 858
Data Shuffling
quality vs. I/O efficiency, 168
shard-level, 168
Data Stall Ratio
definition, 163
Data Storage, 140
Data Transfer Cost, 173
Data Upload
raw, 613
Data Validation, 677
Data Wall
definition, 182
Dataset Scale, 195
Dataset Versioning, 169
Datasheets
dataset documentation, 952
DCQCN, 127
congestion control, 127
DDoS
ML inference, 755
Deadlock
circular dependency, 382
hold-and-wait, 382
pipeline parallelism, 330
Debug Port
debug port vulnerabilities, 779
JTAG vulnerability, 779
Declarative Model, 387
Decode
inference phase, 73
Decode Bottleneck, 1010
Decode Phase, 444, 524
Deep Leakage from Gradients, 786
DeepSeek-V3, 532
Defense in Depth, 757
Defense Selection Framework, 805
Defensive Distillation, 855
Defragmentation, 379
Deionized Water, 904
Delta Encoding, 184
Demographic Parity
definition, 940
final prediction, 940
Dennard Scaling, 66
power wall, 66
sustainability, 869
Dense Layer, 68
Deployment
consistency models for multi-region, 685
declarative API, 708
infrastructure, 26
patterns by scale, 675
trade-offs, 958
Depreciation, 664
Depthwise Separable Convolution, 592
parameter reduction, 592
Determinism Violations, 310
Deterministic Checkpoint Time, 335
Device Plugin, 387
DGX, 80
Differential Privacy, 624
central, 787
dataset size threshold, 814
definition, 806
federated learning, 585, 624
gaussian mechanism, 810
gradient leakage, 584
local, 787
mean-estimation noise, 751
privacy, 22
task complexity, 814
training overhead, 966
Dimension-Ordered Reduction, 281
Diminishing Returns, 1097
Direct Preference Optimization
memory budget, 243
Disaggregated Fairness Metrics, 1003
Disaggregated Serving
decode pool, 526
Disparate Impact Ratio
definition, 944
Displacement of Overhead, 23
DistBelief
distributed training, 197
Distributed Checkpoint, 180
Distributed Checkpointing, 339
Distributed Counters with Gossip, 541
Distributed Inference
latency constraint, 485
throughput constraint, 485
Distributed Scheduling
challenges, 428
Distributed Tracing, 361
ML pipelines, 704
Distributed Training
definition, 197
energy overhead, 890
DistributedDataParallel, 293
Distribution Layer, 194, 338
analysis, 424
Distribution Shift, 533, 834
types, 951
District Heating, 870
DLRM, 28
AllToAll, 258
scale, 485
DMR
detection trade-off, 328
Double Binary Tree, 273
DP-SGD
training cost, 967
training overhead, 809
Dragonfly
topology origin, 122
Dragonfly Topology, 122
DRAM, 880
data pipeline staging, 150
residue fault, 323
Drift Detection
on-device models, 631
Dropout
MC, inference cost, 855
robustness, 854
Dual-Store Architecture, 722
Dual-Use Dilemma
ML security, 851
Duck Curve, 915
Durability
eleven nines, 158
Duty Cycle
accounting, 884
DVFS
accelerator power management, 66
Dynamic Batching, 494
sequence length variance, 245
Dynamic Compute Allocation
definition, 512
Dynamic Load Balancing, 345
Dynamic Placement, 169
Dynamic Power, 876
Dynamic Sharing, 416
D·A·M Taxonomy, 22
algorithm, 22
data, 22
machine, 22

E-Waste
AI hardware, 867
ECMP
hash collision, 127

ECN, 127
Edge AI
  energy paradox, 894
  power budget categories, 907
Edge Computing
  energy paradox, 894
  robustness trade-off, 825
Edge Inference, 583
Edge Intelligence
  definition, 582
Edge Learning
  integration controls, 634
  paradigm, 613
Edge ML, 958
Edge Power Measurement Instruments, 884
Effective FLOP/s, 1059
Effective service rate, 1092
Efficiency
  brain, edge learning, 627
  distributed, 23
  energy dimension, 23
  facility, 877
  interconnect, 1014
  power, 665
  program, 711
  resource, 983
  runtime, 710
  scheduling, 710
Efficiency Frontier
  definition, 436
Egress Fee, 173
Elastic Net Attack to DNNs, 846
Elastic Scheduling, 399
  elastic range, 398
  transition cost, 399
Elastic Training
  definition, 346
  fleet fault tolerance, 1062
  framework support, 349
  persistence, 370
Electrical Interconnect
  energy per bit, 1014
Electricity
  annual electricity, 664
Electromigration
  wear-out mechanism, 315, 322
Elephant and Mice Flows, 107
Elephant Flows
  network traffic, 107
Embedded AI, 915
Embedded Systems
  ML reliability, 827
Embedding
  versioning, 349
Embedding Lookup, 508
Embedding Sharding, 7, 232
Embedding Table, 151
Embodied Carbon, 875
  amortization formula, 887
  hardware lifecycle, 875
Emissions Trading
  compute cap, 926
Emissions Trading for Compute, 926
Emissions Trading System, 926
Energy, 23
  buffering, 870
  carbon-free, 24/7 challenge, 922
  gap intervention cascade, 872
  household, sustainability baseline, 867
  privacy trade-off, 957
Energy Delay Product, 921
Energy Efficiency, 879
  by architecture, 879
Energy Harvesting
  power budgets, 909
Energy Roofline Model, 881
Energy Tax
  communication overhead, 215
Energy Wall, 4, 1014
Energy-Scale Invariant, 23
Energy-to-Communication Ratio, 909
Enhanced Transmission Selection, 131
Environmental Justice
  data center siting, 868
EP, 532
Epistemic Uncertainty
  robustness, 828
Equal Opportunity Difference, 945
Equal Split, 707
Erasure Coding
  definition, 158
  Reed-Solomon, 158
Error Feedback
  definition, 285
Error Masking, 331
Error Model, 330
ESP32
  edge training, 593
ETS, 131
EU AI Act, 22
  energy reporting, 926
EUV Lithography
  energy cost, 893
Evaluation, 677
Event-Driven Computing
  energy, 872
Eventual Consistency, 21, 340
EWC
  continual learning, 628
Exact Artifact Theft, 761
Exclusive Access, 415
Expected Load Per Server, 541
Experience Replay
  forgetting prevention, 634
Expert Buffering, 533
Expert Collapse, 533
Expert Parallelism, 532
  conditional computation, 233
  definition, 234
Explainability, 936
Exponentially Distributed Failures, 335

