CONTEXT ENGINEERING

Designing AI That Understands, Remembers, Acts and Learns



Dr Alexander Mikhalev

Context Engineering

Designing AI That Understands, Remembers, Acts and Learns

DR ALEXANDER MIKHALEV

2026-10-08

Preface

Context Engineering: Designing AI That Understands, Remembers, Acts and Learns argues that reliable software built on large language models requires deliberately engineered context. The book is written for senior developers, engineering managers and architects who have shipped at least one LLM-backed system and found that it behaved differently in production than in the demo.

Every chapter ends with its sources. Chapters 14 and 20 report measurements from the author's own tooling, with source status and limitations so that readers can assess their applicability.

© 2026 Dr Alexander Mikhalev. All rights reserved. Corrections from readers are welcome and will be acknowledged in future editions.

Dr Alexander Mikhalev, October 2026

Acknowledgements

Thank you to my business partners at Zestic AI for your collaboration and support throughout the work behind this book.

I also thank everyone who contributed through Telegram and other channels. Your questions, feedback and conversations have helped me think more carefully about the ideas presented here.

Thank you for your time and generosity.

Introduction

Across the dozen or so engineering organisations the author worked with or advised in 2023 and 2024, something odd happened; this is the author's observation, not a survey. Teams that had spent decades building disciplined practices, statistical process control, test-driven development, continuous integration, looked at their new AI systems and said, in effect, "Let's just see what the prompt does."

Some systems that worked in development failed in production; others revealed failures that testing had missed. A recurring problem was **conceptual failure**: the system's working picture of the domain, the user and the task no longer matched reality, and the system had no check for the mismatch.

This book describes how to detect those mismatches, limit their consequences and revise the system on evidence.

An operational assistant proposes a change that appears to fix yesterday's failed job. The service owner asks whether it can be used tonight. The team needs an agreed meaning of success, evidence that the proposal preserves working behaviour and authority to apply this particular revision. After the job runs, it needs to check what actually happened. Another team's jobs may share the same execution quota: a repair that adds retries could affect people outside the original request.

This maintenance example is synthetic. Its cases, requirements and outcomes are teaching fixtures, not measurements of a deployed assistant. It recurs throughout the book alongside the author's systems, internal evaluations and composite cases, whose evidential status remains separate.

Why This Book, Why Now

In software built around large language models, behaviour depends on prompts, knowledge bases, retrieval systems and the model as well as application code. Engineers must manage changes to each of these inputs to keep the system reliable.

Language models can write code, analyse long documents and work through multi-step problems. The author's experience and measurements appear throughout the book, identified as such so that readers can distinguish them from published research.

In the organisations described above, teams deployed systems before establishing how to version their context or validate their outputs. Prompt changes became a substitute for investigating failures.

The public incidents and first-hand cases discussed in this book show why production systems need checks for failures that conventional code tests can miss.

This book presents that discipline. The author calls it **Context Engineering**.

What Is Context Engineering

Context Engineering is the discipline of making AI systems reliable by explicitly managing the context they operate on.

The argument developed here is: **context engineering connects stakeholder concerns and intended use to explicit requirements, the services that fulfil them, and evidence of actual behaviour as the system changes.** Stakeholders can have conflicting concerns. Some consequences affect people or systems absent from the initial stakeholder list. Establishing what matters, what is agreed and what remains uncertain is part of the engineering work.

Requirements are allocated to services and realised by solution elements. A service here is offered behaviour, not necessarily a separately deployed component. Context supplies relevant meanings, instructions and evidence; the runtime enforces the applicable permissions and operating limits. A description of a control is insufficient until the responsible mechanism applies it.

Two questions recur: **what is the current basis for this use?** And **what may change, on what evidence and with whose authority?** New observations can challenge the intended use or a requirement as well as the implementation. The team develops use claims and solution concepts together, maintaining traceability through authorised revisions. Agents may evolve only within enforced boundaries consistent with the organisation’s approved risk appetite; they cannot enlarge their own authority or weaken those controls.

Prompt engineering focuses on optimising the input prompt. Context Engineering also covers knowledge bases, retrieval systems, validation gates, token budgets and governance.

Treating **context as infrastructure** means versioning its sources, testing changes and monitoring their effects. Incorrect context can produce incorrect answers even when the application code works as intended.

Turkhanov of Zestic AI described this framework in 2025 as “the missing discipline of software architecture.” It adapts engineering control from the 1960s and domain-driven design from the 2000s to AI development.

What You Will Learn

The six reading groups follow the engineering questions below. Chapter numbering is retained; the groups describe the argument rather than a mandatory project lifecycle.

Reading group	Question	Chapters and contribution
I. Establish concerns, use and meaning	What problem matters, who may be affected and what is agreed?	1–4: failure modes, context ownership, engineering discipline and requirements allocated to services.
II. Build reviewed knowledge	What can the assistant retrieve and check?	5–9: graphs, retrieval choices, an inspectable implementation, curation and grounding limits.

Reading group	Question	Chapters and contribution
III. Select context within a budget	What belongs in this decision, and what can be omitted?	10–12: allocation, loss-aware compression and evaluated capability selection.
IV. Assess the proposal and control action	What has been checked, who may permit it and what must survive handoff?	13–19: project rules, validation, harnesses, policy, traces, delegation and runtime hooks.
V. Observe operation and respond	Does the composed service work under its actual conditions?	20–23: an operational example, enterprise governance, hybrid retrieval and maintained engineering practice.
VI. Choose components and reopen claims	Which components fit, and where is more evidence needed?	24–25: technology selection and testable research questions.

Reading the Evaluation Expansion

After Chapter 14 defines what must be checked and why, Chapter 15 constructs an inspectable assessment, Chapter 16 separates that assessment from permission to act, and Chapter 17 retains identity, evidence and outcome links. A favourable verdict cannot perform all four jobs. Each numbered chapter includes a written recap and a recap diagram. Appendix H collects the synthetic maintenance example's reference templates and requirement-to-evidence links.

The example gradually acquires a **decision packet**: the concern and intended use, requirement and service, selected context and versions, candidate, assessment, authority and observed outcome. This is a teaching record that can refer to separate stores. Chapter 10 deliberately loads only the portion needed for the current decision, with references to the rest.

How to Read This Book

Figure 1 shows the reading groups and their handoffs. Figure 2 shows how the mechanisms contribute to a decision and its later review.

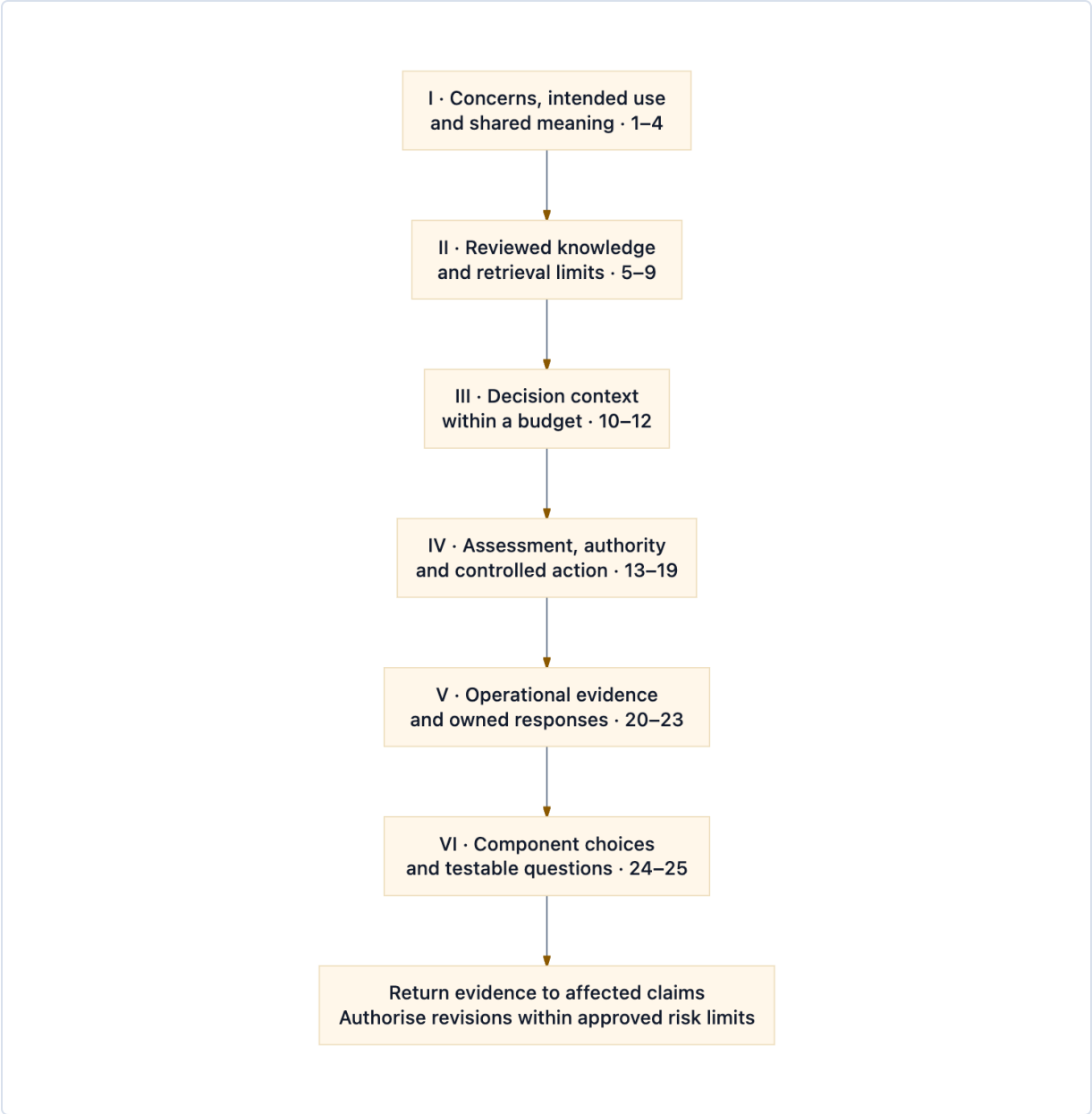


Figure 1: The six reading groups connect concerns and intended use to reviewed knowledge, bounded context, controlled action, operational evidence and renewed inquiry.

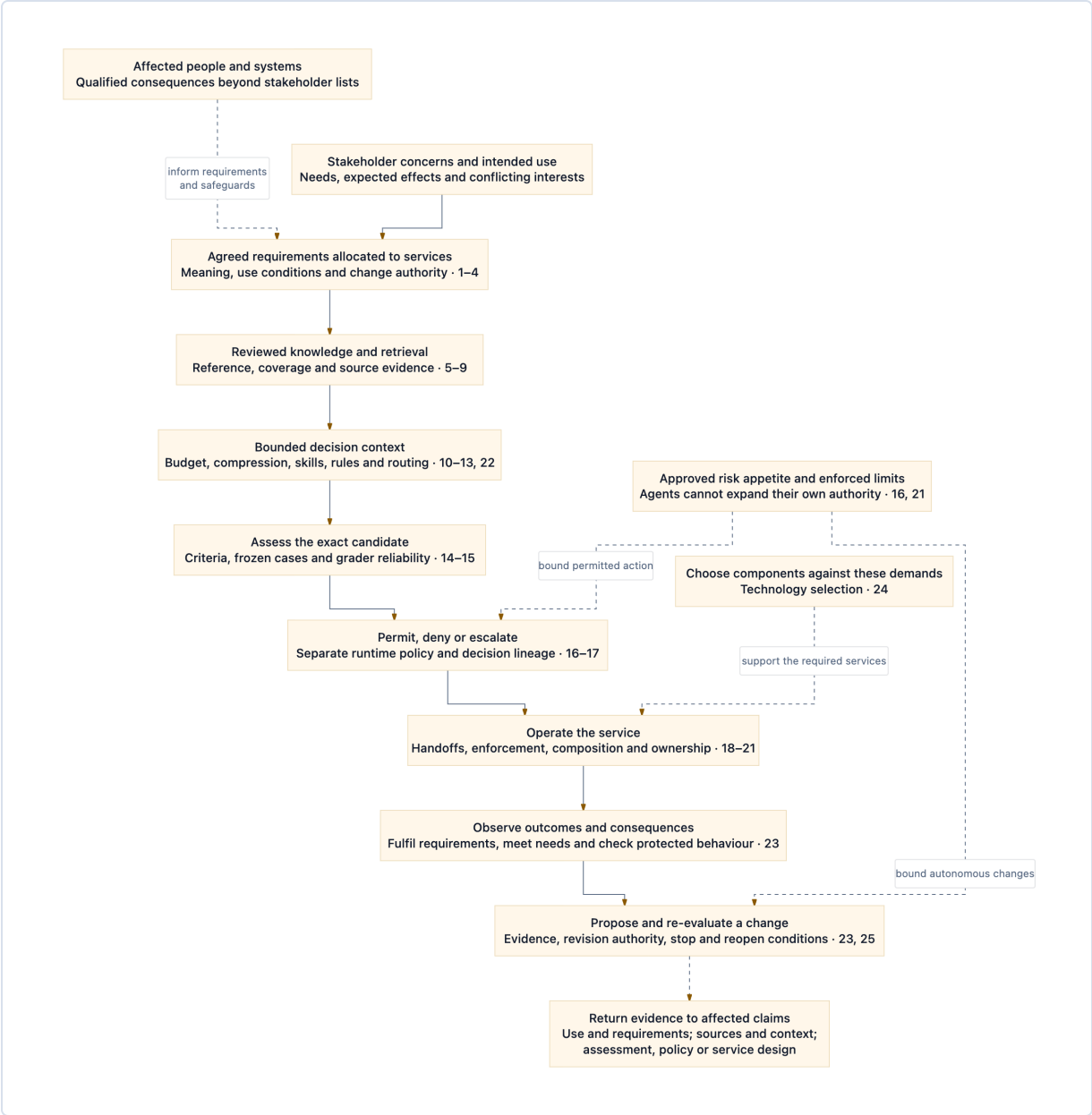


Figure 2: Conceptual links across the book: concerns and intended use inform requirements and services; knowledge, assessment and policy support operation; observed consequences return to the affected claims. Chapter references use the current numbering.

Solid arrows show the teaching progression. Dotted arrows identify supporting relations and the return of evidence for review. The diagram is not a universal sequence: use and solution choices develop together, and a changed claim can reopen earlier work.

Read the book in order for the full argument, or use individual chapters to investigate a specific problem.

If you are new to AI software development, start from the beginning. The foundational chapters define context, domain models and the failure modes used throughout the book.

If you are already building AI systems and encountering specific problems, you can jump to relevant chapters. The token budget chapter (10) addresses context limits. The knowledge graph chapters (5-9) examine retrieval quality. The governance chapter (21) covers controls for deployments in regulated environments.

If you are a technical architect or engineering leader, pay special attention to reading groups IV and V. For retrieval design, read Chapter 22 alongside Chapters 6–9 and 12. Read Chapter 24's selection criteria against the requirements and operating constraints established earlier.

One exercise before Chapter 1. Take the last incident your AI system caused and ask whether the model was wrong or the context was: did it have the fact it needed, was the fact current, was it told what mattered? In the author's incidents the answer is usually the context, and the rest of the book is organised around making that answer checkable.

A Note on Sources

This book draws from several kinds of source, listed in full at the end of this introduction and of every chapter:

- Zestic AI's "Context Engineering: The Missing Discipline" (Turkhanov, 2025) supplies the book's framework.
- **Real systems**: OpenClaw (the assistant system the author maintains) and Terraphim (the author's knowledge graph platform); client deployments are not described
- **Case studies**: The MedGemma competition entry, where knowledge graph grounding caught a tenfold dosage error in the author's evaluation (Chapter 9), and composite enterprise cases (Chapter 1)
- **Research**: Microsoft GraphRAG, Armin Ronacher's "Agent Design Is Still Hard," academic literature on knowledge graphs, vector embeddings, and AI safety
- **Practitioner experience**: Lessons learned from building AI systems in production
- **Framework distinctions**: Selected FPF and Systems Engineering DPF passages inform the separation of concerns, intended use, consequences and evidence. This book adapts those distinctions; it does not claim FPF conformance or introduce its pattern vocabulary as a prerequisite.

Where possible, the book cites primary sources. Where it draws on the author's experience, it says so and tries to make the underlying principle explicit so you can test it in your own context.

Scope of the Framework

The framework in this book comes from the author's practice. Chapter 25 identifies its open problems and the evidence still needed to evaluate proposed solutions.

Chapter 1 starts with the failure modes that these methods are intended to catch.

Summary

The continuing question is whether the service fulfils agreed requirements under current conditions, with acceptable consequences, and what evidence justifies changing it. Stakeholder concerns and intended use inform those requirements. Reviewed knowledge and bounded context support a proposal; assessment, permission and execution remain separate. Observations can reopen the use, requirement or solution through the applicable authority. The examples distinguish teaching fixtures, internal evaluations and practitioner experience from independently established results.