Fabric Behavior, 124
Fail-Safe Defaults, 758
Fail-Stop Failure, 312
Failover
  active-active, 565
  gradual recovery, 565
Failsafe Mechanism
  ML serving, 827
Failure
  at scale, 101
  inevitability at scale, 370
  model-type impact, 316
Failure Classifier, 344
Failure Domain, 117
  ML infrastructure, 313
Failure Rate
  annualized count, 5
  daily, 4
  FITs, 303
  fleet scale, 8
Failure-Domain-Aware Placement, 157
Fair-Share Scheduling, 385
Fairlearn
  bias toolkit, 960
Fairness, 377
  gerrymandering, 943
  impossibility theorems, 942
  incompatibility, 939
  stability constraint, 936
  tax, 939
  threshold-dependent, 961
  validation in ml, 681
  validation, impossibility trade-off, 681
Fake Profile Injection
  definition, 849
Fallback, 356
Fallback Redundancy, 827
Fallback Strategies, 713
False Positive Rate, 699, 1003
Fan-Out Cost, 731
Fast Gradient Sign Method, 845
  adversarial training, 845
Fat-Tree
  Clos origin, 118
  definition, 118
Fault Characteristics, 332
Fault Injection, 330
Fault Injection Attack, 775
Fault Model, 330
Fault Tolerance, 195
  definition, 302
  detection speed, 366
  spectrum, 366
  stateless vs. stateful serving, 351
Feature Complexity, 721
Feature Engineering, 677
Feature Extraction, 509
Feature Freshness Requirements by Type, 723
Feature Interaction, 507
Feature Squeezing, 854
  adversarial defense, 854
Feature Staleness, 734
Feature Store, 169
  definition, 649
Feature-Parallel Batching, 492
FEC, 109
  latency tax, 110
FedAvg
  communication efficiency, 615
Federated A/B Testing, 633
Federated Analytics, 695
  privacy-preserving telemetry, 631
Federated Averaging Protocol, 615
Federated Coordination
  edge constraints, 598
Federated Learning, 812
  definition, 613
  edge devices, 7
  etymology, 586
  fairness constraint, 955
  introduction, 28
  local data sampling, 614
  model aggregation, 614
  model initialization, 614
  model update, 614
  model updates, 613
Fermi Estimate, 1015
Fidelity, 332
Filter Bubble, 732
Financial Loss, 336
Fine-Tuning
  selective, on-device, 591
Finish-Time Fairness, 408
FinOps
  definition, 712
FIPS 140-2
  security levels, 802
First-Come-First-Served Scheduling, 492
Five-Level Model, 107
Five-Pillar Framework, 25
FlashAttention, 447
Flat Network, 277
Fleet Anomaly Attribution, 701
Fleet Baseline Comparison, 472
Fleet Energy Productivity, 24
Fleet Law, 23
Fleet Orchestrator, 11
Fleet Stack, 104
  definition, 25
  textbook structure, 28
Fleet-Level Property, 1000
FLOP/s
  energy distinction, 918
  work vs. throughput, 14
FLOP/s vs. FLOPs, 14, 918
Flowlet Switching, 127
Format Problem, 155
Forward Error Correction, 109
Four-Fifths Rule, 944
FP16, 69
FP32, 69
FP4 Support, 526
FPGA
  fault injection, 331
Fragmentation
  scheduling, 379
Fragmentation Index, 419
Fraud Detection, 658
Free cooling
  site selection, 669
FSDP

memory trade-off, 265
Full Fine-Tuning, 599
Full Mesh, 74
Fully Sharded Data Parallel
memory reduction, 208

Galvanic Corrosion, 904
Gandiva, 407
iteration-boundary time slicing, 407
Gang Scheduling
deadlock prevention, 428
definition, 380
origins, 375
starvation, 424
Gboard
federated learning, 586
GDPR
Article 22, 938
ML systems, 775, 796
on-device learning, 584
GELU
kernel, 445
GEMM
kernel, 445
Generality Tax
accelerator design, 41
Generative Alignment, 978
GHG Protocol, 894
scope taxonomy, 895
Global Explanation, 938
Global Privacy Budget, 813
Gloo, 290
Go-Back-N, 112
Goodput, 1088
cluster-wide, 408
program, 23
runtime, 23
scheduling, 23
Goodput Ratio, 1057
Google Sustainability Framework
machine, 925
map, 925
mechanization, 925
Google Tensor SoC
federated learning, 595
Governance Layer, 748, 934
GPFS
distributed metadata, 155
GPipe
pipeline parallelism, 228
GPT-3
energy consumption, 871
energy scale, 871
GPTQ, 451, 568
column-wise update, 568
GPU, 41
fault rate, 320
fault rates, 320
hardware failure, 309
manufacturing embodied carbon, 867
MTBF variation, 304
power monitoring, 883
procurement, 666
GPU Memory
pressure, 545
GPU Utilization, 73, 411, 501
check, 132
efficiency, 707
GPUDirect RDMA, 81, 111
zero-copy, 111
GPUDirect Storage, 87
data path, 172
definition, 170
GPUDirect family, 170
Graceful Degradation, 634, 828
definition, 355
triggers, 358
GradCAM
visual explanation, 973
Gradient
bus, 104
quantization, communication, 620
sparsification, top-k, 620
Gradient Accumulation, 86
communication overhead, 207
Gradient Checkpointing
edge memory, 597
Gradient Clipping, 810
Gradient Compression, 260, 1010
bandwidth reduction, 222
Gradient Hook, 293
Gradient Inversion Attack, 786
Gradient Offloading, 86
Gradient Staleness
definition, 216
Gradient Synchronization
definition, 254
energy cost, 891
Grand Teton, 95
Graph Break
definition, 455
Graph Capture, 454
Graph Compilation
graph-level optimization, 454
Gray Failure, 345, 698
Gray Failure Detection, 472
Gray-Box Access, 757
Greenwashing
AI sustainability, 927
Grid Carbon
marginal emissions, 923
Group Distributionally Robust Optimization, 837
Group DRO, 837
Grouped Query Attention, 521
Grow-Shrink Elasticity, 408
GSPMD, 455