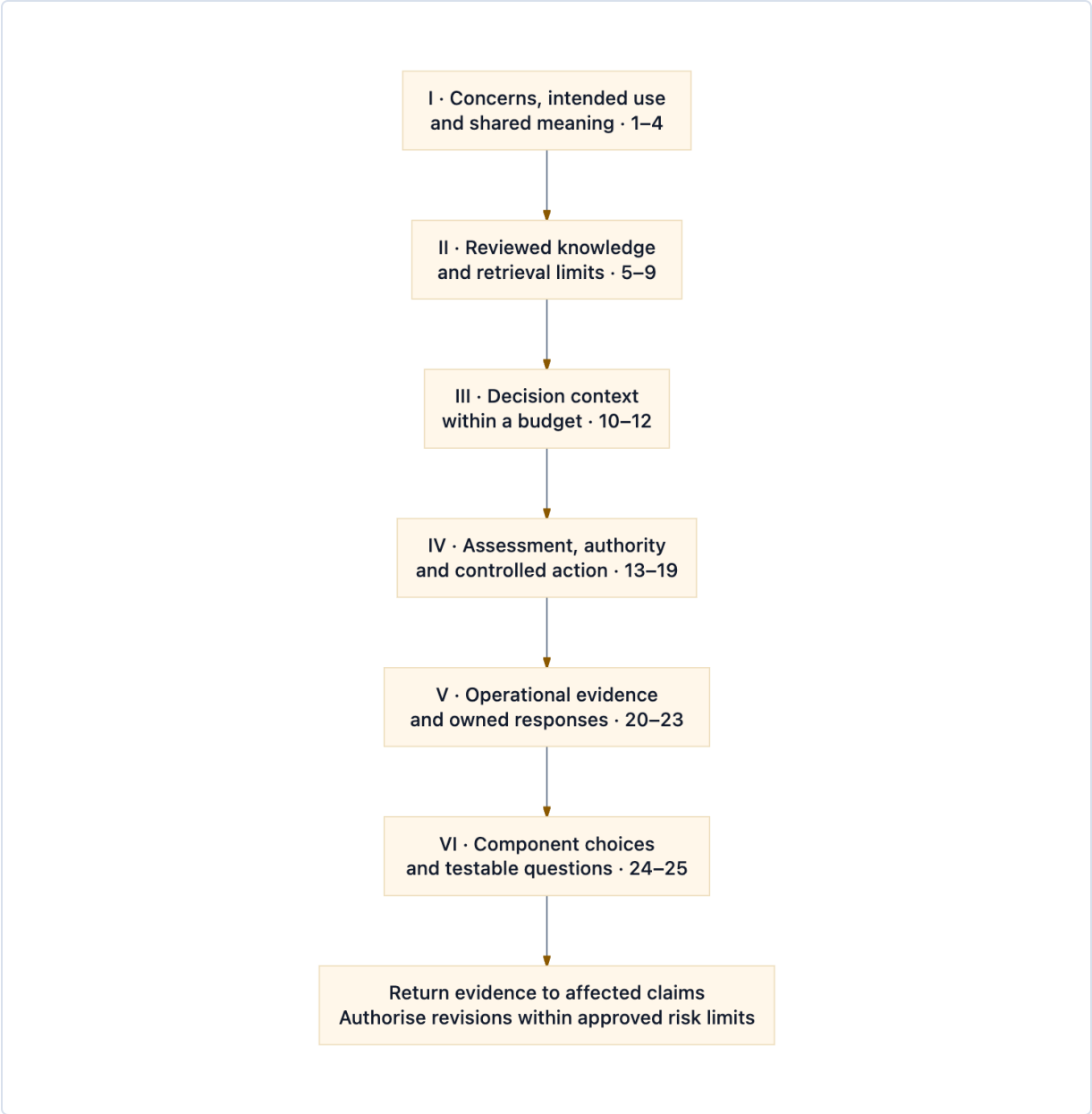


Figure 3: Introduction recap: each reading group supplies a result needed to connect stakeholder concerns, intended use and requirements to service behaviour, evidence and authorised revision.

Key Sources

- Alexander Turkhanov, “Context Engineering: The Missing Discipline in AI Architecture”, Zestic AI blog, https://zestic.ai/blog/context-engineering-missing-discipline-ai-architecture (2025): the framework this book builds on
- Microsoft, GraphRAG paper and documentation (2024)
- Armin Ronacher, “Agent Design Is Still Hard” (2025)
- The author’s systems, OpenClaw and Terraphim, described in Chapters 7, 9 and 20
- Anatoly Levenchuk, FPF, revision 0c6ade2, 8 October 2026, inspected 9 October 2026: selected core distinctions and Systems Engineering DPF patterns SYSE.2 and SYSE.17; Chapter 4 identifies the relevant passages

Chapter 1: Why AI Software Projects Fail Differently

Consider a case assembled from incidents of a kind that have been reported publicly and from the author's own evaluation work in Chapter 9; the details are a composite, not a single company's history. A healthcare provider deploys an LLM-powered assistant to help clinicians with dosing questions. Within months it is recommending dangerous doses. The root cause is not an ordinary code defect; the application code did what it was written to do. The context the assistant relies on has drifted. A pharmacology reference in its retrieval corpus contains a typo, "800mg" where "80mg" was meant, and the model faithfully reproduces it, because nothing between the corpus and the clinician checks the number against anything.

The pattern has three evidence bases, which this book keeps separate: incidents in the public record, such as the fabricated legal citations in *Mata v. Avianca* (2023); the author's own evaluation, in which a grounding gate caught a tenfold dosing error (Chapter 9); and invented connective detail, such as the timeline above. **AI software can fail while its application code continues to work as intended.** Errors in context can accumulate without producing an error signal. This chapter explains the reliability checks needed when those outputs affect patients, customers or money.

The Traditional Software Failure Model

For fifty years, software engineering has operated on a relatively stable model of failure. When traditional software fails, it fails in deterministic ways:

Bugs are reproducible errors in logic or implementation. A null pointer exception happens every time certain conditions are met. The bug exists in code, and code can be fixed.

Schedule overruns are failures of estimation and resource planning. They are painful, but visible. You know when you're behind.

Scope creep is the gradual expansion of requirements beyond original intent. It erodes margins and delays delivery, but it is manageable through change management.

These failure modes share a common characteristic: **they are testable**. You can write unit tests for null pointers, track burndown charts for schedules, and enforce change control for scope. The failures are discrete, observable, and fixable through engineering discipline.

Agile, Waterfall and DevOps all include processes for finding and correcting failures in code and delivery.

The New Failure Model: Context Drift

Figure 1.1 contrasts the two failure models this chapter describes.

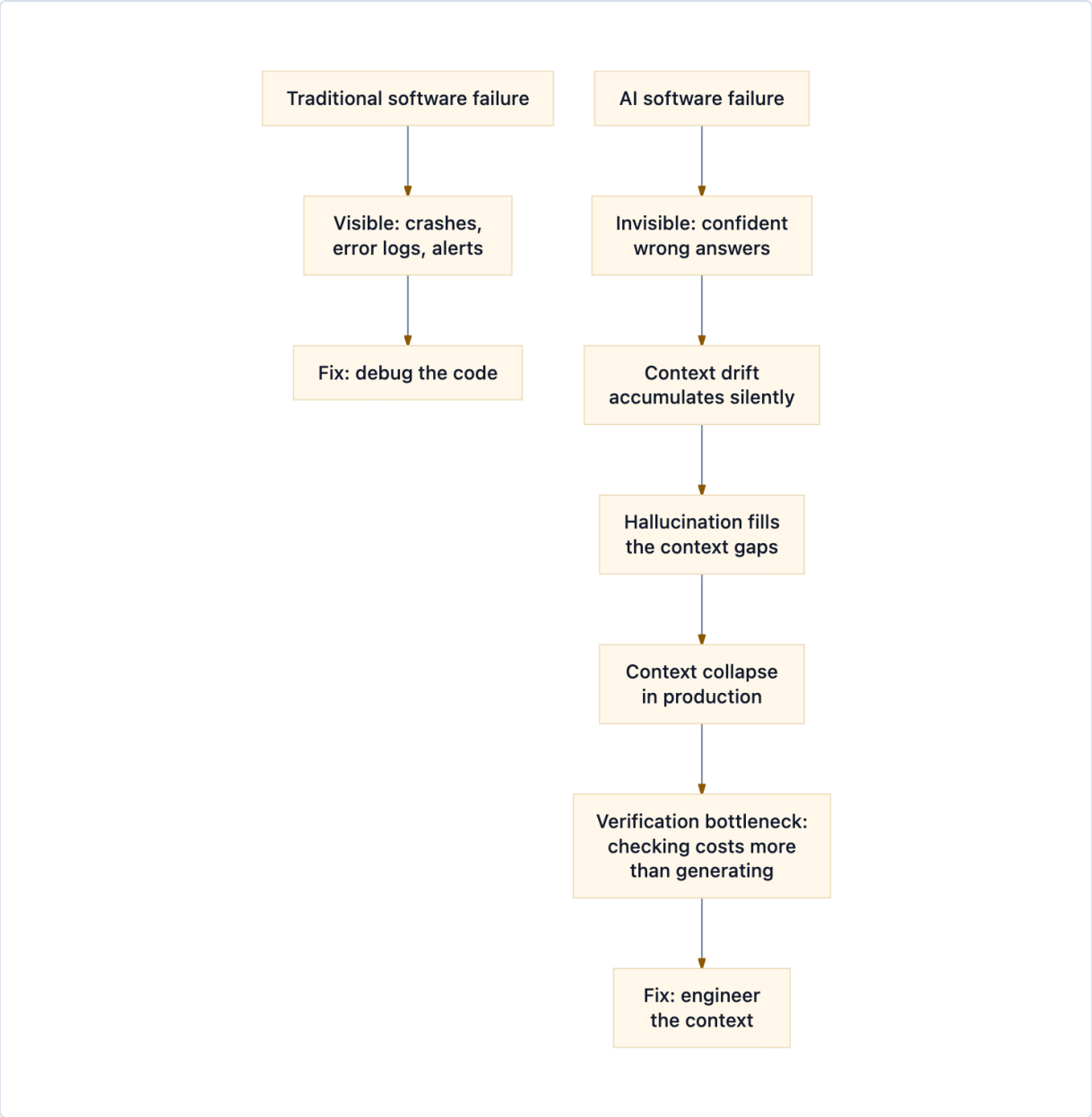


Figure 1.1: Traditional failures are visible and debuggable; AI failures accumulate silently as context drift until the verification cost exceeds the generation cost.

AI software introduces a fundamentally different failure mode: **context drift**.

In AI systems, “context” has two distinct layers. **Runtime context** is everything injected at inference time: the prompt, the accumulated

conversation, the retrieved documents, the system instructions, and the configuration parameters. **Training context** is baked into the model itself: the weights learned from training data, fine-tuning adjustments, and embedded knowledge. This book focuses primarily on runtime context – the layer you can engineer, version, and validate. But training context matters too: when training data contains errors, no amount of runtime context engineering can fully compensate. Runtime context can be versioned; in most deployments the author has seen, it is not, and training context cannot be versioned by anyone but the vendor. Both shift: runtime context with every interaction and retrieval, training context with every model update. Context engineering starts with the layer that can be controlled.

Context drift is the gradual misalignment between what the AI system understands and what its operators intend. Small inconsistencies in documents, instructions and conversation history can accumulate without triggering an error.

Consider a customer service chatbot whose retrieval corpus is the support tickets of 2024 (the runtime context, not the model's training). By 2026, the product has changed three times. The model's knowledge of the product is now inconsistent with reality. But the system doesn't crash. It just gives answers that are subtly wrong. The drift is invisible until a customer receives incorrect information.

The chatbot can keep responding while its answers become less reliable.

Enter Context Engineering

Zestic AI's 2025 framework describes context engineering as a software architecture discipline. It addresses assumptions, stakeholder interpretations, business models and objectives that can change independently of application code.

The Zestic AI framework identifies key failure patterns in enterprise AI programmes:

- Goals agreed in discovery are reinterpreted during development.
- Account teams commit to outcomes before engineering validates feasibility.
- Domain terms acquire different definitions between workshops.
- Documents contradict one another without a reconciliation check.
- Projects pass stage gates without checking semantic consistency.

When context expands faster than it is structured, each new interpretation enters without being reconciled against the existing ones, so contradictions accumulate. A reconciled model records the agreed interpretations, and gates check additions against them. The following chapters develop those mechanisms.

Confabulation and Incorrect Context

The second failure mode is hallucination, or **confabulation**. Predicting likely continuations can fill gaps in training data or

retrieved context with plausible but unsupported content.

An AI system can present incorrect input as a confident answer. An explicit error alerts the operator to a failure; a fluent but unsupported answer can pass unnoticed.

In the opening healthcare case, the model reproduced an error in its retrieved context. The composite illustrates a context-integrity failure: the model could repeat a supplied number without any check that the number was correct.

The application can therefore complete its programmed workflow while returning an incorrect answer.

Narratives Without Explanatory Power

The Russian systems engineering tradition offers a precise vocabulary for this problem. Medvedeva (School of Systems Management, 2021) distinguishes between **explanations** and **narratives**. An explanation is grounded, detailed, and falsifiable – it indicates the internal structure of cause-and-effect relationships and can be tested against reality. A narrative is sequential, persuasive, and easily remembered – but it does not require grounding and is not subject to the rules of evidence.

Both create a sense of understanding, but only an explanation lets you predict what happens next.

An LLM can produce a plausible causal account without evidence that the account is correct. The System School’s distinction is useful

here: a “transparent box” exposes how a result is produced, while a “black box” is understood through observed inputs and outputs. A fluent account from the model does not, by itself, expose or validate its mechanism.

Knowledge graphs (Part II), validation gates (Chapter 14), and deterministic search (Chapter 6) make relationships and supporting evidence explicit. Reviewers can inspect those records when checking a model’s explanation.

The System School offers five criteria for distinguishing strong explanations from weak ones: (1) **more grounded** – specific, with narrow claims rather than vague generalities; (2) **more detailed** – transparent-box, showing internal mechanism; (3) **more falsifiable** – cutting off impossible worlds, not just describing possible ones; (4) **fewer assumptions** – not introducing new unsupported premises; (5) **solves problems of the old theory** – offering something the previous explanation could not. Applied to a concrete output, the criteria are a checklist. Suppose a support assistant answers “the sync failed because the two systems were out of alignment”. It is not grounded (which systems, which field?), not detailed (what mechanism?), not falsifiable (what observation would show it wrong?), it introduces an assumption (that alignment was the cause) and it explains nothing the ticket did not already say. An answer that passes is specific: “the order export rejected 14 rows whose `currency` field was empty; the importer treats empty as invalid since the 2.3 release”. The validation frameworks in Chapter 14 turn this checklist into automated checks.

The Illusion of Simplicity

An LLM API presents a simple interface:

```
response = llm.generate(prompt="What is 2+2?", temperature=0.7)
```

The short interface hides the work needed to select context, check outputs and handle failures in production.

This book calls the gap between a working prompt and a dependable production system the **AI slope**. A prompt demonstration may take minutes; retrieval validation, monitoring and regulatory checks require considerably more work. Teams that estimate from the demonstration can commit to delivery before understanding that work.

Traditional software also takes more work to deploy than to prototype. Testing, CI/CD and code review address some of that gap. AI teams also need evaluations, context checks and model monitoring; practices for these are still developing.

Case Studies: Context Drift in the Enterprise

The two cases below are composites. Each describes a pattern that has appeared in public incidents; the clearest documented example of the second is *Mata v. Avianca* (S.D.N.Y. 2023), in which lawyers filed a brief containing case citations fabricated by ChatGPT and were sanctioned for it.

The Finance Example

A quantitative trading firm deployed an LLM to generate daily market summaries. The system was fed a knowledge base of financial news, earnings reports, and market data. Within two weeks, the summaries began contradicting themselves. One day, the model would report that a company "exceeded earnings expectations." The next day, the same company would be described as "missing targets."

The root cause: the knowledge base was updated daily, but the context window could only hold the most recent 50 documents. The model was comparing recent positive news against older negative news without temporal awareness. The "context" was internally inconsistent, and the model faithfully reflected that inconsistency.

This case illustrates **semantic misalignment**: the AI system has access to conflicting information but no framework for resolving the conflict. A conventional program given inconsistent data would typically fail a validation step or raise an error. AI software produces inconsistent outputs that appear authoritative.

The Legal Example

A law firm implemented an AI assistant trained on case law to support legal research. The system worked brilliantly for six months. Then it began citing cases that didn't exist. The model was generating fictional case citations that sounded plausible.

The root cause, as in the documented *Mata v. Avianca* case, was the absence of any check between generation and filing: the model produced plausible citations and nothing verified them against an authoritative legal database. The “worked for six months” part is the dangerous part. Nothing had changed in the system; the failure was there from the first day and surfaced only when a citation was checked.

This illustrates **unverified generation**: a system that produces references without a gate that resolves them is correct exactly as often as the model happens to be, and nobody knows the rate until someone looks. Chapter 9’s grounding gate exists for this case.

Context Complexity vs. Code Complexity

Traditional software engineering has managed complexity through abstraction: modules, APIs, layers, contracts. We control complexity by encapsulating it.

Semantic complexity concerns meaning, interpretation and intent. Module boundaries alone do not resolve conflicting definitions or unstated assumptions.

Consider the difference:

- **Code complexity** is quantifiable: lines of code, cyclomatic complexity, coupling metrics. You can measure it and reason about it.

- **Context complexity** is qualitative: it is the sum of all the assumptions, interpretations, and implicit knowledge that the AI system relies on. No single number captures it; it has to be described, which is what the domain model of Chapter 4 is for.

A context containing 500 terms, 200 relationships, 50 documents and 50 conversation turns needs checks beyond those applied to a codebase. Its dependencies include synonyms, assumptions and decisions that may be implicit.

Managing this complexity requires explicit definitions and checks for contradictions between context sources, alongside the tools used to manage code.

The Concept Modelling Crash Course

Three terms recur in context models: **entities, relationships and constraints**.