Hallucination
generative AI, 842
Hamming Code
ECC origin, 327
Hardening Strategy
ML systems, 823
Hardware Security
threat landscape, 773
Hardware Security Mechanisms, 795
Hardware Security Module, 801
certification, 802
performance trade-off, 801
Hardware Utilization, 471, 590
Hardware-in-the-Loop, 695
HBM, 49
bandwidth, 53
definition, 50
energy efficiency, 52
memory hierarchy, 437
origin, 49
per-GPU usage, 86
storage hierarchy, 149
HBM3
bandwidth, 484
HDD
random IOPS bottleneck, 153
Headroom
facility design, 670
Health Check, 350
timeout configuration, 547
Health Monitoring Daemon, 344
Healthcare Algorithm
proxy bias, 939
scale, 939
Healthcare ML
FDA compliance, 794
Healthcare ML Compliance, 794
Heartbeat, 340
Heat Rejection, 902
Hedged Requests, 157
Heisenbug
distributed debugging, 359
Hessian
quantization, 567
Heterogeneity of the Fleet, 613
Heterogeneous Gang Scheduling, 384
Hexagon DSP
mixed-precision training, 597
Hey Siri
power constraint, 608
Hierarchical AllReduce, 277
Hierarchical Capability Matching, 634
Hierarchical Network, 277
Hierarchical Placement, 395
Hierarchical Profiling, 470
Hierarchy-Aware Parallelism
definition, 237
High Bandwidth Memory, *see* HBM
High Likelihood/High Impact, 755
High Likelihood/Low Impact, 755
HIPAA
enforcement, 774
ML compliance, 774
ML requirements, 795, 796
on-device learning, 584
HNSW, 182
DRAM footprint, 182
Hoarding Problem, 421
Holistic Preemption, 383
Homomorphic Encryption, 788
etymology, 788
FHE, 797
Hop Count, 120
Horizontal Pod Autoscaling, 411
Horovod, 12
Host Networking, 389
Hot Tier, 175
Household Energy Baseline, 867
HPA, 411
HPCC, 127
congestion control, 127
Huber Loss
poisoning defense, 858
Human Oversight, 936
Human vs. Machine Perception, 844
Human-in-the-Loop
systems overhead, 936
Hybrid Architecture, 391
Hybrid Parallelism
definition, 236
Hybrid Search, 182
Hysteresis, 413

I/O Bottleneck, 86
I/O Jitter, 165
I/O Pipeline, 86
I/O Wall
definition, 141
IB, 111, *see* InfiniBand
Idle Accelerator Cost, 174
Idle Resource, 336
IEEE 754
bit-flip vulnerability, 319
ImageNet
bottleneck analysis, 153
In-Memory Computing, 927
In-Network Telemetry, 127
In-Place Operations, 920
In-Processing Approaches
fairness, 961
Incast, 128
algorithm selection mitigation, 128
definition, 128
layer-staggering mitigation, 128
Incident Response, 728
Incremental Backup, 333
Industrial Coupling, 870
Infant Mortality, 315
Inference
configuration for gpu, 549
decode phase, 905
distribution tax, 486
infrastructure, 72
prefill phase, 905
prefill vs. decode, 524
risks, 750
stateful serving, 493
streaming, 542
InfiniBand, 111
origin, 112
RDMA, 8
Infinite Timeline, 335
Influence Functions
data sanitization, 858
Infrastructure Layer, 338
analysis, 423

Infrastructure Planning, 668
Inherently Interpretable Models, 974
Input Validation, 792
Instance Diversification, 713
Institutional Deference, 981
INT4, 70, 570
INT8, 70, 570
Integrated Gradients
  attribution method, 973
Intel SGX
  memory constraints, 798
Inter-Token Latency, 466
Interactive/Debug, 418
Interconnect
  bandwidth, 537
  latency, 537
Interconnect Fabric, 667
Interference
  GPU sharing, 415
Interleaving Experiments
  sample efficiency, 689
Intermittent Fault, 320
  mechanism, 323
Internal and External Fragmentation, 517
Intra-package interconnect, 78
Invariant, 22
IOPS
  charges, 174
  ML irrelevance, 143
IoT
  device growth, 915
  device vulnerabilities, 779
  security vulnerabilities, 779
Iron Law, 20, 434, 583, 836, 934
Iron Law of Machine Learning Fleet Efficiency, 710
Iron Law of Scale, *see* Fleet Law
IsoFLOP Curve, 14
IT Power, 901
Iterative Statefulness, 10

Jacobian-Based Saliency Map Attack
  definition, 846
Jailbreak, 704
Jailbreaking, 771
Jensen-Shannon Divergence, 839
Jevons Paradox, 917
  AI rebound effect, 917
Job Preemption
  checkpoint trade-off, 709
Job Restart, 343
Job Scheduling Delay, 367
Job Termination, 343
Joule, 864
Joules Per Token, 921

k-Anonymity
  privacy technique, 963
Kernel
  timeline, 467
Kernel Autotuning, 456
Kernel Fusion, 63
Knee of the Curve, 1096
Knowledge Distillation
  sustainability, 919
Knowledge Transfer, 585
Kolmogorov-Smirnov Test
  adversarial detection, 854
Krum, 313
Kubeflow
  orchestration, 678
Kubernetes
  gpu allocation, 387
Kubernetes-over-Slurm, 391
Kueue
  Kubernetes job queueing, 389
  Kubernetes scheduling, 389
Kullback-Leibler Divergence
  drift detection, 839
KV Cache, 177
  definition, 492
  handoff, 527
  inference autoscaling, 412
  quantization, 520
  serving, 412
  serving fault tolerance, 354

L1 Cache, 880
L2 Cache, 880
Laplace Mechanism, 809
Laser Fault Injection, 775
Last-Mile Problem, 129
Latency, 378, 436
  bandwidth hierarchy, 76
  bound, 260
  breakdown, 508
  compute time, 56
  compute trade-off, 158
  detection, 703
  dominated regime, 114
  gate thresholds by model type, 680
  memory time, 56
  standard, 120
  storage hierarchy, 146
Latency Budget, 509
Latency-dominated communication, 1074
Layer-Wise Partitioning
  model parallelism, 226
Layered Defense Stack, 785
LayerNorm
  kernel, 445
Layers of Responsibility, 989
Leaf Switch Domain, 393
Leakage, 724
Leakage Current, 876
Learning Rate Warmup, 217
Lease-Based Resource, 383
Least Privilege
  ML pipeline security, 856
Level of Abstraction, 708
Life Cycle Assessment, 912
  AI hardware, 912
  AI system lifecycle, 913
Lifecycle Assessment, 1011
Lifecycle Carbon Calculation, 888
Lifecycle Policy, 160
  storage tiering, 161
Lighthouse Archetypes
  at scale, 27
  fleet scale, 6
Lightweight Transformer
  mobile training, 596
LIME
  explainability overhead, 954
Lineage Tracking, 727
Linear Scaling Fallacy, 30
Linear Scaling Rule
  convergence, 248
Link Aggregation, 260
Link Budget, 110
Link Contention, 260
LinkDownedCounter, 132
Lipschitz Constant, 833
  robustness, 971
Lipschitz Continuity
  robustness, 833
Liquid Cooling, 923
Liquid Cooling Flow Rate, 66
Little's Law
  capacity planning, 497
  verification using, 1094
Liveness Probe, 342
Llama-3
  lighthouse archetype, 28
LLM, 680
  conversations, 542
  decode phase, 495
  inference demand, 488
  inference, memory bottleneck, 56
Load Balancing, 350
  inference, 484
  metrics, 534
Load Shifting, 922
  software, 922
Local Explanation, 938
Local Learning and Sample Efficiency, 872
Local SGD
  communication frequency, 223
Locality
  temporal, 143
Locality Cost, 394
Locality-Aware Scheduling, 169
LogGP model, 1075
  crossover message size, 1074
  link bandwidth, 1074
  message size, 1074
  startup latency, 1074
Logic Wall, 511
LogP Model
  definition, 260
  nonoverlappable overhead, 261
  overlappable latency, 261
  processor overhead, 260
Logs
  observability, 363
Lookahead Decoding, 522
LoRA
  parameter efficiency, 602
Loss Spike, 342
Lossless Delivery, 110
Low Likelihood/High Impact, 755
Low-Rank Adapter, 602
LRU
  ML failure mode, 143
Lustre
  etymology, 155

M/G/c/K Queues, 1091
MAC, 71
Machine Learning Fleet
  definition, 7
Machine Unlearning
  gradient ascent, 968
  model update strategies, 968
  privacy cost, 965
Mahalanobis Distance
  anomaly detection, 856
Maintenance Overhead, 716
MAML
  few-shot adaptation, 622
Manufacturing Variances, 310
Mapping
  hierarchy, 237
MapReduce, 10
Marginal Cost, 707
Marginal vs. Average Emissions, 923
Mars Polar Lander
  software failure, 827
Masked Language Modeling, 859
Masking Effect, 332
Materialization, 722
Matrix Multiply Unit
  TPU, 61
Maximum Mean Discrepancy
  drift detection, 832
MCUNet, 920
MDS, 155
Mechanistic Interpretability
  research field, 974
Medical Device
  ML security, 779
Medical Device Security, 779
Medical Efficacy, 983
Medusa, 522
Megatron-LM, 12
  tensor parallelism, 230
Meltdown/Spectre
  ML impact, 774
Membership Inference
  privacy, 755
  privacy attack, 950
  privacy risk, 624
Membership Inference Attack, 624, 755, 950
Memory
  bit flips, 309
  leak, gpu training, 328
  paged, 497
  persistent, 177
  timeline, 467
Memory Access, 525
Memory Allocation
  contiguous, 517
Memory Analysis

layer-wise, 920
Memory Bandwidth, 590
protection analysis, 320
Memory Exhaustion, 195
Memory Footprint, 590
Memory Hierarchy
energy costs, 880
Memory Layout, 518
Memory Wall, 8, 49, 1010
data parallelism, 205
dominance, 100
Memory-Bound, 440
optimization path, 473
Memory-boundedness, 56
Metadata Server, 155
Metric
observability, 363
METUX Model
user experience, 999
MFCC
audio compression, 611
MFU
sustained vs. peak, 101
Microcontroller
power budget, 639
STM32F4, 593
MIG
A100 profiles, 388
GPU isolation, 388
orchestration, 388
partitioning, 554
Minimal Attack Surface, 757
Minimax
adversarial training, 858
Minimum Replica Count, 412
Minimum Viable Scale, 401
Minimum Write Time, 337
Mirai Botnet
scale, 755
Mixed-Precision
numerical fault tolerance, 361
numerical issues, 361
Mixed-Precision Quantization, 456
Mixed-Precision Training, 70
edge constraint, 595
Mixture-of-Experts, *see* MoE
ML API
attack surface, 760
ML CO2 Impact, 927
ML FinOps, 406
ML Framework, 9
ML Networking
data-center workload inversion, 106
ML Platform
cost breakdown, 712
ML Productivity Goodput, 23, 710
MLCommons, 927
MLOps
responsible AI tension, 993
MLPerf, 472
MLPerf Tiny
benchmark suite, 921
Mobile ML, 958
MobileNet
efficiency, 592
lighthouse archetype, 28
v2 storage footprint, 592
Mode Collapse, 789
Model Architecture Selection, 721
Model Bandwidth Utilization, 466
Model Card, 952
documentation, 952
Model Collapse
definition, 182
Model Decay, 734
Model Development, 25
Model Distillation
theft, 762
Model Extraction
definition, 759
preview, 22
Model FLOPs Utilization, 64
distributed training, 241
fleet performance, 472
fleet-scale behavior, 471
local node baseline, 472
Model Interpretability Spectrum, 974
Model Inversion
federated attack, 623
privacy, 761
Model Inversion Attack, 623, 761
Model Loading
isolation, 554
latency, 177
Model Memorization
diffusion model memorization, 966
privacy risk, 949
Model Metrics, 727
Model Parallelism, 6
definition, 223
Model Registry
schema, 672
Model Replication, 73
Model Repository, 760
supply chain, 760
Model Rollback, 794
Model Rollout
storage bandwidth, 144
Model Serialization
security, 761
Model Size
scaling, 719
Model State Size, 337
Model Synchronization, 564
Model Theft
strategies, 761
Model Warm-Up, 546
Model Watermarking
IP protection, 790
Model Weights, 84
Model-Type Monitoring Parameters, 703
Model-Type Operational Requirements, 658
MoE
communication, 258
forward reference, 104
infrastructure challenge, 68
performance engineering, 461
routing, 233
serving, 533
Moments Accountant, 811
Monitoring
fleet, 697
network, 131
Monte Carlo Dropout, 854, 855
Montréal Carbon Pledge, 927
Moore's Law
sustainability limits, 868
MPI
legacy, 289
MPS
GPU sharing, 414
MTBF, 94
checkpoint frequency, 320
GPU field variation, 304
system, 4
MTBF cascade, 1084
Multi-Tenancy
cost efficiency, 549
tradeoffs, 550
Multi-Worker Data Loading, 168
Multicalibration
intersectional fairness, 961
Multiply-Accumulate, 872
Mutual TLS
ML serving, 791