Entities

An **entity** is a distinct concept that the AI system must understand. In a customer service context, entities include: customer, product, order, refund, warranty, complaint. In a medical context, entities include: patient, diagnosis, treatment, dosage, contraindication.

The entity “dosage” needs attributes for safe ranges, administration routes and patient-specific modifiers. A model given only the word has to infer those details.

Critically, **assigning an entity to the wrong type** causes cascading errors. A systems-engineering primer on ontology ("Ontology in Plain English", translated from Russian) makes the point with five stages of problem-solving, problem, requirement, strategy, plan, execution, each a different type, and shows how stating a plan as if it were the problem collapses the solution space. The technical form of the mistake is everywhere in the prompts the author reviews. A task file says "the API must use Redis for caching": that is an implementation stated as a requirement, and the real requirement, say a 200-millisecond response bound, is never written down. The agent then optimises for Redis, and when Redis is the wrong tool it cannot know, because the thing it was meant to satisfy was never in its context.

Relationships

A **relationship** connects entities and defines how they interact. The relationship between "drug" and "contraindication" is critical for medical AI. The relationship between "customer" and "warranty" determines whether a return is authorised.

Relationship types distinguish how entities are connected. A drug may be *prescribed to* one patient and *contraindicated for* another. The graph must record that distinction explicitly.

Constraints

A **constraint** is a rule that limits how entities and relationships can be combined. "Dosage must be positive" is a constraint.

“Contraindication applies when patient has condition X” is a constraint.

Constraints give safety gates rules to check, such as rejecting an 800mg recommendation when the reference dose is 80mg.

These fundamentals are applied in Chapter 8, where we build a knowledge graph using exactly these concepts: entities from the domain model, relationships between them, and constraints that the AI must respect.

The Verification Bottleneck

Entities, relationships and constraints are the vocabulary for describing context; the next question is who checks that the description holds. There is a deeper structural problem beneath context drift and hallucination: **verification does not scale at the same rate as generation**.

The Economics

Figure 1.2 lays out the economic argument of this section.

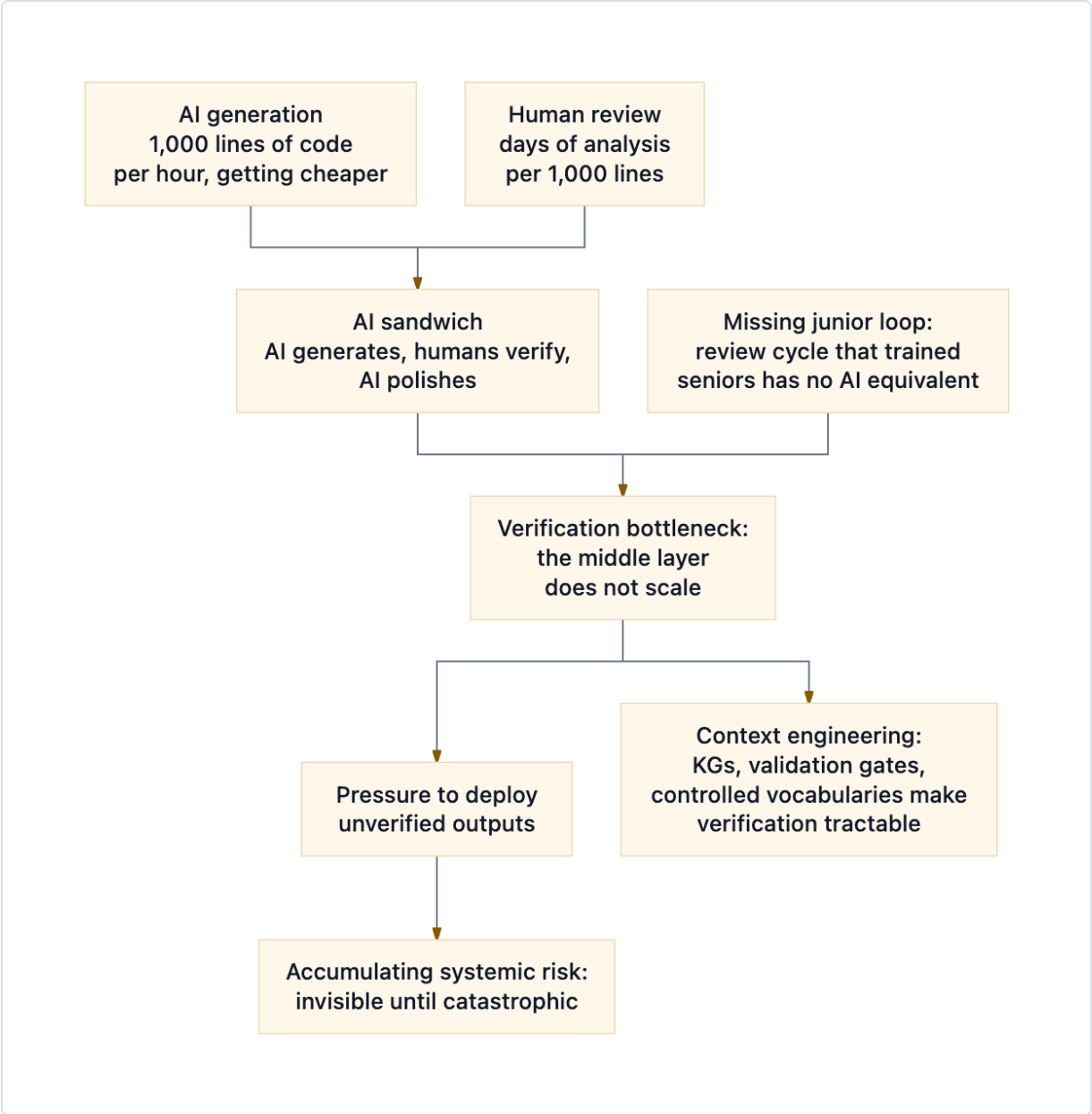


Figure 1.2: The verification bottleneck: generation gets cheaper, human review does not, and the missing junior loop removes the training mechanism that used to close the gap.

The cost of AI generation falls with every model release. An LLM can produce code far faster than anyone can read it. In the author's delivery work, a model writes in an hour what a senior engineer needs days to review with care; that is the author's experience of the ratio, not a measurement, and the reader's own ratio is the one that

matters. The pressure to ship leads to superficial verification. Subtle bugs accumulate as hidden technical debt.

Catalini, Hui and Wu (2026, arXiv 2602.20946) argue that the binding constraint on growth is no longer intelligence but human verification bandwidth; that claim is theirs. The firm structure that follows, and the name for it, are this book's. This book calls the resulting firm structure the "AI sandwich": AI does the measurable generation, humans do the unmeasurable verification, AI does the final polish. But the middle layer – human verification – does not scale. At some point, it becomes economically rational to deploy unverified AI outputs. The result is accumulating systemic risk: failures that are statistically invisible until they are catastrophic.

The Missing Junior Loop

Historically, junior engineers served as a training pipeline: they did measurable work (write code, run tests, fix bugs), seniors reviewed their output, and through that review cycle, juniors developed the judgement to become seniors. AI agents are now filling the junior role – generating code at scale, running tests, fixing linting errors.

But the review cycle that trained seniors has no equivalent in AI-assisted workflows. When a junior engineer writes code, a senior reviews it and provides feedback. When an AI generates 1,000 lines of code in an hour, no human can review it at that pace. The pipeline that creates expertise is under pressure.

Why This Matters for Context Engineering

The verification bottleneck is the economic argument for why context engineering is not optional. Without structured context – knowledge graphs, validation gates, deterministic checks – you are relying entirely on human review to catch AI errors. And human review does not scale.

Context engineering provides the infrastructure that makes verification tractable:

- **Knowledge graphs** enable deterministic fact-checking (Chapter 5)
- **Validation gates** catch errors before they reach production (Chapter 14)
- **Controlled vocabularies** reduce the surface area that needs review (Chapter 8)
- **Hybrid pipelines** route queries to the cheapest reliable method (Chapter 22)

The following chapters develop verification infrastructure to support growing volumes of generated work. Where human review cannot cover that work, unchecked outputs become an operational risk.

What You Can Do This Week

This chapter diagnoses rather than prescribes, but three exercises turn the diagnosis into evidence about your own system:

1. **List your five most critical entities** (for a support assistant: customer, order, refund, entitlement, complaint) and, for each, write the definition your system actually uses. If you cannot find it written anywhere, the definition lives in a prompt or in someone's head, and that is a context drift waiting to happen. Pass: all five written and consistent with each other. Warn: written but contradicted somewhere. Fail: any of the five unwritten.
2. **Find one number in a recent output and trace it.** Where did it come from: a retrieved document, the conversation, the model's training? If you cannot say, you have found the gap the grounding gate of Chapter 9 closes. Pass: the number traces to a document or a graph entry. Warn: it traces to the conversation only. Fail: nobody can say where it came from.
3. **Measure your verification ratio.** For one week, note how long your system spends generating and how long people spend checking what it generated. If the second number is not growing with the first, something is going unchecked. Pass: review time grows in step with generation. Warn: it grows more slowly. Fail: it is flat while generation rises.

Summary: The New Failure Landscape

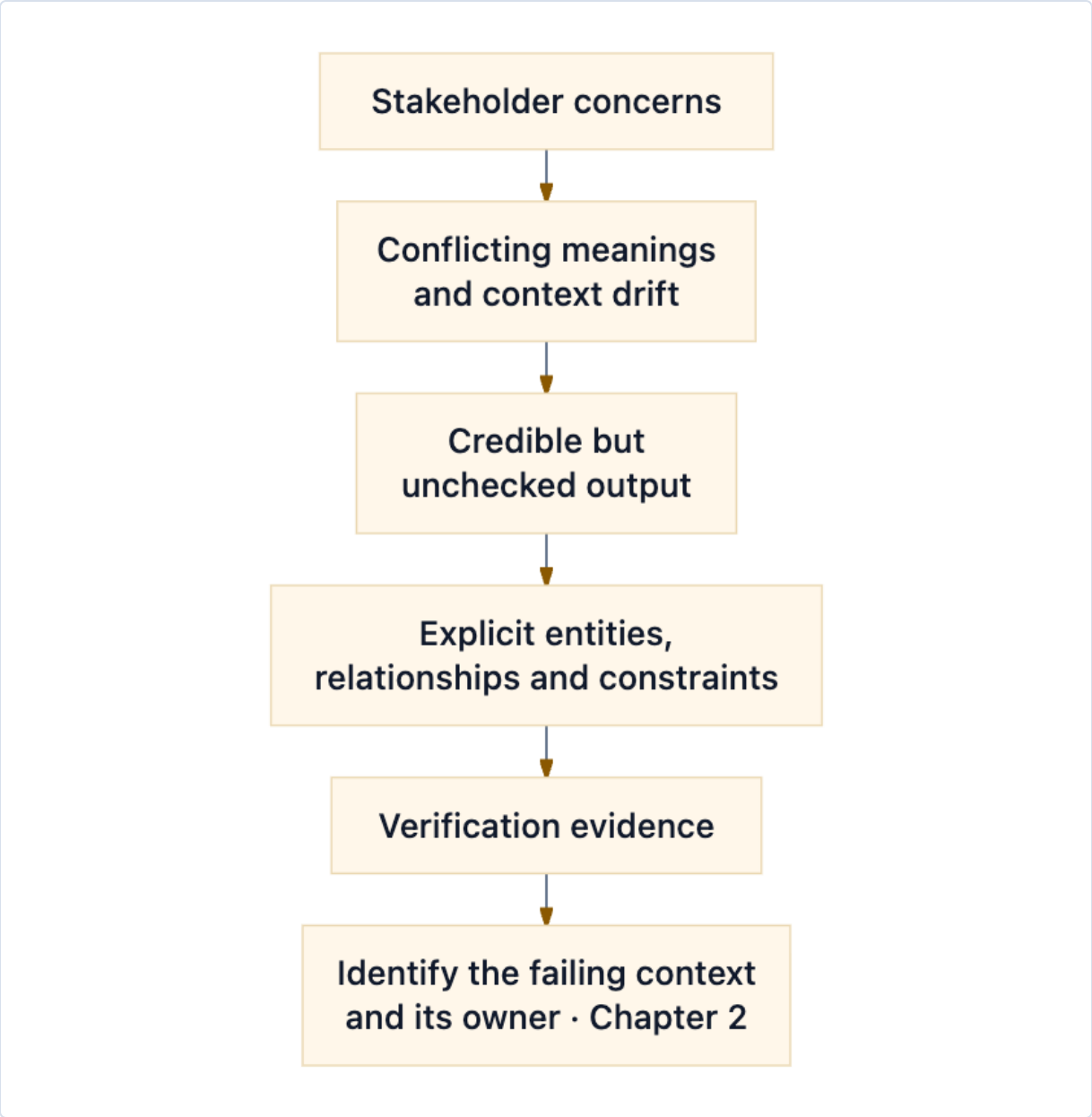


Figure 1.3: Chapter 1 recap: Context drift and conflicting meanings accumulate unless definitions, constraints and evidence are checked.

AI software fails differently because:

1. Context drift accumulates without necessarily producing an error signal.

2. Confabulated answers can appear credible until checked against a source.
3. A working prompt understates the work needed for production reliability.
4. Conflicting meanings require checks beyond those used to manage code complexity.
5. Entity definitions, typed relationships and constraints make those checks possible.

The synthetic maintenance case has the same diagnostic question: a rule can parse correctly while removing a required review or reporting a job restored before delivery. A code check alone cannot establish that the service fulfils the owner's need. Chapter 2 identifies which context supplied the meaning, instruction or missing observation, and who can change it.

Key Sources

- Alexander Turkhanov, "Context Engineering: The Missing Discipline in AI Architecture", Zestic AI blog, https://zestic.ai/blog/context-engineering-missing-discipline-ai-architecture (2025)
- Christian Catalini, Xiang Hui and Jane Wu, "Some Simple Economics of AGI" (arXiv 2602.20946, February 2026): verification bandwidth as the binding constraint; the "AI sandwich" label is this book's

- Armin Ronacher, “Agent Design Is Still Hard” (blog, 2025)
- *Mata v. Avianca, Inc.*, No. 22-cv-1461 (S.D.N.Y. 2023): sanctions for fabricated citations
- MedGemma competition case study (Chapter 9); the opening and enterprise cases in this chapter are composites
- Medvedeva, “Creating Explanations” (Ontologics and Communication, School of Systems Management, 2021) – explanation vs narrative distinction, epistemic status, grounding
- “Ontology in Plain English” (translated from Russian, systems engineering community) – entity type misassignment

Chapter 2: What “Context” Actually Means in AI Development

This chapter defines context, separates its components and explains how to assign owners and checks to them.

Context is everything, other than the model’s weights, that determines what the model produces for a given request: the visible conversation, the hidden instructions, the retrieved material, the generation settings, and the assumptions nobody wrote down. Conventional configuration surfaces (settings files, environment variables, feature flags) are enumerable and can be listed, versioned and diffed, even though production systems also carry runtime state and environmental dependencies that are harder to pin. Context includes unstated assumptions beyond that surface, so configuration tools alone cannot manage it. Decomposing it into dimensions with owners and checks makes the management task explicit.

Beyond the Chat Window

When most people think of “context” in AI, they think of the chat window, the conversation history the model uses to generate responses. The chat window is visible and directly influences the output, which is why it gets the attention; it is also the smallest part of what the model relies on. The components fall into two groups: **managed runtime context**, which the engineering team controls,

and **external influences**, which shape the output but sit outside that control.

Managed runtime context:

- Conversation history: the visible exchange between user and system.
- System prompt: instructions defining the model's persona, capabilities and constraints.
- Retrieved documents: reference material supplied to the model.
- Configuration parameters: temperature, max tokens, top-p and other generation settings.

External influences:

- Training context: what the weights encode, which the vendor controls. Chapter 1 distinguishes it from runtime context.
- Implicit assumptions: what the model takes for granted about the world from training, without stating it.

Visibility differs by role. The user sees the conversation. The operator can see the system prompt, the retrieved documents and the parameters if the system logs them, and many systems do not. Nobody sees the training context or the assumptions directly; they are observed only through their effects. This is why context management is so difficult: much of what shapes the output is examined by no one.

This list of six components answers “what is in the context”. The four dimensions that follow answer “whose context is it and how fast does it change”, and the four assumptions later in the chapter answer “what does the model take for granted”. They are three views of one thing, not three competing taxonomies: each component belongs to one or more dimensions, and the assumptions are what the components leave unstated.

The Four Dimensions of Context

Figure 2.1 shows the four dimensions and where they meet.