Namespace Isolation, 157
NaN Detection, 361
NaN Propagation
training corruption, 361
NCCL
debug logging, 736
debug logs, 132
debugging, 736
performance, 289
timeout, 342
timeout trade-off, 342
topology, 198
Near-Memory Computing
edge training, 596
Netflix
deanonymization, privacy, 761
Network Contention, 265
Network Fabric, 11
Network Packet Loss, 309
Network Partition
distributed training, 322
fault tolerance, 309
federated learning, 624
Network Requirements, 73
Network Slices, 131
Network Switch Failure, 313
Network Topology
definition, 116
Network Virtualization, 130
Network Wall, 4
Network-bound workload, 1062
Networking Between Instances, 667
Neural Architecture Search, 732
carbon cost, 892
Neural Engine
on-device training, 594
Neuromorphic Computing
definition, 927
NHTSA
cybersecurity guidance, 754
Node, 74
crash, 309
definition, 75
Noisy Neighbor, 550
GPU multi-tenancy, 711
isolation, 388
Noisy Neighbor Problem, 157, 711
Non-Blocking Collective, 290
Non-blocking Fabric
definition, 117
Non-IID
federated learning, 588
Non-von Neumann Architecture, 927
Normative Pluralism, 984
Nsight Compute, 464
Nsight Systems, 467
NUMA
aware allocation, 151
GPU data loading, 79
inference latency, 415
Numerical Instability, 310
NVIDIA
H200, memory wall, 56
NVLink, 86
distributed training, 198
topology-aware scheduling, 393
NVMe
ML storage tier, 151
offloading, 84
queue parallelism, 151
NVSwitch, 80
NVSwitch Crossbar, 74

OAuth
ML API authentication, 791
Object Classification Stability, 826
Object Storage
erasure coding, 158
Object Storage Server, 155
Observability, 697
Observability Stack, 390
Observed Behavior, 338
Offensive ML
use cases, 782
Offline Store, 722
On-Demand Capacity, 405
On-Demand Instances, 667
On-Demand Staging, 176
On-Device Adaptation, 583, 587
On-Device Learning
battery budget, 908
challenge, 642
constraint-solution mapping, 598
definition, 583
trade-offs, 611
training cost, 908
On-Device Prediction Strategy, 586
Once-for-All Networks, 920
One-Sided Communication, 290

oneCCL, 291
Online Softmax
  FlashAttention, 447
Online Store, 722
ONNX Runtime
  mobile, 636
  portable inference, 636
OOM, 209
Operation-Level Timing, 361
Operational Complexity Growth at Scale, 649
Operational Expenditure, 661
Operational Simplicity, 549
Operator Error, 313
Operator Fusion, 445, 920
  element-wise, 446
  operator-specific, 446
  reduction, 446
OpEx, 661
Opportunistic Scheduling, 596
Optical Interconnect, 110
  co-packaged optics, 134
  energy per bit, 1014
Optimal Regime
  Chinchilla frontier, 15
Optimization
  bilevel, poisoning, 767
  capacity, 517
  ctr, proxy misalignment, 997
Optimizer
  memory energy cost, 875
  state lag, 368
Orca
  iteration-level scheduling, 500
Orchestration
  layer, 1014
Organizational Pattern Decision Factors, 729
Oscillation, 413
OSS, 155
  PFS disambiguation, 155
Out-of-Memory, 209
Outlier Feature
  quantization challenge, 450
Output Guardrail, 843
Output Monitoring, 792
Over-Selection
  straggler mitigation, 619
Over-Specification, 424
Over-Subscription, 380
Oxide Breakdown
  scaling reliability, 322

P99, 125
Packet Spraying
  load balancing, 128
Page Table Per Sequence, 518
Paged Memory Management, 492
PagedAttention, 443
  block table, 517
  definition, 517
  memory management, 443
  throughput impact, 519
PAM4, 108
PAM4 Signaling
  definition, 108
Parallel File System
  bandwidth aggregation, 154
  definition, 154
  ML infrastructure, 154
Parallel Systems, 306
Parallelism
  communication costs, 195
  infrastructure interaction, 213
Parameter Server
  architecture, 232
  bottleneck, 255
Parameter-Efficient Fine-Tuning, 909
Pareto Frontier, 24, 1096
Parity Bit Error Detection, 324
Parquet, 161
Participation Bias
  federated learning, 619
Patient Autonomy, 983
Paved Road, 732
Payback Period, 656
PCIe Gen5
  node hierarchy, 79
PCIe Root Complex, 79
Peak FLOP/s, 58
Penetration Testing
  ML limitations, 783
Pentium FDIV Bug
  error regions, 322
Per-Rank Memory Tracking, 737
Percentage Change, 705
Percentage Change Threshold, 705
Perception Gap
  adversarial vulnerability, 844
Performance-Aware Topology Monitoring, 398
Perftest
  point-to-point bandwidth tests, 132
Peripheral Isolation, 884
Permanent Fault, 320
Persistent Collective, 290
Personalization
  trade-offs, 622
Personalization Layer, 622
Personalized Federated Learning, 621
Perspective API
  evasion, 768
  poisoning, 768
PFC Storm, 135
Phased deployment
  data centers, 670
Phishing
  AI-generated, 759
Physical Layer
  validation, 132
Physical Monitoring, 131
Physical Unclonable Function, 802
  arbiter PUF, 803
  device authentication, 802
Pinned Memory
  DMA transfer, 150
Pipeline
  balance, 63
  bubble fraction, 1079
  bubble time, 1079
  ci/cd, 328
  parameterization, 678
  stages, 509
Pipeline Bubble
  utilization trade-off, 226
Pipeline Parallelism, 6, 129, 258
  definition, 227
Pipeline Stress Test, 164
Pipelined Execution, 262
Pipelining
  data loading, 165
Placement Group
  topology, 375
  topology-aware scheduling, 375
Platform Engineering, 708
Pluggable Optics, 110
  power cost, 134
Pod, 92, 118
PodGroup, 388
Point-in-Time Correctness
  feature stores, 169
Point-in-Time Join, 724
Poisson Process
  failure modeling, 304
Policy Model, 242
Pollux, 408
Pool Fragmentation, 424
Population Stability Index
  drift detection, 702, 837
  proxy metric, 703
PortRcvData, 131
PortXmitData, 131
PortXmitDiscards, 131
Positive Predictive Value, 945
Post-Hoc Methods, 973
Post-Training Quantization, 452
Postprocessing Methods
  fairness, 961
Power alley
  site selection, 669
Power Consumption
  jump, 778
Power Delivery, 88
Power Density
  crisis, 1071
  rack, 1067
Power Distribution
  DC distribution, 896
Power Distribution Unit, 896
Power of Two Choices
  routing, 539
Power Profile, 777
Power Ramp, 898
Power Supply Failure, 313
Power Traces, 783
Power Usage Effectiveness
  definition, 877
Power Wall, 64
Power-Law Relationship, 18
Practitioner Decision Framework, 995
Preallocated Memory Management, 492
Precision, 570
  BF16, 70
  FP8, 70
Predicate Pushdown, 161
Predictive Diagnostics, 698
Predictive Scaling, 412, 558
Preemptible Instances, 667
Preemptible VMs, 403
Preemption and Swapping, 501
Preemption Budgets, 421
Preemption Cascade, 420
Preemption Tax, 420
Preemptive Scheduling, 493
Prefetch Buffer, 150
  depth calculation, 167
Prefetching, 87, 267
Prefill
  inference phase, 73
Prefill and Decode
  definition, 524
Prefill Phase, 443, 524
Prefill Pool, 526
Prefix Caching, 354
  hit rate, 520
  memory savings, 520
Preprocessing Methods
  fairness, 961
Prestaging, 176
Prewarming, 412
Price Sensitivity Model, 676
Primary Model, 356
Principle of Least Privilege, 856
Priority Flow Control, 125
  definition, 125
Priority Inversion
  active blockage, 383
  definition, 383
Priority Preemption, 507
Priority-Aware Scheduling, 493
Privacy
  accuracy trade-offs, 812
  definition, 750
  preserving machine learning pipeline, 950
  protection, 983
  utility tension, 812
  utility trade-off, 786
  utility, trade-off, 812
Privacy Loss Random Variable, 811
Proactive Path, 653
Proactive Replacement, 345
Process Group, 258
Product Embedding Model, 676
Production Serving, 418
Production Training, 418
Profiling
  distributed profiling, 737
Progress Monitoring, 383
Projected Gradient Descent Attack, 846
  adversarial training, 971
Prometheus
  monitoring, 708
Prompt Caching, 354
Prompt Injection, 843
  LLM security, 795
Proportional Allocation, 707