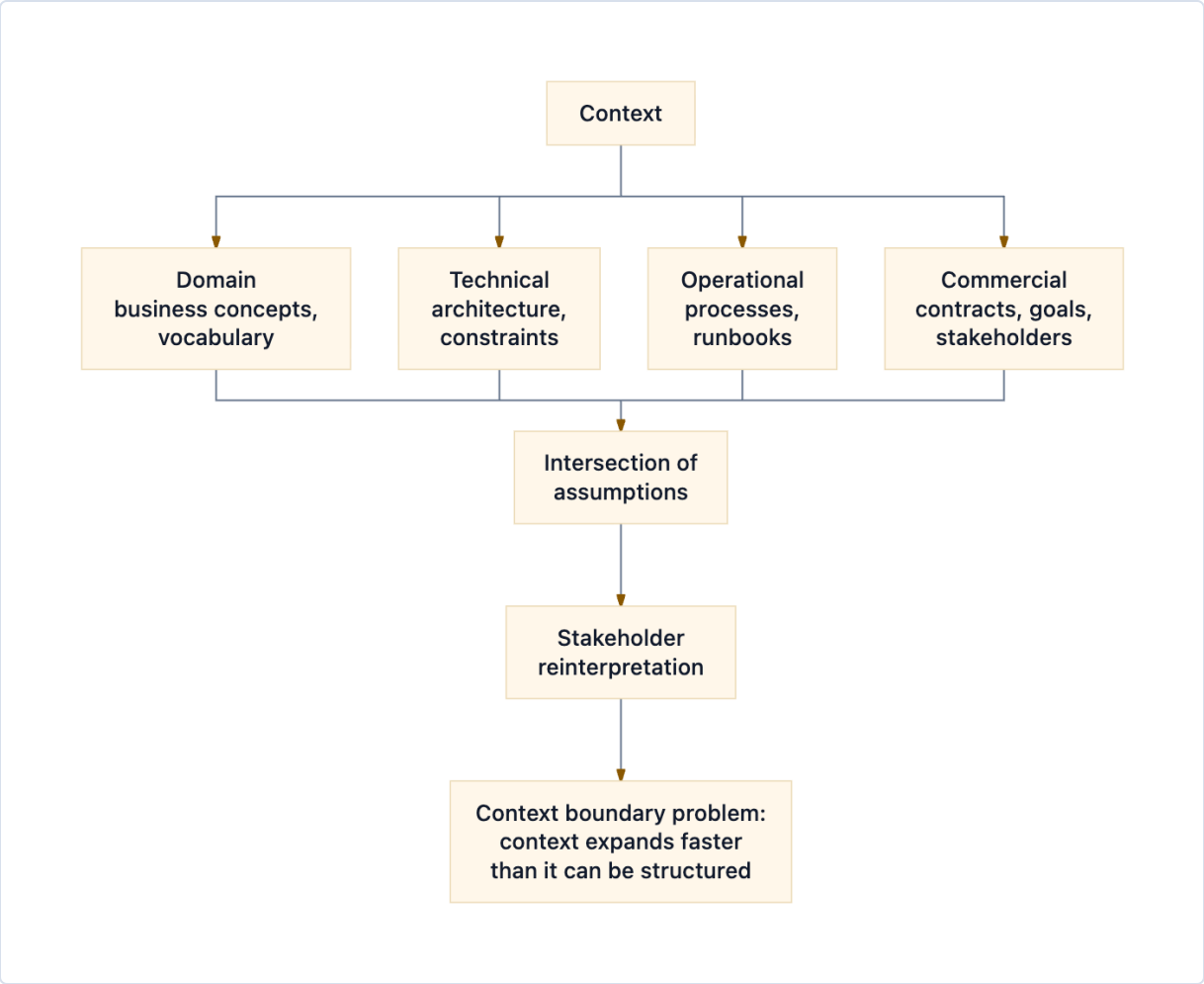


Figure 2.1: The four dimensions of context converge on an intersection of assumptions that stakeholders reinterpret, which is why context expands faster than it can be structured.

Drawing from the Zestic AI framework (Turkhanov, 2025; the four dimensions are the framework's, their descriptions below are the author's), we identify four distinct dimensions of context that must be managed separately:

1. Domain Context

Domain context describes the problem space: its entities, relationships, rules and constraints. In medical AI, it includes drug interactions, dosage limits, contraindications and diagnostic criteria.

In legal AI, it includes case law, statutes, precedents and procedural rules.

Domain context changes with the domain, for example when drugs or regulations change. Errors here can directly cause harmful outputs.

2. Technical Context

Technical context describes the system's capabilities, limitations, tools and integration points. In Claude Code, it includes the available tools (Bash, Edit, Read, Write), MCP servers, project structure and execution environment.

Adding a tool or changing its behaviour changes technical context. Version and validate tool definitions so that the model's instructions match what the tools actually do.

3. Operational Context

Operational context describes the current session and task: the user's goal, completed work and expected outcome. In a coding assistant, it includes the current file, the error being debugged and the failing test.

Operational context is the most volatile. It changes with every message, every tool use, every outcome. This is the context that must be managed in real-time, with careful attention to what enters and exits the context window.

4. Commercial Context

Commercial context describes business constraints and priorities: budget, timeline, stakeholder expectations and go-to-market requirements. Zestic AI's framework identifies neglected commercial constraints as a recurring source of delivery failures.

Commercial context is often implicit. It lives in the heads of stakeholders, in slide decks, in contract terms. When it conflicts with the other three dimensions, the AI system has no way to detect the conflict. It will confidently generate outputs that are technically correct but commercially disastrous.

Context as the Intersection of Assumptions

A useful way to think about context is as the **intersection of assumptions** that the AI system must navigate:

- Domain: which entities, relationships and rules apply?
- User: what do they know, want and intend to do with the output?
- System: what can it do, and which actions are prohibited?
- Reality: which facts are current and supported by reliable evidence?

When these assumptions are aligned, when the AI's understanding matches the operators' intentions, the system works. When they diverge, which in the author's experience happens in every multi-team programme sooner or later, the system fails.

Some assumptions remain unstated in the prompt and retrieved material. They come from patterns in the model's training or from its interpretation of earlier interactions. Operators observe their effects in the output rather than inspect them directly.

In plain terms: every person and every system on a project carries its own list of what it takes to be true, and the lists differ. The Russian systems engineering tradition formalises this as **epistemic status** (Medvedeva, School of Systems Management, 2021): the enumeration of what each agent believes to be true about the world. Not all agents describe the world with the same set of statements. One stakeholder might believe statements A, B, and C are true, while another believes A, C, and D are true – and considers B to be

formulated in a meaningless way. Critically, each agent also holds beliefs about *other agents' epistemic statuses*: what they think others believe.

When an AI system operates on context assembled from multiple stakeholders, it inherits conflicting epistemic statuses without any mechanism to detect the conflict. The model cannot distinguish between a statement that all parties agree on and one where half the stakeholders would reject the premise. Context engineering addresses this by making epistemic status *explicit*: the reconciled domain model (Chapter 4) forces alignment, and validation gates (Chapter 14) catch contradictions before they propagate.

Why Context Expands Faster Than It Can Be Structured

If assumptions are the hidden content of context, the next question is why there is always more of it. Turkhanov describes this as the **entropy problem**: in enterprise AI programmes, context expands faster than it can be structured. Every stakeholder interaction adds new interpretations. Every product change adds new domain knowledge. Every integration adds new technical constraints.

In software engineering, stakeholder concerns drive the elicitation of needs and requirements. Specifications, user stories and acceptance criteria help make those requirements explicit and checkable. Chapter 4 follows the links from stakeholders and concerns to the

services the solution shall provide, using a shared domain vocabulary.

In AI software, there is no equivalent structure by default. The model's flexibility means that almost anything can be added to context, and nothing in the model forces the addition to be explicit, validated, or consistent; only the engineering around the model can. The context grows organically, accumulating assumptions, exceptions, and edge cases that no one explicitly approved.

Context engineering exists to impose structure on this growth: reconciliation at gates, vocabulary curation, and budgets.

The Role of Stakeholder Reinterpretation

In the delivery programmes the author worked on, **stakeholder reinterpretation** caused substantial damage; this is an observation, not a measured ranking. Developers, product managers, customer-success staff, salespeople and executives used different definitions and priorities, and remembered different decisions.

When these stakeholders interact with the AI system, they inadvertently introduce their reinterpretations into the context. The model doesn't know that "the feature" means different things to different people. Nothing in its input marks one speaker's usage as more authoritative than another's.

If earlier interpretations remain in context after decisions change, the model receives incompatible descriptions without knowing which

one takes precedence. Its output may then match none of the stakeholders' intentions.

Architecture description provides vocabulary for these differences. ISO/IEC/IEEE 42010 defines a *viewpoint* as the conventions for constructing a view that frames particular stakeholder concerns, and a *view* as the description produced from that viewpoint; Medvedeva (School of Systems Management, 2021) teaches the same idea as descriptions composed from a role's position, with the role's objects of interest highlighted. Each stakeholder occupies a different role, and each role highlights different objects of interest. The product manager sees the system through a market-fit viewpoint. The engineer sees it through an implementation viewpoint. The regulator sees it through a compliance viewpoint. These viewpoints produce different descriptions of the same system.

When an AI system ingests context from multiple viewpoints without labelling which viewpoint each piece belongs to, it cannot distinguish between complementary perspectives and contradictory ones. Context engineering addresses this by making viewpoints explicit: the role-based graphs in Chapter 7 provide different synonym edges and relationships per viewpoint, and the reconciled domain model in Chapter 4 forces alignment at the points where viewpoints must agree.

The four ideas of this middle section fit together as follows:

Idea	What it names	Which dimensions it mostly affects	What to do about it

Idea	What it names	Which dimensions it mostly affects	What to do about it
Four assumptions	What the model takes for granted	All four	Make them explicit in the domain model (Chapter 4)
Epistemic status	That each party's list of truths differs	Domain, commercial	Reconcile the lists at gates; validate contradictions (Chapter 14)
Entropy	That context grows faster than it is structured	Operational, technical	Budgets and compression (Chapters 10 and 11); vocabulary curation (Chapter 8)
Viewpoints	That each role sees a different system	Domain, commercial	Label context by role; role-based graphs (Chapter 7)

AI-Specific Context Variables

Beyond the four dimensions, AI systems have context variables that are unique to the technology:

Model version. A prompt that worked on GPT-4 may produce different results on GPT-4.1. Record the model version so that changes in output can be investigated.

Prompt evolution. Special cases and workarounds accumulate as new situations arise. Track those changes and their reasons so that later reviewers can understand them.

Model updates behind a stable name. Vendors can repoint model aliases to newer versions with different behaviour. Record the resolved version with each call to detect updates that occur without a change to application code or prompts.

Output interpretation. One stakeholder may treat an answer as a suggestion while another treats it as a decision. State how outputs are intended to be used.

Diagnosing Which Dimension Is Failing

The four dimensions are a diagnostic tool as well as a description. Each has a characteristic rate of change, a natural owner, a typical failure, and a check that detects it:

Dimension	Changes	Owner	Typical failure	Check that detects it
Domain	Slowly, with the real world	Domain expert, through the domain model (Chapter 4)	A wrong rule or value with confident output	Grounding gate against the knowledge graph (Chapter 9)
Technical	With every release	Platform or tooling team	Tool used wrongly after a definition changed	Tool definitions versioned and validated in CI (Chapter 14)

Dimension	Changes	Owner	Typical failure	Check that detects it
Operational	With every message	The session itself, bounded by the budget	Needed context truncated or stale context retained	Token tracking and compression reports (Chapters 10 and 11)
Commercial	With every stakeholder conversation	Account owner, kept separate from engineering (Chapter 3)	Technically correct output that breaks a commitment	Commercial constraints stated as decision containers and reconciled at gates (Chapter 4)

For a short audit, name the owner of each dimension; an absent owner leaves responsibility unresolved. Ask where its current content is recorded and how that record is maintained. A prompt or chat thread needs the same ownership and revision checks as other context sources. Ask what would detect a change; “someone would notice” identifies a missing check. Start with the dimension that has the most gaps.

The Context Boundary Problem

A fundamental challenge is determining **what belongs in context and what doesn’t**. In a traditional system, the boundary is clear:

code is in, everything else is out. In an AI system, the boundary is fuzzy: almost anything could be relevant.

Too much context overwhelms the model and exceeds token limits. Too little context leaves the model without the information it needs. The right boundary is task-dependent, which means it must be actively managed rather than statically defined.

Three questions decide inclusion for a given task. Does the task's output depend on this item, or could it be omitted without changing the answer? Is the item current, or has something it describes changed since it was written? Is it already represented by something smaller, such as a summary or a reference the model can follow? An item that passes all three goes in; the token budget of Chapter 10 and the progressive disclosure of Chapters 10 and 12 decide how much of it, and in what form.

Summary: A Multi-Dimensional Problem

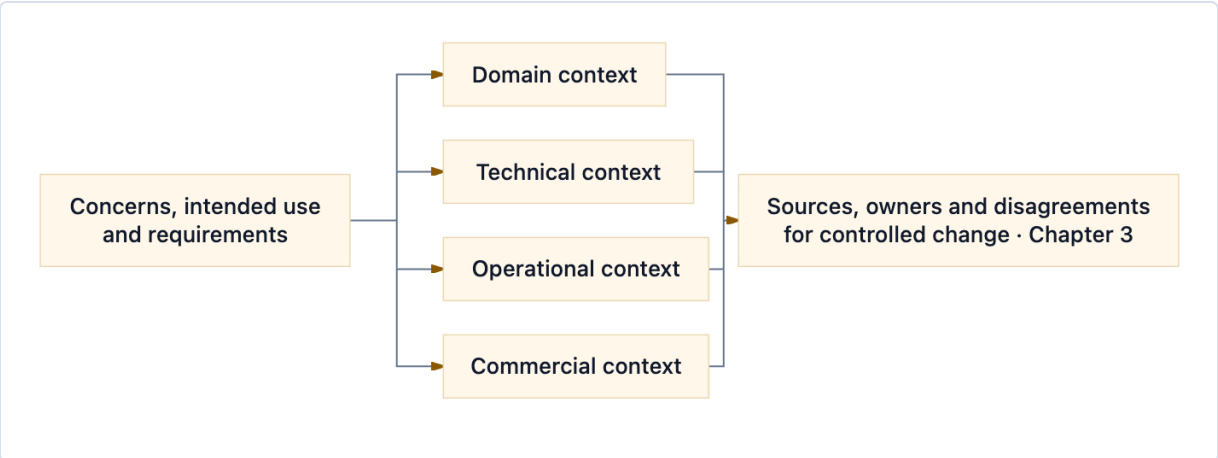


Figure 2.2: Chapter 2 recap: Four interacting dimensions of context need explicit ownership and reconciliation.

In this chapter, we have established that “context” in AI development is not a single thing but a multi-dimensional construct:

1. Domain context describes the problem space.
2. Technical context describes system capabilities and tools.
3. Operational context describes the current session and task.
4. Commercial context describes business constraints and priorities.

In the maintenance case, “restored” is domain meaning, the scheduler and quotas are technical context, the current failed job is operational context, and the promise to a client is commercial context. Record their sources, owners and disagreements. Chapter 3 supplies the decision records and change discipline needed when a promise exceeds the service’s demonstrated capability.

Key Sources

- Alexander Turkhanov, “Context Engineering: The Missing Discipline in AI Architecture”, Zestic AI blog, https://zestic.ai/blog/context-engineering-missing-discipline-ai-architecture (2025)
- Zestic AI delivery framework, internal methodology of the author’s company (2025); the published summary is the Turkhanov article above
- ISO/IEC/IEEE 42010:2011, *Systems and software engineering: Architecture description*, for the viewpoint concept

- Medvedeva, “Creating Explanations” (Ontologics and Communication, School of Systems Management, 2021) – epistemic status of agents
- Medvedeva, “Complex Descriptions and Meta-Languages” (Ontologics and Communication, School of Systems Management, 2021) – viewpoints, ISO 42010

Chapter 3: Context Engineering as a Discipline

Chapters 1 and 2 described context drift and defined the context that engineers can control. This chapter sets out the working practices of **Context Engineering**: modelling the domain, tracking decisions and checking changes at delivery gates.

What Context Engineering Is Not

Figure 3.1 summarises the boundary this section draws.

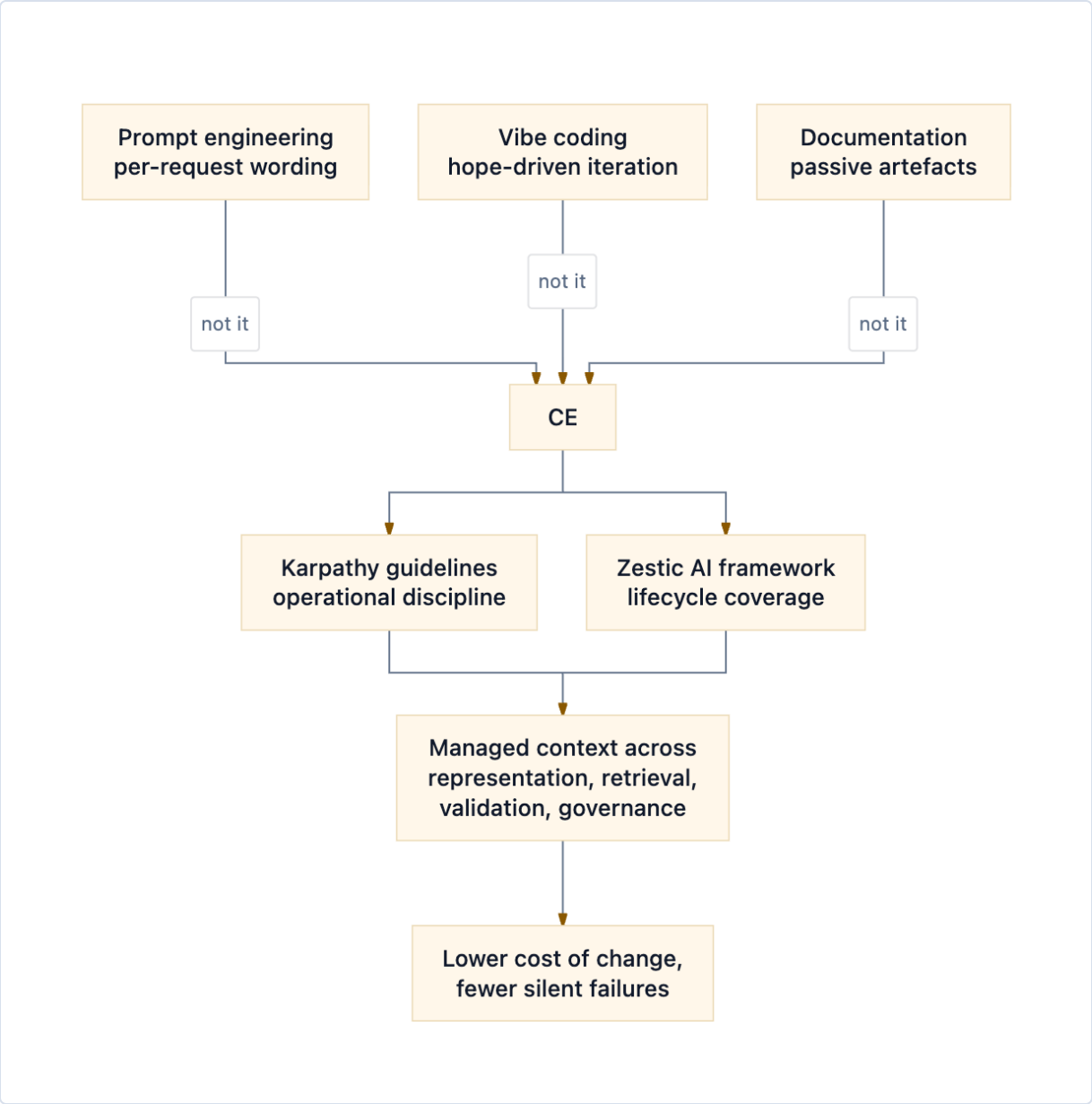


Figure 3.1: Positioning context engineering against the three things it is routinely confused with, and the two operational frameworks that give it substance.