ProtoPNet
interpretability, 974
Provenance Chain
definition, 182
Provisioning Granularity, 667
Proximal Policy Optimization
model asymmetry, 242
Proxy Metrics
Goodhart's Law, 996
ProxylessNAS, 920
Pruning
energy impact, 919
PUE, 1067
efficiency metric, 900
evolution, 871
facility load, 1067
financial savings, 877
gap, 922
hyperscale efficiency gap, 922
IT load, 1067
ML optimization, 878
overhead reduction, 877
PyTorch Profiler, 466
PyTorchFI, 332

QoS
edge training, 639
Qualcomm
Trepn Profiler, 883
Quality of Service, 131
priority traffic, 128
Quantization, 70, 571
aware training, edge power, 594
block-wise, 450
energy savings, 919
Hessian matrix, 567
per-tensor, 450
quantization-aware training, 567
rotation-based, 569
Quantization-Aware Training, 452
QuaRot, 569
Hadamard rotation, 569
Queries Per Joule, 921
Quota Governance, 421
Quota Review Cycle, 421

Race Condition
distributed training, 329
Rack, 88
definition, 88
Rack-level DC power distribution, 90
Radix, 117
RAID
ML storage, 141
Rail-Optimized Topology, 397
definition, 120
Random-Access IOPS, 181
Random-k Sparsification, 284
Randomized Smoothing, 853
certified defense, 971
Ranking Head, 507
Ranking Model, 676
RAPL Energy Measurement, 883
Rawput, 1088
RBAC
ML pipeline, 791
RCCL, 291
RDMA
definition, 111
Reactive Scaling, 414
Readiness Probe, 342
Rebound Effect, 917
Recall vs. Latency vs. Capacity, 181
Recommendation Model, 68
Recommendation Sessions, 542
Recommendation System, 232, 266, 495
freshness fault tolerance, 317
inference demand, 488
Recourse, 985
Recovery Time
scale impact, 343
Recovery Time Equals Checkpoint Time, 335
RecSys Freshness, 317
Recursive Halving-Doubling, 273
Red Team
ML security, 783
Red Team Exercises, 783
Red Teaming, 704
etymology, 986
Reduce, 265
ReduceScatter, 265
Redundancy
fault tolerance model, 1062
Redundant Replicas, 350
Reference Model, 242
Register File, 438
accelerator hierarchy, 49
bandwidth, 49
register spilling, 438
Registration, 677
Regression Test Automation, 328
Reinforcement Learning from Human Feedback
alignment technique, 998
multi-model coordination, 241
Reliability
engineering, ml extension, 829
Reliability Gap
definition, 19
Remote Attestation
trusted execution environments, 797
Remote Direct Memory Access, *see* RDMA
Rendezvous
elastic training, 343, 349
Replay Buffer, 609
Replication
active-active replication, 351
active-passive replication, 351
Representativeness Gap, 978
Request Classification, 506
Reserved Instances, 667
Residual Adapter
parameter efficient finetuning, 601
Resilience, 20
ResNet-50
standard, 853
Resource Contention, 550
Resource Quotas, 551
Resource Reclamation, 343
Resource Utilization, 549
Responsibility-Sensitive Safety
framework, 999
Responsible AI
at scale, 22
definition, 935
lifecycle, 937
measurement baselines, 959
performance impact, 959
Retraining
frequency, 721
Retrieval Fee, 174
Retrieval Noise, 843
Retrieval-Augmented Generation, 715
alignment, 978
Reward Hacking
value misalignment, 997
Reward Hacking Loop, 997
Reward Model, 242
Ridge Point
definition, 58
Right-to-Repair, 916
Rigid Scheduling, 399
Ring, 74
Ring algorithm
bandwidth optimal, 1075
Ring AllReduce, 25, 115
AllGather phase, 269
bandwidth, 276
bandwidth optimality, 269
data movement per GPU, 271
latency, 276
origin, 255
scatter-reduce phase, 269
sequential steps, 271
Ring Attention, 232, 449
Risk Mitigation, 682
Risk-Based Rollout Strategy Selection, 695
Robotics, 509
Robust AI
definition, 823
Robustness Margin, 828
Robustness-Privacy
trade-off, 785
Robustness-Privacy Trade-Off, 785
RoCE, 112
ROI Ratio, 651
Rollback
automated configuration, 696
Rolling Maintenance Window, 698
Roofline Model, 20, 57, 1068
origin, 57
performance engineering, 439
ridge point, 439, 1068
Round-Robin Routing, 416
Router
Mixture of Experts, 461
Routing
least-outstanding-requests, 416
memory-aware, 416
Routing Affinity, 542
Routing Instability, 533
Row-Oriented Format, 161
Rényi Differential Privacy, 813

S-box, 784
Safety
critical applications, 822
critical systems, ml deployment, 822
Saliency Map
explainability, 954
Sampled Least-Connections, 541
Samples Per Joule, 921
Samples Required, 703
Savings
annual, 649
net, 654
weekly, 656
Scale
qualitative effects, 1015
Scale Moment, 4
failure certainty, 4
output-to-impact shift, 4
processor-to-network shift, 4
Scaling
batch size, 555
breakdown types, 18
dimension tradeoffs, 555
event-driven, 559
horizontal, 555
metric-based, 556
prewarming, 557
regression, 472
user, 719
Scaling Cliff, 212
Scaling Efficiency, 24, 401
definition, 211
elastic scheduling, 399
problem size dependence, 211
Scaling Laws, 12
compute optimality, 14
loss curves, 15
model, 871
saturation, 18
Scaling Tax
communication overhead, 213
Scaling Wall
definition, 215
Scan Chain
manufacturing test, 326
Scheduled Burn-In Testing, 698
Scheduling
ML-specific policies, 429
topology awareness, 429
Scheduling Policy
infrastructure coupling, 429
Scope 1 Emissions, 892
Scope 2 Emissions, 893
Scope 3 Emissions, 893, 926
Secondary Model, 356
Secure Aggregation, 624
federated privacy, 624
Secure Boot, 799

open-source stacks, 801
Secure Boot Sequence, 800
Secure Enclave
architecture, 798
Secure Multiparty Computation, 788
collaborative training, 788
performance overhead, 788
Security
boundaries, 550
definition, 749
privacy distinctions, 752
Selective Layer Updating, 605
Self-Consistency, 842
Self-Determination Theory
autonomy, 999
Self-Speculative Decoding, 522
Self-Supervised Learning
robustness, 859
sensor data, 627
Semantic Caching
trade-off, 565
Semantic Saturation, 18
Semiconductor Fabrication
water consumption, 911
Semiconductor Water Scale, 911
Sensitivity
global sensitivity, 809
Sequential execution
D·A·M coordination, 1049
Sequential Read Bandwidth, 181
SerDes, 110
definition, 110
signaling, 78
Serialization
zero-copy, 486
Serializer/Deserializer, 110
Series Systems, 306
Serving
cost metric, 73
hierarchy, 489
replica level, 489
Serving Cost
annual serving cost, 487
cost per request, 487
Serving Cost Dominance Law, 487
Session Affinity, 353
loss, 565
SEU
error rate, 309
SGD
microcontroller constraint, 594
shuffle requirement, 143
SHA-256, 794
Shadow Deployment, 686
Shadow Model
evaluation, 632
Shapley Value
explainability, 973
Shard Contention
collision probability, 141
Sharded Checkpointing, 341
I/O parallelism, 341
Sharded Data Parallelism
definition, 208
Shared Memory
accelerator SRAM, 49
SHARP
in-network reduction, 279
Showback, 406
Side-Channel Attack
ML, 762
Side-Channel Attack Vulnerability, 778
Silent Data Corruption, 159
mechanism, 310
ML systems, 824
training detection, 309
Silent Degradation, 132
Silent Failure Mode, 822
Silent Waste, 131
Silicon Lottery
fleet consistency, 47
Simple Composition, 813
SIMT
definition, 41
warp divergence, 41
Single Failure Mode, 335
Single-Model vs. Platform Operations, 656
Site Reliability Engineering
ML extension, 738
Six Systems Engineering Principles, 26
SLA
requirement, 338
SLO
complexity, 550
constrained optimum, 1096
reliability target, 686
SLO-Driven Scaling, 413
Slurm
imperative model, 384
partition configuration, 385
scheduling, 384
vs. Kubernetes, 390
workload manager, 384
Slurm-alongside-Kubernetes, 391
SM Partitioning, 463
Small File Problem
definition, 155
Small-Scale Canaries, 472
SmoothQuant, 451, 569
Sociotechnical Dynamics
definition, 935
Sociotechnical System
etymology, 935
Software Bugs, 313
Software Fault
mitigation strategies, 328
Software Timeouts, 309
Software-Based Fault Injection, 332
Span, 360
Sparse Feature Lookup, 507
Sparse Pull, 233
Sparse Push, 233
Sparsification, 284
Specification Gaming
objective design, 998
Spectral Signature, 849
data sanitization, 858
Speculative Decoding
bandwidth optimization, 460
speculation tax, 522
Speculative Execution
side channel, 774
Speech Activity Detector, 792
Speed of Light
latency bound, 255
Spiking Neural Network, 872
energy efficiency, 872
Spine Switch, 393
Spot Instance Economics, 562
Spot Instances, 667
cost optimization, 403
inference, 562
termination handling, 562
Spurious Correlation, 837
Square Root Scaling Rule
batch size, 217
SR-IOV
definition, 130
SRAM, 438
energy efficiency, 49
SRAM PUF, 803
SSD Endurance, 154
Stability
responsible AI, 934
Stability-Plasticity Trade-Off, 628
Staffing
annual staffing, 664
Staged Ensemble Deployment Pattern, 674
Stale Synchronous Parallel
bounded staleness, 203
Staleness Penalty, 217
State Reconciliation, 626
State Synchronization, 343
Stateful Serving
KV cache migration, 370
Static Batching, 491
Static Fallback, 356
Static Partitioning, 416
Static Placement, 168
Static Power, 876
Static Power Waste, 905
Static Rail Alignment, 699
Statistical Efficiency
distributed training, 216
Statistical Multiplexing, 549
Statistical Parity Difference
definition, 945
Statistical Process Control
ML monitoring, 701
Statistical Profiling, 471
STM32F4
compute constraints, 593
Storage
distributed, 11
economics, 173
failure, 309
savings, 599
system, 337
system stress test, 333
total data movement, 181
workload inversion, 141
Storage Hierarchy
extended memory hierarchy, 146
Storage Refresh Cycle, 175
Straggler, 8
definition, 345
federated learning, 626
problem, 626
step time, 346
synchronization delay, 203
Straight-Through Estimator
quantization, 452
Stranded Resources, 424
Streaming
data staging, 176
Streaming Adaptation, 607
Streaming Format
sequential, 161
Strict Synchronous, 340
Stripe Count, 155
Stripe Size, 155
Striping, 156
Structured Logging
JSON log entry, 359
Stuck-at Fault Model, 322
Stuxnet, 753
Subnet Manager, 111
reliability, 322
Subpopulation Attack, 767
Subpopulation Poisoning, 849
Supply Chain, 666
Sustainability, 864
Sustainability Paradox, 864
Sustainable AI
definition, 864
efficient hardware selection, 925
measurement first, 925
model optimization, 925
Switch Radix
port density, 118
Switching Activity Factor, 876
Symbol Rate, 108
SymbolErrorCounter, 131
Synchronous Checkpointing, 338
Synchronous Tight Coupling, 8, 10
Synchronous Transients, 897
Synthetic Data
privacy trade-off, 789
provenance overhead, 182
raw payload, 182
verification factor, 182
verified data, 182
Synthetic Data Generation, 788
System Integrity Checks, 793
System Jitter, 345
System Prompt
governance, 978
Systemic Vulnerability
ML pipelines, 823
Systolic Array
definition, 42
etymology, 42