Three related practices address parts of the problem:

Approach	What it optimises	Why it is not enough	What context engineering adds

Approach	What it optimises	Why it is not enough	What context engineering adds
Prompt engineering	The wording of one request	Treats context as a single input; cannot fix a misaligned knowledge base or a conflicting commercial constraint	Manages every dimension of context, not one knob
Vibe coding	Speed of iteration by feel	Drift is invisible and cumulative; nothing detects it or allows rollback	Explicit structure, validation and baselines
Documentation	Describing what is	Passive artefacts that drift from the running system	Artefacts as decision containers that the workflow enforces

Not Prompt Engineering

Prompt engineering focuses on optimising the input prompt to get better outputs from the model. It is concerned with word choice, instruction order, examples, and formatting. It treats the prompt as a tuning knob.

Optimising one prompt cannot resolve every source of context drift. The knowledge base may use conflicting definitions, tool instructions may disagree with tool behaviour, or commercial commitments may conflict with domain constraints. Each needs its own checks alongside prompt revisions.

Not “Vibe Coding”

Andrej Karpathy coined **“vibe coding”** in February 2025 to describe an approach that relies on generated code and iteration by feel, with limited inspection of the implementation.

Iteration by feel can miss the cumulative misalignment described in Chapter 1. Explicit definitions, validation checks and versioned baselines let a team detect drift and roll back to a known-good state.

Not Documentation

Specifications, design documents, API references and runbooks can become disconnected from the running system. Treating them as **decision containers** adds a record of the decisions they express and checks that dependent work remains consistent with them.

The Historical Roots

Context Engineering adapts established practices from systems engineering, domain-driven design and configuration management.

Engineering Control (1960s)

In the 1960s, as software-intensive systems grew in complexity, the systems engineering community developed **engineering control** practices. These included:

- Configuration management controls versions, changes and baselines.

- Requirements tracing links requirements to design, implementation and tests.
- Decision gates check progress against agreed criteria.

These practices emerged because the costs of failure in aerospace and defence systems were catastrophic. They imposed structure on what would otherwise be organic growth.

Context Engineering applies these same principles to AI systems, adapting them for the unique characteristics of probabilistic, generative technology.

Domain-Driven Design (2000s)

In the early 2000s, Eric Evans wrote **Domain-Driven Design**, introducing concepts like the **ubiquitous language** (a shared vocabulary between developers and domain experts), **bounded contexts** (explicit boundaries around domain models), and **aggregates** (clusters of related entities with consistent boundaries).

These concepts map directly to Context Engineering:

- The **ubiquitous language** becomes the controlled vocabulary that the knowledge graph enforces
- The **bounded context** becomes the context boundary we manage
- The **aggregate** becomes the entity cluster with its invariants

Context Engineering adapts these DDD practices to the vocabulary and boundaries of AI systems.

Configuration Management

Configuration management controls changes to code, build scripts, environment variables and deployment specifications.

In AI, configuration expands to include prompts, knowledge bases, model parameters, retrieval configurations, and validation rules. Context Engineering applies configuration management principles to this expanded scope.

The Scientific Method of Context Building

A parallel tradition from the Russian systems engineering community provides a methodological insight. Medvedeva's *Ontologics and Communication* (School of Systems Management, 2021) is a practitioner course text rather than a peer-reviewed paper; it is cited here for its method, which the author has applied in practice, not as an authority on outcomes. Medvedeva observes that explanations are *not* created by observing the world, collecting data, generalising, and then explaining. This is the common misconception. Instead, explanations are created through:

1. **Problem statement** – identifying what is surprising, confusing, or broken
2. **Listing hypotheses** – both existing explanations and new conjectures

3. **Pre-experimental argumentation** – having at least two competing explanations *before* collecting data, so the data can distinguish between them
4. **Listing observations** – observing with a purpose, knowing which observations favour which hypothesis

This matters for context engineering because teams routinely skip steps 2 and 3. They observe a context problem (the AI gives wrong answers), jump to a solution (add more documents to the knowledge base), and never test whether the diagnosis is correct. Context engineering applies this scientific method: define the problem explicitly, generate competing hypotheses about the cause, design validation that can distinguish between them, and only then make changes. This is the methodology behind the validation gates in Chapter 14.

Operational Discipline: The Karpathy Guidelines

While Context Engineering provides the structural framework, day-to-day execution requires operational discipline. The four principles below are the author's distillation of Andrej Karpathy's public guidance on LLM-assisted coding (2025); the quoted lines are paraphrases of that guidance, not citations of a single document:

1. Think Before Coding

> "Don't assume. Don't hide confusion. Surface tradeoffs."

Before generating any AI output, validate that the context is complete and unambiguous. State assumptions explicitly. When requirements are ambiguous, present multiple interpretations rather than silently choosing one. This mirrors the pre-LLM validation described in Chapter 14: catch problems before the model runs.

2. Simplicity First

> "Minimum code that solves the problem. Nothing speculative."

Apply to context as well as code. A smaller knowledge graph whose relationships are precise is more useful than a larger one whose relationships are vague, because every vague edge is a place for the model to confabulate. Every element in the context window should earn its tokens. Remove speculative context – information included "just in case" – because it dilutes the signal and consumes budget.

3. Surgical Changes

> "Touch what is needed, nothing more."

When updating knowledge graphs, prompts, or validation rules, change only what the task requires. Broad refactoring of context introduces drift risk. Small, targeted changes are easier to validate and rollback.

4. Goal-Driven Execution

> "Start from the goal. Always know where you are."

Every context engineering decision should trace back to a measurable objective. If you cannot explain why a piece of context exists and what goal it serves, it is a candidate for removal. This connects to the Zestic AI framework's artefact governance: every artefact is a decision container with a traceable purpose.

These four principles give reviewers checks to apply when a prompt, graph or rule changes.

The Zestic AI Framework

This book draws its structure from the Zestic AI framework, developed by the author's company. That affiliation matters when assessing the author's preference for it. The framework has the following components:

The Reconciled Domain Model

The framework centres on a **reconciled domain model**: agreed definitions of the entities, relationships, rules and constraints in the problem space.

Business scenarios, process models, data schemas, APIs, AI use cases and UX flows must all use definitions consistent with this domain model.

Reconciliation records the definitions stakeholders have agreed to use. Teams can check their artefacts against that shared model and revisit disagreements when the domain changes.

Artefacts as Decision Containers

In the Zestic framework, an artefact is a **decision container**. It records what was decided, why, which alternatives were considered and what follows from the decision.

Traditional documentation says "what is." Decision containers say "what was decided and why," and the difference enables:

- Impact analysis traces which decisions a proposed change affects.
- Rollback restores the previous decision when a change proves wrong.
- An audit trail records the reasoning for reviewers.

Gated Lifecycle with Reconciliation

Enterprise AI programmes need **gated stages** that check progress against agreed criteria before work proceeds.

At each gate, named people do the reconciliation:

- The domain owner validates the domain model for consistency
- The gate reviewer checks the artefacts against each other for coherence
- The delivery lead assesses risk

- The configuration manager sets the freeze, after which changes enter only through formal change requests

This does not mean requirements cannot change. They will. Architecture can change. Even the domain model can change. But the stage makes those changes visible, tracked, and aligned. Progression is conditional, not automatic.

Separation of Account Management and Engineering

A key insight from the Zestic framework is the danger of **conflating account management (commercial) with engineering (technical)**. When the same channel handles client expectations and technical feasibility, decision integrity collapses:

- Commercial urgency overrides engineering baselines
- Technical rigour is relaxed to meet commitments
- Scope creep occurs from informal commitments

Account management and engineering need separate decision records and an agreed process for changing scope. Engineering must have the authority to reject commitments that violate architectural constraints.

Engineering pushback is **positive friction** where a client request becomes a delivery commitment. John Cutler argues that AI automation can remove opportunities for judgement between stages; he recommends deliberate checks and links to original

feedback. See "AI, the Loss of Positive Friction, and What to Do About It" (24 September 2026).

Here, account management brings stakeholder concerns; engineering checks proposed requirements against intended use, service capability and approved risk limits. Engineers should inspect the original request, record assumptions, offer feasible alternatives and reject or escalate unsupported commitments through the agreed authority. Match review effort to the consequences of the decision.

The Cost of Ignoring Context Engineering

Unmanaged meaning creates **structural debt** alongside debt in the code. Its mechanism is the context drift of Chapter 1, the gradual misalignment between what the system understands and what its operators intend. It takes four forms:

- Stakeholders reinterpret goals downstream (semantic drift)
- Domain definitions drift between workshops (vocabulary drift)
- Artefacts exist but contradict each other (coherence drift)
- Prompts evolve without version control (configuration drift)

By the time issues surface in integration or testing, correction costs have multiplied. The classic software-economics finding that later discovery costs more (Boehm, 1981) is disputed in its exact shape, but nobody disputes its direction, and context problems add a twist: a misaligned term is cheap to fix in a glossary and expensive to fix

once a knowledge graph, a prompt library and a model's accumulated conversation all encode it.

This is compounded by AI's probabilistic nature. A deterministic bug reproduces on demand and a test can pin it; a probabilistic misalignment appears in some fraction of outputs and reproduces only statistically, so it needs sampling and a reference to detect (Chapter 14). The combination of structural debt and that detection cost is what makes the cycle vicious.

Summary: A Necessary Discipline

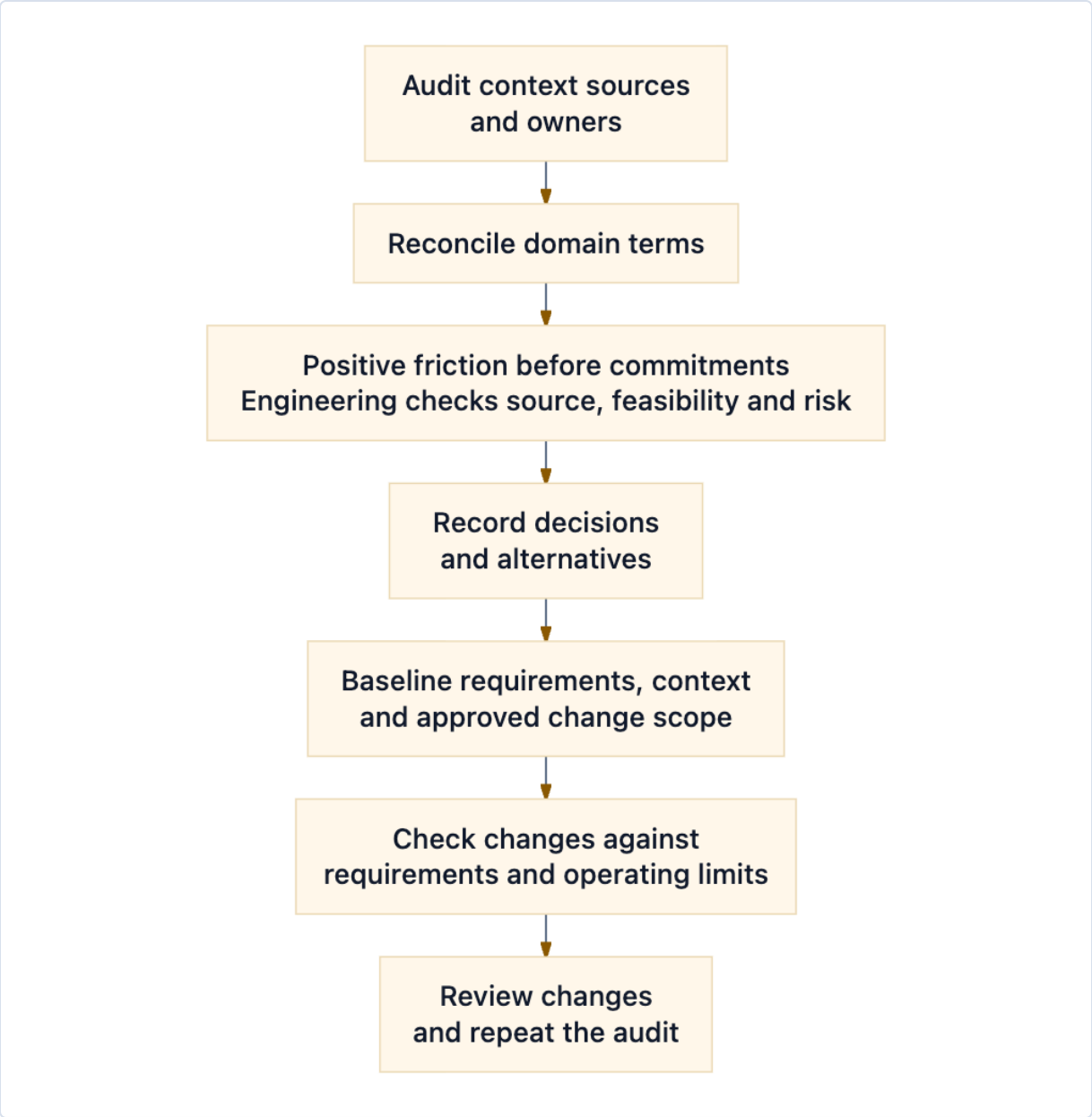


Figure 3.2: Chapter 3 recap: Context engineering joins shared meaning, recorded decisions and lifecycle controls, with positive friction before commitments.

Context Engineering is the discipline of managing the multi-dimensional context that AI systems operate on. It draws from:

- Engineering control (1960s): configuration management, decision gates and baselines.

- Domain-driven design (2000s): ubiquitous language and bounded contexts.
- Zestic AI framework (2025): reconciled domain models, artefact governance and lifecycle gates.

Engineering pushback provides positive friction before commercial commitments: check the source request, feasibility and risk, and retain the decision.

It is distinct from:

- Prompt engineering, which focuses on the wording of model input.
- Vibe coding, which relies on iteration by feel.
- Documentation that describes the system without tracking its decisions.

The cost of ignoring it is structural debt that compounds over time, which is why the discipline is worth its overhead for any AI system whose outputs someone relies on.

For the maintenance proposal, the result is a controlled change request with its source concern, alternatives and review authority. Engineering can challenge a promised repair that exceeds the agreed scope or quota. Chapter 4 turns that review into explicit intended-use conditions, requirements and service allocations, including consequences for other teams.

Getting Started

Four actions, each a day's work or less, that begin the discipline without waiting for the framework:

1. **Audit your context sources.** List everything that reaches the model: prompts, rules files, retrieved corpora, tool definitions. For each, name the owner and the last time it was reviewed.
2. **Find your three most inconsistent terms.** Ask three people what "customer", "order" or your domain's equivalent means. Where the answers differ, you have found where the reconciled domain model (Chapter 4) starts.
3. **Write one decision container.** Take the last significant decision about the system and record what was decided, why, and what was rejected. Put it where the next change will find it.
4. **Pick one gate.** Choose one point in your delivery flow where a context change must be checked before it lands, and make the check mandatory, even if the check is a human reading a diff.

Key Sources

- Alexander Turkhanov, "Context Engineering: The Missing Discipline in AI Architecture", Zestic AI blog, https://zestic.ai/blog/context-engineering-missing-discipline-ai-architecture (2025)
- Eric Evans, "Domain-Driven Design" (2003)

- Andrej Karpathy, public posts on LLM-assisted coding (2025) and the “vibe coding” coinage (February 2025); the four guidelines in this chapter are the author’s distillation
- Systems engineering literature on configuration management (1960s)
- Barry Boehm, *Software Engineering Economics* (1981): cost of late defect discovery
- Medvedeva, “Creating Explanations” (Ontologics and Communication, School of Systems Management, 2021) – scientific method of context building
- John Cutler, “TBM 441: AI, the Loss of Positive Friction, and What to Do About It”, *The Beautiful Mess*, 24 September 2026

Chapter 4: The Architecture of Meaning

Chapter 3 established the working discipline, including engineering's authority to challenge unsupported commercial commitments. This chapter makes the basis for a commitment explicit: concerns, intended use, requirements, allocated services and the evidence needed to review them.