Tail Latency, 125, 141
Taints and Tolerations
GPU isolation, 398

Kubernetes scheduling, 398
Targeted Attack, 767, 852
Targeted Profiling, 470
TCO
financial framework, 661
TCO Framework, 715
TDP, 64
thermal constraint, 64
Telemetry
in ML systems, 957
ML monitoring, 957
paradigms at scale, 657
Tensor Arena
optimization, 920
Tensor Core
definition, 61, 62
evolution, 61
Tensor Parallelism, 78, 258, 474
definition, 230
parallel group, 416
TensorFlow Lite
mobile inference, 636
TensorRT
kernel selection, 474
Tertiary Model, 356
Test-Time Compute, 511
TF32, 69
TFRecord, 155
TFX
data validation, 708
Themis, 408
Thermal Design Power
definition, 65
Thermal Interface Material, 67
Thermal Stress
training throughput, 324
Thermal Throttling, 309
Threat Model, 754
Threat Modeling
ML systems, 755
Three Pillars Framework, 830
Through-Silicon Via
HBM stacking, 52
Throughput, 377, 436
loss, 751
Tier III data center, 91
Tier IV data center, 91
Tiered Staging
checkpoint storage, 179
Tiered Storage, 87
Tiering Strategy, 175
Tiling, 50, 447
Time Per Output Token, 485
reading speed, 485
Time to First Token, 466, 485
responsiveness, 485
vs. time per output token, 485
Time Travel, 724
Time value of compute, 670
TinyML, 958
extreme optimization
techniques, 919
market scale, 610
scale, 610
TinyTL
memory optimization, 600
memory reduction, 600
Tiresias, 407
TMR
majority voting, 328
Token
vocabulary size trade-off, 13
Token Dropping, 533
Token Generation Rate, 552
Tokens Per Second Per Dollar, 72
Top-k Sparsification, 284
Top-k Truncation, 765
Top-of-Rack switch, 90
failure domain, 117
Topology Detection, 280
Topology Discovery, 397
torch.compile
kernel fusion, 474
TorchDynamo
bytecode tracing, 455
graph capture, 455
TorchInductor, 455
Torus, 122
Torus Topology, 281
Total Cost of Data
delivery, 173
Total Cost of Ownership
break-even, 664
carbon-aware placement, 866
definition, 715
hardware, 661
TPU, 47
inference, 576
systolic array, 5
v5p, 48
Trace, 360
observability, 363
Trackable Resource, 387
Traditional I/O, 170
data path, 171
Traffic Class, 131
Traffic Throttling, 795
Training, 677
carbon intensity, 18
centralized, 613
compute evolution, 5
compute trends, 13
elastic, 346
energy consumption, 865
image, 148
local, 614
resumption, 343
slowdown, 832
text, 148
Training Communication Overhead, 871
Training Cost, 487
Training Data Extraction, 771
Training Duration, 195
Training Operator Ecosystem, 389
Training state rule, 1063
Training Time
total FLOPs, 93
Training-Serving Gap, 721
Training-Serving Skew, 721
platform invariant, 649
Transferability, 846
black-box attack, 846
Transient Fault, 320
mechanism, 321
Transparency, 936
Transport Monitoring, 131
Tree AllReduce, 272
bandwidth, 276
broadcast phase, 272
latency, 276
reduce phase, 272
TRES, 387
Triad of Distributed Efficiency, 22
Trimmed Mean, 313
Triton
kernel programming, 457
Trust
calibration gap, 982
Trusted Execution Environment, 797
ML constraints, 797
TSV
3D stacking, 52
Two-Phase Commit
checkpoint atomicity, 340
Two-Tier Checkpointing, 404
Two-Tower Architecture
serving, 576

Ultra Ethernet, 133
Uncertainty Quantification
generative AI, 842
Unfused Execution, 445
Uniform Parallelism, 237
Uninterruptible Power Supply, 896
Universal Efficiency Fallacy, 29
Universal Scaling Law, 12
Unreliable Substrate, 319
Unstructured Sparsity, 919
Upstream Model Change, 734
USB Attack Vector, 753
Useful Life, 315
User Embedding Model, 676
Utility-Privacy Trade-Off, 750
Utilization
allocated, 419
based reclamation, 421
compute, 419
productive, 419

Validation, 338, 971
performance, 634
Validation Gate
performance, 679
Validation Stage, 344
Value Alignment, 936
AI safety, 936
Value Model, 242
Value-Sensitive Design
methodology, 999
Verbs API, 112
Version Consistency, 564
Vertical Pod Autoscaling, 412
Victim Flows, 125
Video, 509
Virtual Function
definition, 130
Virtual Lane, 111
Vision Models, 495
inference demand, 488
vLLM
etymology, 491
Volcano
Kubernetes scheduling, 388
Von Neumann
bottleneck energy cost, 927
Von Neumann Bottleneck, 927
VPA, 412

Wafer-Scale Engine, 11, 43
Wake-Word Detection
power constraint, 587
Warehouse-Scale Computer, 11
Barroso, 11
definition, 94
Warm Cache, 152
local NVMe cache, 154
remote-read baseline, 153
Warm Replica, 177
Warm Restart, 344
Warm Tier, 175
Warm-Up Period, 884
Warmup
time, 367
Waste Heat Reuse, 902
Waste Ratio, 502
Wasted Compute, 336
Water Usage Effectiveness, 877
Weak scaling
fleet scale, 1070
Wear-Out, 315
WebDataset
streaming format, 156
Weighted Assignment, 541
White-Box Access, 757
White-Box Attack
robustness bound, 846
Whole-Program Compilation, 455
Wireless Upload Asymmetry, 620
Working Set Size, 143
Workload Characterization, 668
Write Pattern, 143, 338
WSC
data center scale, 94

XLA
Accelerated Linear Algebra, 455
XNOR-Net Operations, 920

Young-Daly Checkpoint Law
optimal interval, 249
Young-Daly Formula, 304
history, 304

Z-score, 705
Z-Score Method
anomaly detection, 856

Z-Score Threshold, 705
ZeRO, 12
  definition, 208
  memory optimization, 208
  stage 3, 84
Zero-Bubble Pipeline Parallelism, 229
Zero-Copy
  serialization, 486
Zero-Copy Communication
  RDMA, 256
Zero-Day, 753
Zero-Day Exploit, 753
Zombie Models
  deprecation, 675