The Zestic AI framework's position, and this chapter's, is that **meaning has to be architected**. Left unstructured, context drifts in the ways Chapter 1 described; given explicit structure, it can be baselined, traced and changed on purpose.

Artefacts as Decision Containers

Figure 4.1 is the map of this chapter: each box is a section below.

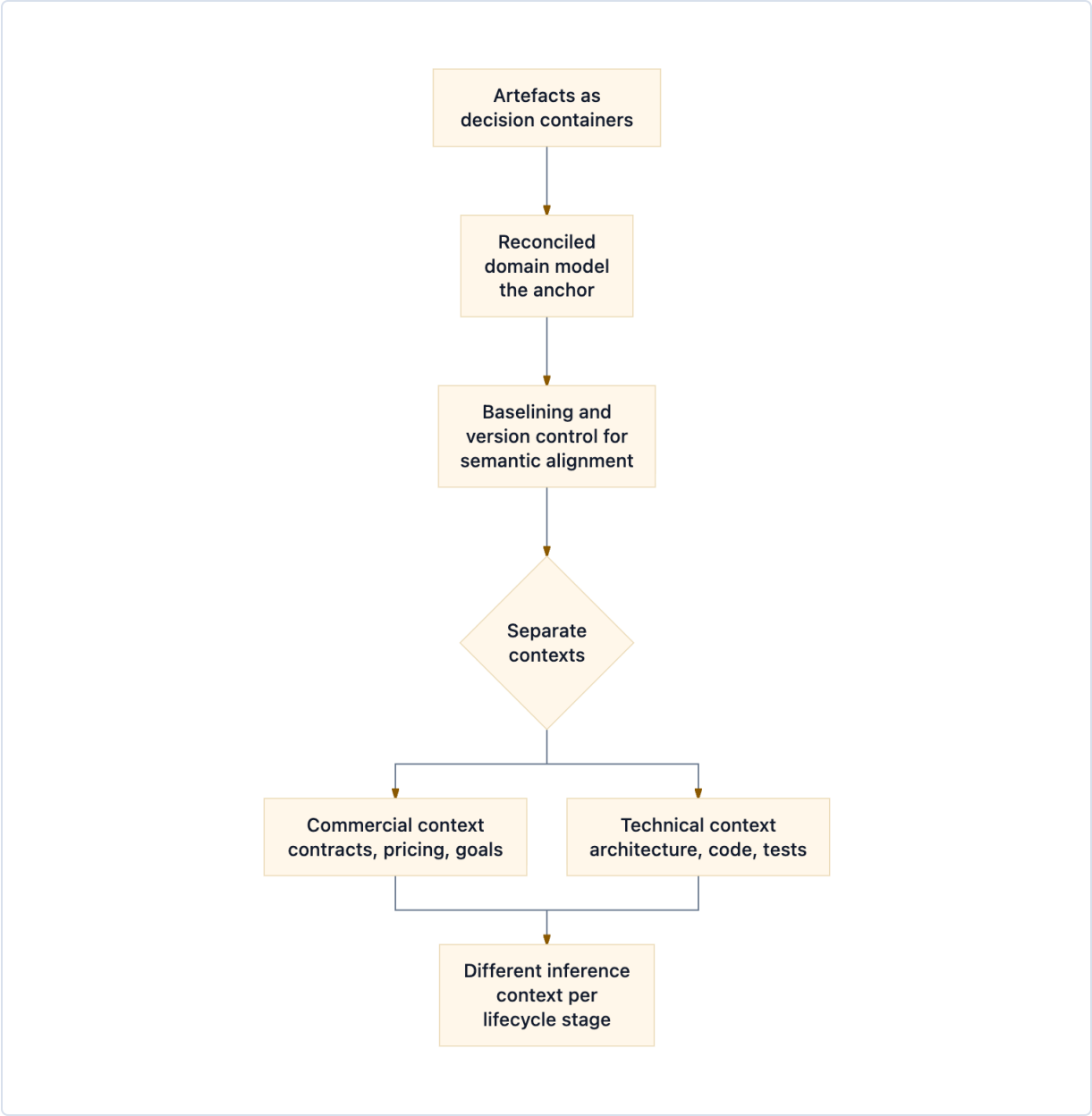


Figure 4.1: From decision containers to the reconciled domain model, baselining, and the separation of commercial and technical context across lifecycle stages.

Treating **artefacts as decision containers** means recording the reasoning behind a specification and checking later changes against it.

Requirements documents, design documents, API specifications and runbooks describe the system. When they omit the decisions behind that description, later reviewers have to reconstruct why the design took its current form.

An artefact records what was decided, why, which alternatives were considered and what follows from the decision. Validators and stage gates use that record to check later work.

Medvedeva's course text *Ontologics and Communication* (School of Systems Management, 2021; a practitioner course, cited here for its method) sharpens this distinction. A *description* is a carrier of information, but the information is useless without an *interpretation instruction*. A description *with* an explicit instruction can be interpreted mechanically, provided the instruction is a formal schema with agreed semantics and a validator; natural-language fields keep whatever ambiguity they had. A description *without* an instruction relies on the reader's prior experience and "narrowing of the interpretation field" – a process that is "usually simply not reflected upon." Most project documentation is descriptions without interpretation instructions. Schemas, type definitions, and metadata *are* interpretation instructions. An artefact with explicit metadata and schema is deterministically interpretable; one without is ambiguous.

The logistics analogy helps explain the workflow. A shipping container has a manifest that systems read when checking its route through ports. An artefact has metadata that validators read when checking its progression through stage gates. The manifest

corresponds to the metadata; the ports to the gates; and the routing rules to the validation checks.

Consider an API specification. As a traditional document, it says "the API accepts these parameters." As a decision container, it says:

- "We decided to expose these parameters (not others) because…"
- "We considered alternative designs and rejected them because…"
- "Any change to this specification requires impact analysis"

These records support **impact analysis** by identifying decisions affected by a change, **rollback** by retaining previous decisions, and **audit** by preserving the reasoning for reviewers.

The Argumentation Structure of Decision Containers

The Toulmin model of argumentation (Toulmin, *The Uses of Argument*, 1958; as taught in the School of Systems Management, Medvedeva 2021) provides a formal structure for how these decisions should be recorded. The model identifies six components:

- **Claim**: The decision itself – "we chose composite IDs for the Order API"
- **Data/Evidence**: What supports it – "legacy systems require ID parsing"
- **Warrant**: The connection between evidence and claim – "composite IDs enable backwards compatibility"
- **Backing**: Additional support if the warrant is contested – "integration tests confirm compatibility"

- **Rebuttals**: Limitations on the claim – “this adds parsing overhead for new clients”
- **Qualifier**: The epistemic status – “this is the best option given current constraints” (not “this is the only possible approach”)

Recording only the Claim leaves its supporting argument implicit. Including all six components lets reviewers and AI systems assess the evidence, assumptions and limits of a decision.

Implementing Decision Containers

In practice, decision containers can be implemented as structured files with explicit metadata:

```
# artefact: order-api-spec.yaml
decision:
  id: D-2025-042
  date: 2025-03-15
  type: architectural    # architectural | design | operational
  status: approved       # draft | proposed | approved | superseded
  title: Order API accepts composite IDs
  author: engineering-lead
  alternatives_considered:
    - separate_order_id_and_product_id: rejected due to client
complexity
    - uuid_only: rejected due to legacy system compatibility
  implications:
    - requires composite ID parser
    - affects order service and fulfillment service
  traced_to: domain-model.yaml#entities.order
  supersedes: null       # ID of previous decision, if any
```

This structured approach is what enables the **configuration manager agent** of the Zestic framework: an AI agent whose job is to

validate artefacts against the baseline, trace the decisions they contain, and manage which context is active at each lifecycle stage. It can do that only because the artefacts carry machine-readable decisions. Chapter 20 shows a reduced form of the same role in a running system.

From Stakeholder Concerns to Services

Systems thinking starts with the system of interest, its environment and the people or organisations affected by it. **Stakeholders have concerns; those concerns drive the elicitation of needs and requirements. Requirements are mapped to services through which the solution shall fulfil them.** The domain model supplies the shared vocabulary for this chain; it does not replace the stakeholders or determine their requirements on its own.

ISO/IEC/IEEE 42010:2022 provides the architecture-description concepts of stakeholders, concerns, viewpoints and views. ISO/IEC/IEEE 29148:2018 addresses requirements engineering, including stakeholder needs, requirements definition and traceability. The service mapping used here is the book's application of those concepts, rather than a claim that either standard mandates a service-only architecture.

Record the links explicitly:

Link	What the project records

Link	What the project records
Stakeholder → concern	Who is affected, what matters to them and why
Concern → need → requirement	The desired outcome, constraints and an agreed, checkable obligation
Requirement → service	Which externally provided behaviour must satisfy the requirement
Service → solution elements	Which components, data, people or processes realise that behaviour
Requirement → evidence	How satisfaction will be demonstrated and who accepts the result

For example, an account holder is concerned about losing money through a rejected transfer. Requirement R-TR-01 states: **"The transfer service shall leave the source account balance unchanged when a transfer is rejected for insufficient funds."** Allocate it to the transfer service, record its dependencies on account data and transaction handling, and verify rejection and an unchanged balance in the acceptance scenario in Appendix G. Stakeholder validation then checks whether that behaviour addresses the original need in the intended operating context.

A service here means a capability or behaviour offered by the system, not necessarily a separately deployed software service. The links can be many-to-many: one concern can drive several requirements, and several services can jointly satisfy one requirement. System-wide constraints, such as availability or privacy, also need allocation across the relevant solution elements. Record conflicts and trade-offs rather than treating stakeholder agreement as automatic. When a concern

or requirement changes, trace the impact through services, decisions and verification evidence, then revisit stakeholder acceptance.

Intended Use and Consequences Beyond the Stakeholder List

A concern states what matters to a stakeholder. Intended use identifies the proposed situation, expected effects and conditions. A requirement states an agreed obligation. Keep these distinct: recording a concern does not approve a requirement, and approving a requirement does not demonstrate its fulfilment.

The synthetic maintenance case introduced earlier gives these links a concrete subject. A service owner wants a failed scheduled job restored while preserving jobs that already work. The initial use is a routine repair with a known owner and a defined execution scope. Unknown ownership remains a reason for manual review. Another team shares the execution quota; additional retries could consume its capacity. That is a possible consequence to investigate, not an observed harm established by the story.

Case requirement	Allocated service	Evidence needed
R-USE-01: the maintenance service shall report restoration only after recording successful completion and required delivery for the identified job.	Maintenance	Job, completion and delivery records referring to the same run; unresolved delivery remains visible.

Case requirement	Allocated service	Evidence needed
R-EVAL-01: the evaluation service shall assess each candidate against frozen triggering and protected cases before promotion.	Evaluation	Candidate-specific results, including preservation of manual review for unknown ownership.
R-BOUND-01: the execution service shall withhold an action unless its scope and resource use can be established to remain within approved per-run and cumulative limits.	Execution	Scope and quota checks, including concurrent work, exhausted limits and unavailable quota evidence.

These requirements and their proposed checks are teaching fixtures. Appendix H extends the matrix; Chapters 15–17 develop the assessment, permission and trace contracts. A shared quota may require reservation or another enforceable allocation mechanism. Stating a limit in an instruction does not enforce it. Where the organisation cannot bound exposure within its approved risk appetite, the assistant must stop or propose a change for accountable review.

The current Systems Engineering DPF, SYSE.2 develops use claims and system concepts together. SYSE.17 investigates consequence bearers beyond stakeholder lists and organisational boundaries. The application here is to revisit the repair design when shared-resource exposure changes the use assumptions, rather than forcing the original requirements to fit the first solution.

Revisions need authority as well as evidence. A service owner can challenge whether a requirement still meets the need; the applicable owners decide changes to requirements, risk limits and permissions. The assistant cannot widen its own scope. Retain the previous baseline, affected links, decision and new evidence. Until an authorised revision takes effect, the current requirements and limits still govern operation.

Concept Modelling Fundamentals

Chapter 1 introduced entities, relationships and constraints as the vocabulary of meaning. The domain model is built from them, so they need to be precise before the model is. Chapter 8 applies them in practice.

Entity Types

In the domain model, entities are typed. The four types below follow Evans's *Domain-Driven Design* (2003), which this chapter adopts rather than invents:

Type	Definition	Example
Core entity	A business object with identity that persists through change	Customer, Order, Product
Value object	An immutable object defined only by its attributes	Money, Address

Type	Definition	Example
Event	Something significant that happened, with a time	OrderPlaced, PaymentReceived
Aggregate	A cluster of related entities with one consistency boundary	Order with its line items

A worked example holds the three fundamentals together. Entity: Order. Relationship: an Order contains one or more LineItems (one-to-many). Invariant: the Order's total equals the sum of its LineItems' prices. A domain model that records all three lets the validator of Chapter 14 reject an order whose total is wrong, and lets the grounding gate of Chapter 9 reject an answer that claims an order has no line items.

Discovering a domain model is a short procedure: list the nouns stakeholders use; for each, decide its type from the table above; for each pair that interacts, write the relationship and its cardinality; for each rule anyone states as "always" or "never", write it as an invariant; then reconcile the result with every stakeholder before it is baselined.

Relationship Cardinalities

Relationships between entities have cardinalities:

- **One-to-one**: Customer has one Profile
- **One-to-many**: Customer places many Orders
- **Many-to-many**: Product appears in many Orders

These cardinalities matter for AI context because they define what can be assumed. If the AI assumes one-to-many when it's actually many-to-many, it will generate incorrect outputs.

Invariants

Invariants are rules that must always be true:

- "Order total must equal sum of line items"
- "Dosage must be positive"
- "Customer must have verified email"

Invariants become safety-gate inputs when a validator checks them (Chapter 14). Without explicit invariants, there is nothing for a gate to check, and the AI can violate constraints that humans assume are obvious.

The Reconciled Domain Model as Anchor

The **reconciled domain model** records agreed definitions of the entities, relationships, rules and constraints in the problem space.

Every major artefact must trace to the domain model. The Zestic framework states the requirement as follows:

> "At the minimum, every major artefact must trace back to it (i.e., be part of the configuration baseline): business scenarios, process models, data schemas, APIs, AI use cases, and UX flows. This

> ensures semantic coherence across each stage of the software development life cycle." (Turkhanov, 2025)

What a Domain Model Contains

A practical domain model for an AI system includes:

- **Entities**: The key objects in the domain (Customer, Order, Product, Drug, Dosage)
- **Relationships**: How entities relate (Customer places Order, Order contains Products)
- **Business rules**: Inviolable constraints (Dosage must be positive, Contraindication applies when…)
- **State transitions**: How entities change over time (Order placed → Order shipped → Order delivered)
- **Events**: Significant occurrences (OrderPlaced, PaymentReceived, ShipmentDelivered)

Reconciliation Process

Reconciliation checks the domain model for consistency across stakeholders and artefacts. Repeat these checks when the model or its dependencies change:

1. **Stakeholder alignment**: All parties agree on the definitions
2. **Artefact validation**: Every artefact is checked for consistency with the model

3. **Coherence checking**: The model itself is checked for internal consistency
4. **Baseline locking**: At each stage gate, the model is frozen and versioned

Without reconciliation, different parts of the organisation develop different understandings. One team uses "product" to mean the item purchased; another uses it to mean the service subscription. These seemingly small misalignments compound into system-level failures.

A foundational principle from the Russian systems engineering tradition captures why: "Synchronisation of the consequence in actions without synchronisation of the explanation is often a mistake that yields a short-term result but leads to significant losses in the long term" (Medvedeva, School of Systems Management, 2021). In context engineering terms: if two teams agree on *what to build* without agreeing on *why* – without synchronising their explanations of the problem – they will diverge as soon as they encounter an undocumented edge case. Reconciliation is the process of synchronising explanations, not just actions.

Baselining and Version Control for Semantic Alignment

The concept of **baselining** comes from traditional configuration management. A baseline is a formally approved version of something that serves as a reference for subsequent work.

In Context Engineering, baselining applies to:

- The domain model (what entities and rules are approved)
- The vocabulary (what terms mean)
- The prompts (what instructions are approved)
- The knowledge base (what facts are approved)
- The configuration (what settings are approved)

> "A baseline should lock terminology, scope, assumptions, and define what is out of scope. Configuration management ensures version control, traceability of changes, and controlled evolution of artefacts. Without baselining, artefacts become informal references. With baselining, they become governance instruments." (Turkhanov, 2025)

Semantic Versioning for Context

Just as software uses semantic versioning (major.minor.patch), the author's practice applies **semantic versioning to meaning**; it is a convention proposed here, not an established standard:

- **Major**: Breaking change to domain model (entity removed, relationship changed)
- **Minor**: Addition to domain model (new entity, new relationship)
- **Patch**: Non-breaking change (documentation, examples)

This supports **impact analysis**: a major version bump signals that dependent artefacts may need review. A patch label records the

intended compatibility of a change; it does not establish safety or waive the checks required for the affected use.

The author's baseline checklist has five steps: (1) tag the domain model file with its version; (2) record that version in every artefact validated against it; (3) configure the Chapter 14 validator to fail artefacts whose recorded version is behind the current major; (4) require change requests after the lock, naming affected artefacts; (5) record the lock date and approver in the model's lifecycle metadata.

Separating Commercial Context from Technical Context

The second architectural decision is to keep **commercial context and technical context in separate stores with separate owners**, because the two change for different reasons and at different rates, and a model given both without labels cannot tell a promise from a capability.

Commercial Context

Commercial context includes:

- Business objectives and priorities
- Customer commitments and expectations
- Budget and timeline constraints
- Go-to-market requirements
- Competitive pressures

This context lives in sales conversations, contract terms, slide decks, and executive priorities. It changes based on market conditions and customer relationships.

Technical Context

Technical context includes:

- System capabilities and limitations
- Available tools and integrations
- Data schemas and API contracts
- Model configurations and parameters
- Validation rules and constraints

This context lives in code, specifications, and configuration files. It changes based on engineering decisions.

Why Separation Matters

When commercial and technical context are mixed:

- Account managers commit to features without feasibility validation
- Engineers are asked to “make it work” regardless of architecture
- Scope creep happens through informal commitments
- Technical debt accumulates invisibly

When they are separated:

- Engineers can focus on technical feasibility
- Commercial stakeholders have clear escalation paths
- Change requests require formal impact analysis
- Technical integrity is maintained

The OpenClaw system (Chapter 20) demonstrates the separation at small scale. The user's priorities and standing commitments live in USER.md and the memory files; the skills, tool definitions and loader rules live in the repository alongside them but are edited through a different process. When a commitment asks for something the skills cannot do, the gap becomes a change to a skill, reviewed as code, rather than a quiet rewording of the commitment to fit what the skills can do.

In practice the separation needs three things: two stores with different owners (commercial context in the decision containers the account owner maintains, technical context in the repository the engineers maintain); a single crossing point, the change request, through which a commercial commitment becomes a technical requirement after impact analysis; and a gate that refuses artefacts tracing to a commercial decision that has no corresponding technical decision. Without the third, the separation is a filing convention.

Why Each Lifecycle Stage Requires a Different Inference Context

Each lifecycle stage needs a subset of context selected for the decisions being made at that stage.

The Four Stages

Stage	Context includes	Decisions are about
Discovery	Market research, customer interviews, competitor analysis, initial domain model	What to build
Design	Refined domain model, architecture decisions, technical constraints, API contracts	How to build it
Development	Implementation detail, code, tests, integrations	Whether the build is correct
Deployment	Configuration, monitoring, rollback procedures	Whether the system is running correctly

Why Context Must Change

Each stage has a **decision space**: the decisions being made and the evidence needed for them. Discovery usually needs little implementation detail; development usually needs the approved requirements rather than the full market-research corpus.

By bounding context to the decision space:

- The model receives material relevant to its current decisions.
- Superseded decisions can be excluded from active context.

- Each gate checks the context required for the next stage.

The configuration manager agent defined earlier manages these transitions by selecting the context required for each stage's decisions.

Roles in Context Governance

The formal model of roles taught by Medvedeva (School of Systems Management, 2021) provides a precise framework for context partitioning. It builds on the stakeholder, concern and viewpoint concepts of ISO/IEC/IEEE 42010 and adds two components of its own; the triad below is Medvedeva's, not the standard's. A **role** is a functional part of the system, defined by three components:

- **Concern**: the objects that attract the role's attention – what it notices from the background
- **Preference**: what the role wants to happen with those objects
- **Intention**: the strategies the role plans to use

Here, "concern" uses Medvedeva's attention-based formulation. In architecture description, a stakeholder concern is an interest relevant to the entity and its architecture; it need not be a problem or a worry. Retain those concerns as sources for requirements, rather than reducing them to vocabulary filters.

An engineer's concern is system integrity; their preference is architectural coherence; their intention is to enforce constraints. An account manager's concern is client satisfaction; their preference is

feature delivery; their intention is to commit to timelines. These are structurally different roles with different contexts. Reconcile their requirements and record trade-offs; a change of active role does not remove another stakeholder's concerns.

This formal role model explains why each lifecycle stage requires different context: the *active roles change*. During discovery, the product owner's concern/preference/intention dominates. During development, the engineer's does. During deployment, the operator's does. Context engineering partitions context along role boundaries, not just temporal ones. This connects directly to the role-based graphs in Chapter 7, where the same knowledge graph provides different synonym edges and relationships depending on which role is querying it.

Chapter 17 extends this decision container into execution records: requirement and allocated service identifiers link to observed evidence, the assessed artefact and the authorised action. Design-time rationale explains the choice of structure; execution evidence records what occurred. A stated rationale remains a claim to inspect.

Summary: Architecture Enables Discipline

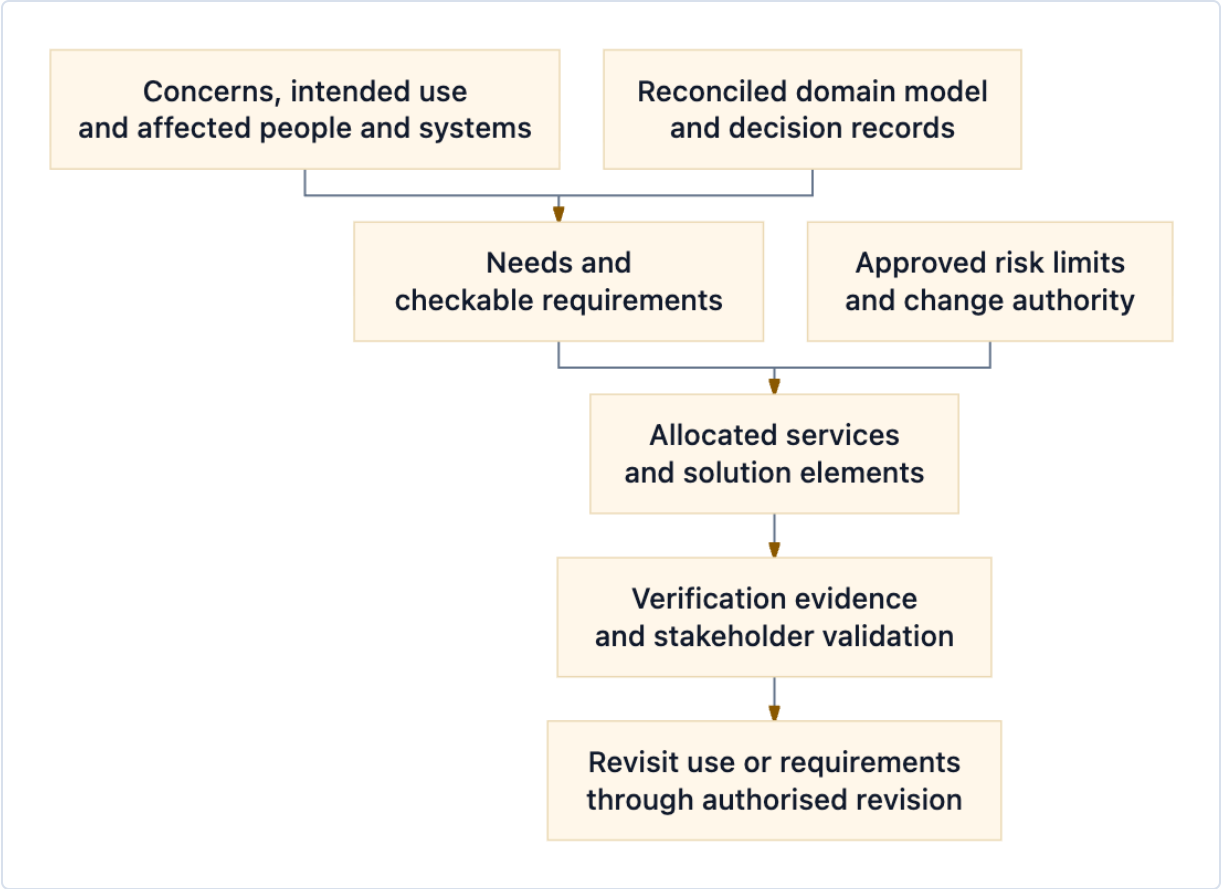


Figure 4.2: Chapter 4 recap: Stakeholder concerns drive requirements and service allocation; shared meaning and baselines keep the links reviewable.

The architecture of meaning provides the structure that makes Context Engineering practical:

1. Decision containers record reasoning and support impact analysis.
2. Entities, relationships and invariants provide the modelling vocabulary.
3. The reconciled domain model gives artefacts agreed definitions to reference.
4. Baselines record the approved context at each stage.

5. Separate owners and change controls distinguish commercial commitments from technical capabilities.
6. Stage-specific loading limits active context to the decisions being made.
7. Stakeholder concerns drive requirements, which are allocated to services and traced through solution elements to evidence of satisfaction.

In the maintenance case, this chapter supplies the use conditions, service allocation and revision authority that later checks need. We have agreed what restoration means and identified a possible consequence for another team. The assistant still needs a reference it can query. Chapter 5 turns the definitions and relationships into reviewed knowledge; Chapters 8 and 9 show how to maintain and check it.

Key Sources

- Alexander Turkhanov, "Context Engineering: The Missing Discipline in AI Architecture", Zestic AI blog, https://zestic.ai/blog/context-engineering-missing-discipline-ai-architecture (2025)
- Eric Evans, *Domain-Driven Design* (2003): entities, value objects, aggregates, bounded contexts
- Stephen Toulmin, *The Uses of Argument* (1958)

- IEEE 828-2012, *Standard for Configuration Management in Systems and Software Engineering*
- ISO/IEC/IEEE 42010:2022, *Software, systems and enterprise — Architecture description*: stakeholders, concerns, viewpoints and views
- ISO/IEC/IEEE 29148:2018, *Systems and software engineering — Life cycle processes — Requirements engineering*: stakeholder needs, requirements and traceability
- OpenClaw context architecture (practical implementation)
- Medvedeva, "Creating Explanations" (Ontologics and Communication, School of Systems Management, 2021) – Toulmin argumentation, synchronisation principle
- Medvedeva, "Complex Descriptions and Meta-Languages" (Ontologics and Communication, School of Systems Management, 2021) – descriptions, interpretation instructions, roles (concern/preference/intention), ISO 42010 viewpoints
- "Ontology in Plain English" (Russian systems engineering community; translated), cited in Chapter 1 for entity-type misassignment
- Anatoly Levenchuk, *FPF*, revision `0c6ade275e9360f0c5ab9715d8f05c0dbaa13cf8` (8 October 2026), Systems Engineering DPF, SYSE.2 and SYSE.17, linked above: intended use, co-evolving concepts and qualified consequences; editorial adaptation, not a claim of framework conformance

Chapter 5: Knowledge Graphs Explained

The domain model from Chapter 4 can be represented as a **knowledge graph**: entities and relationships that a system can query and check against their sources. This chapter explains the structure, its use in retrieval and the curation it requires.

In the synthetic maintenance case, the reference can distinguish a job, its owner, a candidate rule, a quota and a delivery observation. Typed links express which rule applies and which jobs share a resource. The graph holds reviewed claims about those relationships; it does not perform a repair or make a predicted consequence an observed fact. Use this representation where explicit relationships aid the decision, alongside other sources where they are more suitable.

What Is a Knowledge Graph?

A **knowledge graph** is a structure for representing knowledge as a network of **entities** (nodes) and **relationships** (edges). Knowledge graphs are themselves databases – specifically, databases whose schema emphasises relationships between entities. Where relational databases store data in tables and document stores hold unstructured text, knowledge graphs capture the *meaning* of information by explicitly stating how things relate.

The Basic Structure

At its simplest, a knowledge graph consists of:

- **Nodes** (also called vertices): Represent entities, concepts, or things
- **Edges** (also called relationships or arcs): Represent how nodes relate to each other
- **Properties** (also called attributes): Describe nodes and edges

Figure 5.1 shows two typed relationships.

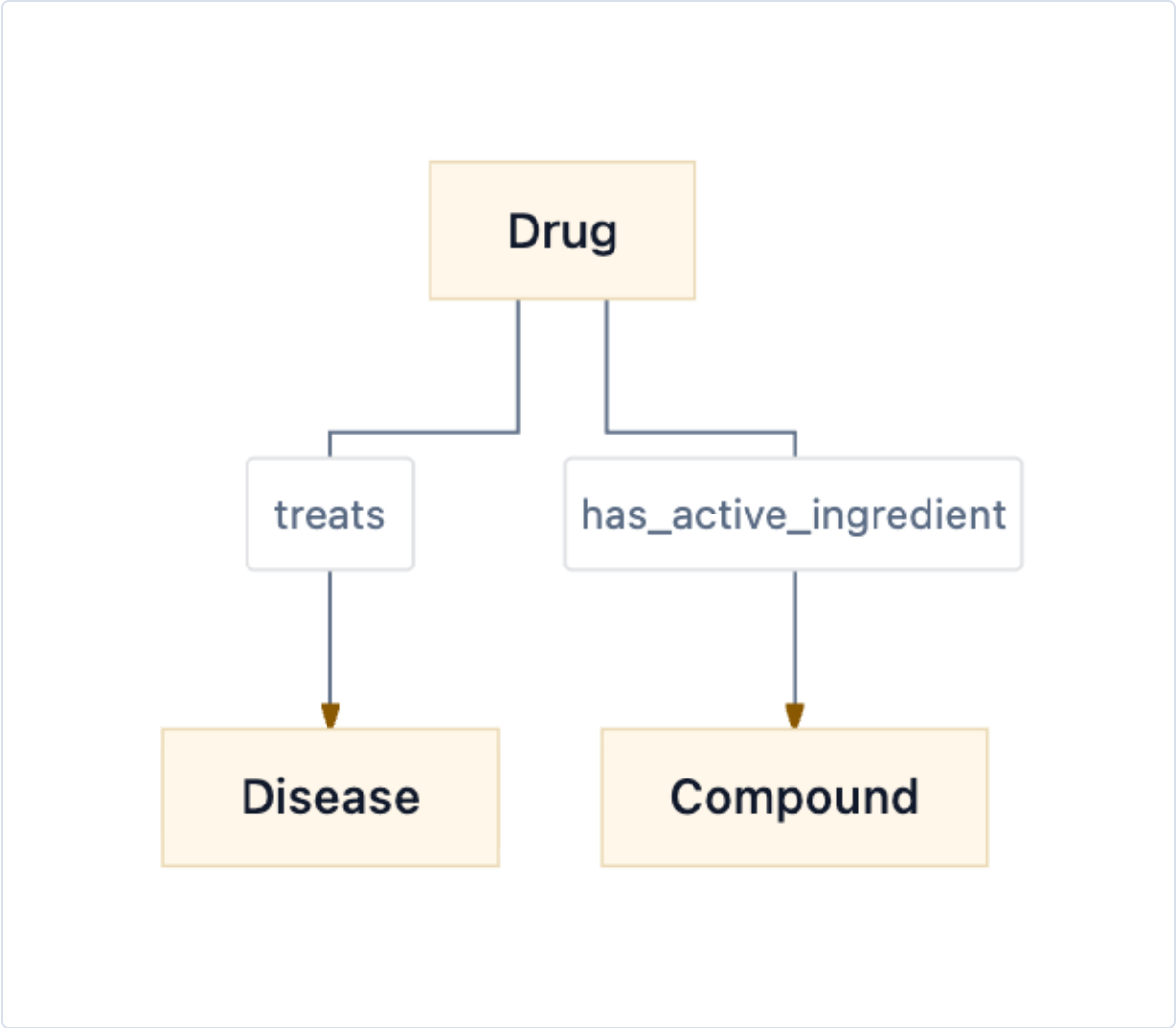


Figure 5.1: A generic knowledge graph links a drug to a disease through treats and to a compound through has_active_ingredient.

In this tiny illustrative graph (the edge types are a generic example, not the vocabulary of any particular terminology): - “Drug” and “Disease” are nodes - “treats” is an edge from Drug to Disease - “has_active_ingredient” is an edge from Drug to Compound

For a query such as “What does Drug X treat?”, the system can follow the stored treatment relationships and return their sources.

Triples: The Atomic Unit

The fundamental unit of a knowledge graph is the **triple**: subject-predicate-object.

Figure 5.2 compares the triple pattern with a concrete example.

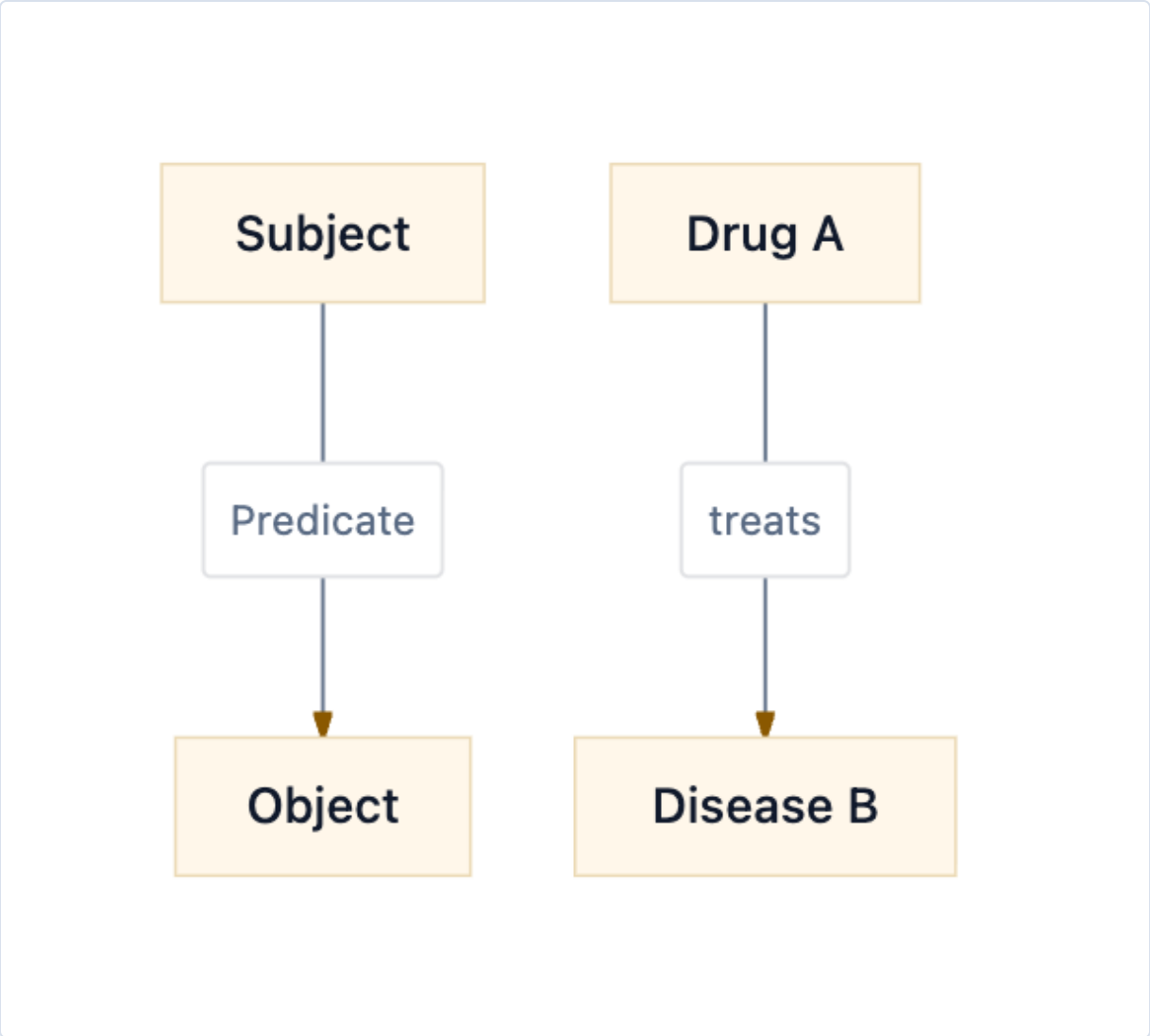


Figure 5.2: A triple names a subject, a relationship and an object; the example states that Drug A treats Disease B.

The triple representation lets software query relationships and trace them to their sources.

Real-World Example: SNOMED CT

One of the largest curated vocabularies in healthcare is **SNOMED CT** (Systematized Nomenclature of Medicine, Clinical Terms). The International Edition holds about 350,000 active concepts and over a million relationships (SNOMED International, International Edition release notes, 2024); counts in this chapter use that release.

SNOMED CT is not a document; it is a graph. It is a clinical terminology: its relationships are mostly *is-a* hierarchies (a myocardial infarction is a heart disease) and defining attributes (a procedure has a site, a disorder has a causative agent). It does not record which drugs treat which diseases or which are contraindicated; in the pipelines of Chapter 9 those edges come from other sources, PrimeKG for drug-disease and drug-gene relationships and CPIC guidelines for pharmacogenomics. When a medical AI queries this combined graph, it is not searching text. It is traversing relationships:

- "Is this finding a kind of heart disease?" traverses SNOMED CT's is-a edges
- "What is the active ingredient of Drug X?" traverses a has-active-ingredient edge
- "What are the contraindications for Drug Y?" traverses PrimeKG's contraindication edges

These stored edges give the system an explicit retrieval path for each relationship.

Historical Context

Google's Knowledge Graph

Google introduced its **Knowledge Graph** in 2012. A search for "Leonardo da Vinci" could display a facts panel with nationality, birth date, works and contemporaries alongside web results.

The panel presented structured facts about an entity alongside links to documents about it.

By 2024, Google's Knowledge Graph powered the structured information panels appearing alongside search results, answering questions directly without requiring users to click through to websites. The exact scope varies by query category, but Google describes the Knowledge Graph as central to its search strategy.

Enterprise Ontologies

Before Google, enterprises had been building ontologies for decades:

- **SNOMED CT** in healthcare (2002-present, with SNOMED predecessors dating from 1965)
- **Dublin Core** in library science (1995-present)
- **RDF** (W3C, 1999-present) and **OWL** (W3C, 2004-present)
- **TopBraid**, **Protege**, and other ontology tools

These efforts showed that knowledge graphs could represent complex domains reliably. But they were expensive to build and

maintain, and they did not integrate well with the statistical AI approaches that dominated the 2010s, so outside a few regulated domains they receded. Two things changed in the 2020s: language models made the expensive step, extracting candidate entities and relationships from text, cheap, and the same models turned out to need something to be grounded against. The graph became both easier to build and more necessary.

The Resurgence in the AI Era (GraphRAG 2024-2025)

RAG (Retrieval-Augmented Generation) supplies a model with relevant documents in its prompt, giving it reference material beyond its training data.

But naïve RAG had limitations:

- **Document retrieval is probabilistic**: “vectors are close” doesn’t guarantee relevance
- **No explainability**: You can’t explain *why* a document was retrieved
- **No validation**: The model can ignore retrieved documents
- **No grounding**: The model can hallucinate even when given correct context

In 2024, Microsoft introduced **GraphRAG**, an approach that uses knowledge graphs to structure retrieval. It has two halves that are

easy to conflate. The indexing half is partly probabilistic and partly deterministic: a language model extracts entities and relationships from the corpus, then a graph algorithm (Leiden community detection) clusters them, and a language model writes a summary for each community. The query half retrieves over that structure instead of over raw chunks. So GraphRAG grounds answers in a graph, but the graph and its summaries are themselves model-generated, which is why Chapter 8 argues for vocabulary-first construction where the stakes are high.

Naive RAG	GraphRAG
Retrieve document chunks	Retrieve entities, relationships and community summaries
Similarity over embeddings	Traversal over an extracted graph
“Vectors are close”	“Drug treats Disease” (as the extractor read it)
No trace	A path through the graph
Model may ignore the context	Model answers over structure, but the structure was model-built

Why Knowledge Graphs Matter for AI Context

In the context engineering framework, knowledge graphs support the following functions:

1. **Semantic backbone**: The domain model (from Chapter 4) is implemented as a knowledge graph

2. **Precise retrieval**: Queries traverse relationships, not just match keywords
3. **Validation gate**: Outputs can be verified against the graph
4. **Explainability**: Every answer can be traced through the graph
5. **Grounding**: claims the gate can see are checked against explicit constraints, which catches confabulation where the vocabulary covers it

Knowledge Graphs vs Vector Databases: Complementary Approaches

Figure 5.3 summarises the complementarity this section argues for.

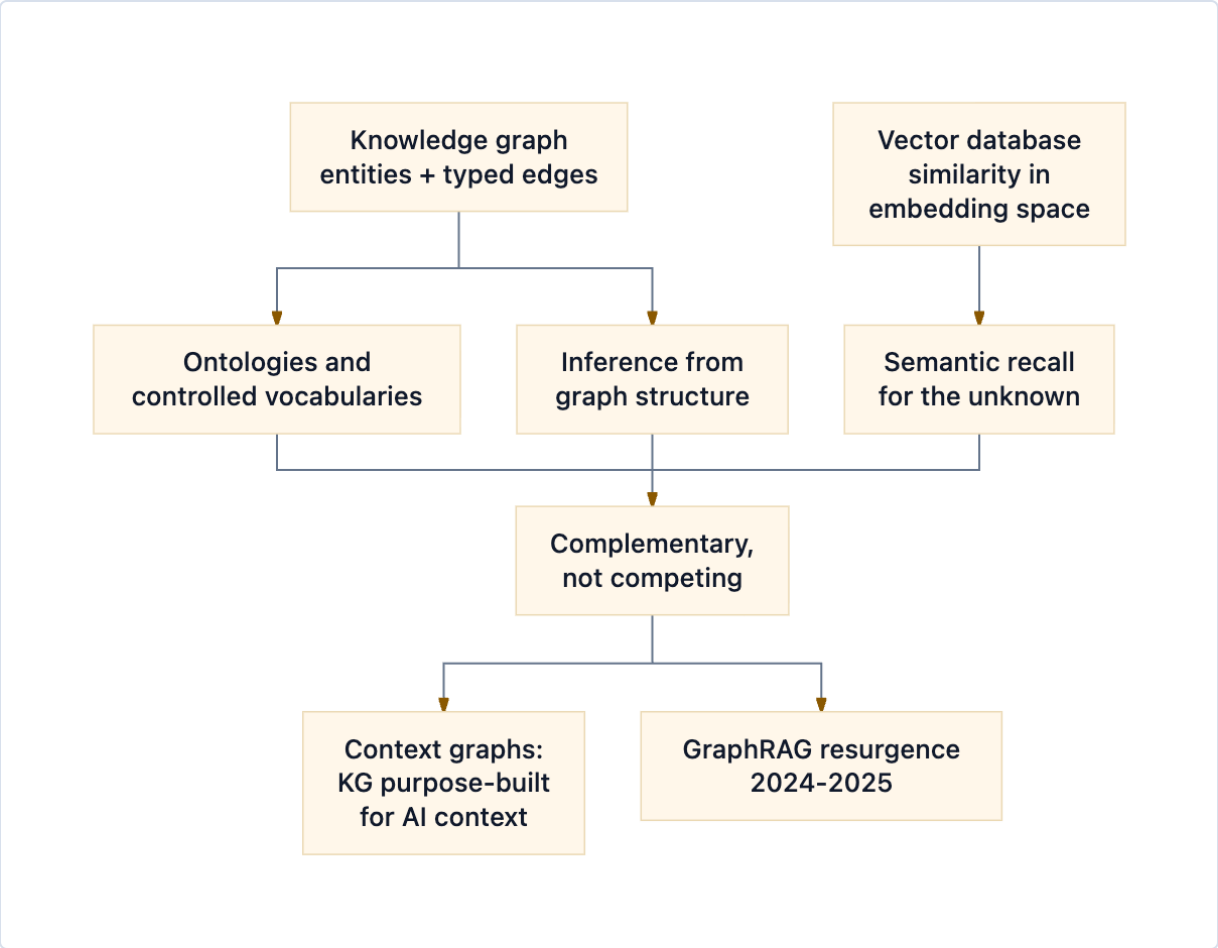


Figure 5.3: Knowledge graphs and vector databases are complementary: typed structure and inference on one side, semantic recall for the unknown on the other, converging on context graphs.

Knowledge graphs and vector databases support different kinds of query. A system may need both.

How Vector Databases Work

Vector databases store **embeddings**, numerical representations of documents or text chunks. Queries retrieve the vectors closest to the query vector under a measure such as cosine similarity or dot product.

Figure 5.4 follows a query through vector retrieval.

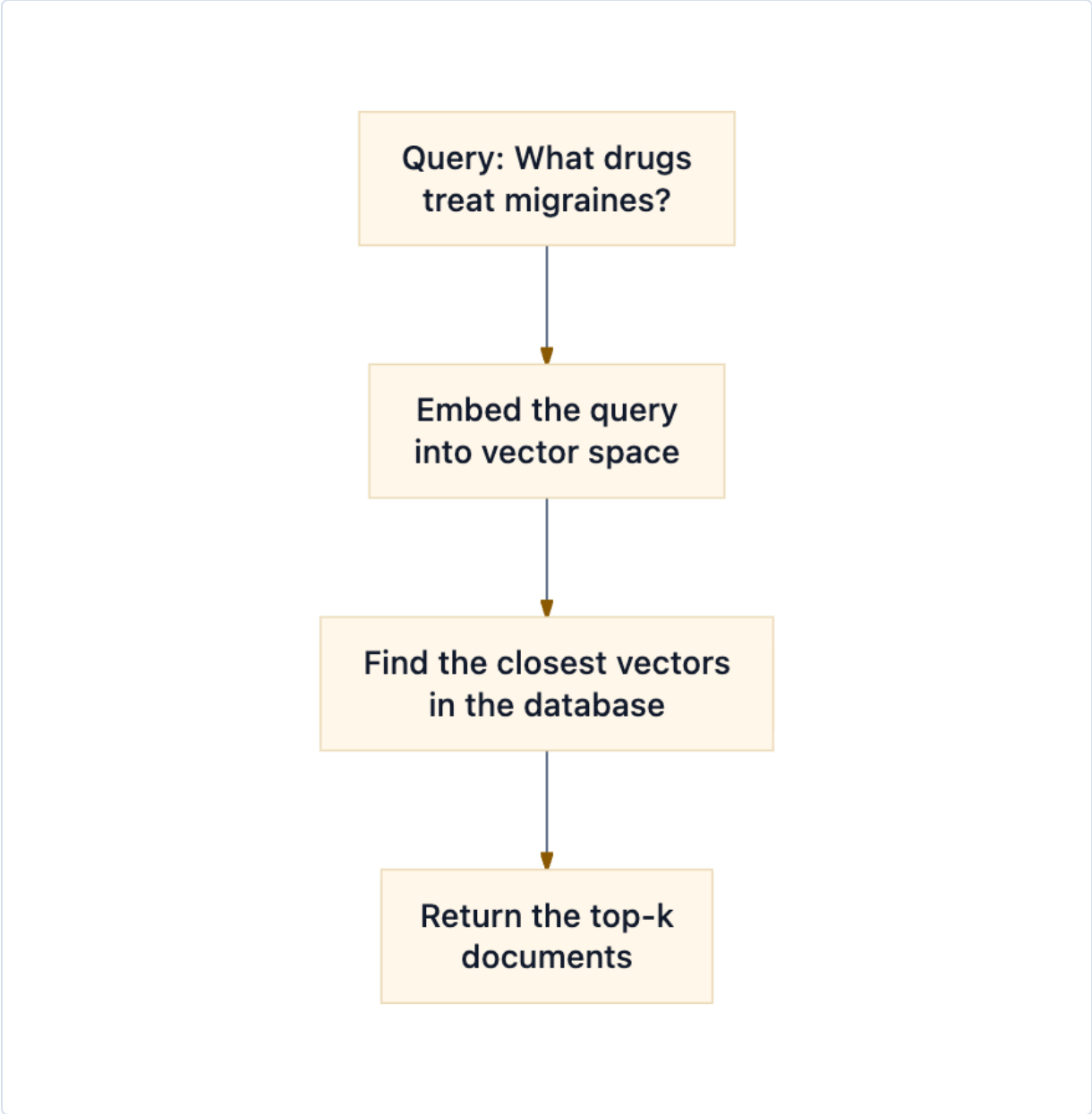


Figure 5.4: Vector retrieval embeds the query, finds nearby vectors and returns the corresponding top-k documents.

A high similarity score indicates that the texts are related under the embedding model. It does not establish that Drug A treats Migraine.

Strengths and Limitations

Aspect	Vector Databases	Knowledge Graphs
Query type	Similarity search	Relationship traversal
Precision	Probabilistic	Deterministic
Explainability	Low (why those vectors?)	High (trace the path)
Maintenance	Easy (add embeddings)	Hard (curate relationships)
Coverage	Broad (any text)	Deep (explicit facts)
Inference	Limited	Transitive reasoning, inheritance, rules

When to Use Each

Use vector databases when: - You have large volumes of unstructured text - Queries are open-ended (“What information do you have about X?”) - Recall matters more than precision - You can’t enumerate all possible queries

Use knowledge graphs when: - The domain has explicit structure (entities, relationships, rules) - Precision and explainability matter - You need validation gates - The domain is well-defined (e.g., healthcare, legal, finance)

Use both when: - You need the breadth of vector search AND the precision of graph traversal - Open-ended queries go to vector DB; specific queries go to knowledge graph - The knowledge graph validates and filters vector search results

This hybrid approach is covered in Chapter 22.

Before building anything, four checks decide whether a domain is ready for a graph: a vocabulary exists or can be written (if nobody can list the fifty most important terms, start there, Chapter 8); the facts that matter have authoritative sources the graph can cite; someone owns the vocabulary and will curate it; and the cost of a wrong answer justifies the curation. A domain that fails the first or third check should start with a thesaurus and a grounding gate rather than a full graph.

The Role of Ontologies and Controlled Vocabularies

A **controlled vocabulary** is a curated list of terms with agreed definitions. It gives participants a definition to check when, for example, they use the term “Patient” differently.

An **ontology** defines terms and their relationships. SNOMED CT is both a controlled vocabulary and an ontology.

Why Controlled Vocabularies Matter for AI

Without controlled vocabularies, the AI system must infer meaning from text. With controlled vocabularies, meaning is explicit:

Without Controlled Vocabulary	With Controlled Vocabulary
“patient”, “pt”, “subject”, “client”	Single term: Patient

Without Controlled Vocabulary	With Controlled Vocabulary
“treats”, “indicated for”, “therapy for”	Single relationship: treats
AI infers meaning	AI traverses relationship

Terraphim’s **synonym expansion** maps listed variants to canonical terms, so that different surface forms retrieve the same concept.

The Thesaurus Pattern

A **thesaurus** defines synonyms and related terms:

```
{
  "term": "myocardial infarction",
  "synonyms": ["heart attack", "MI", "cardiac infarction"],
  "related": ["coronary artery disease", "angina"],
  "broader": ["cardiovascular disease"],
  "narrower": ["STEMI", "NSTEMI"]
}
```

When the AI receives a query with “heart attack,” it maps to “myocardial infarction” and traverses the graph from there. This is the core of Terraphim’s approach.

Case Study: SNOMED CT in Healthcare AI

Figure 5.5 gives the scale of the vocabulary and the results of grounding against it.

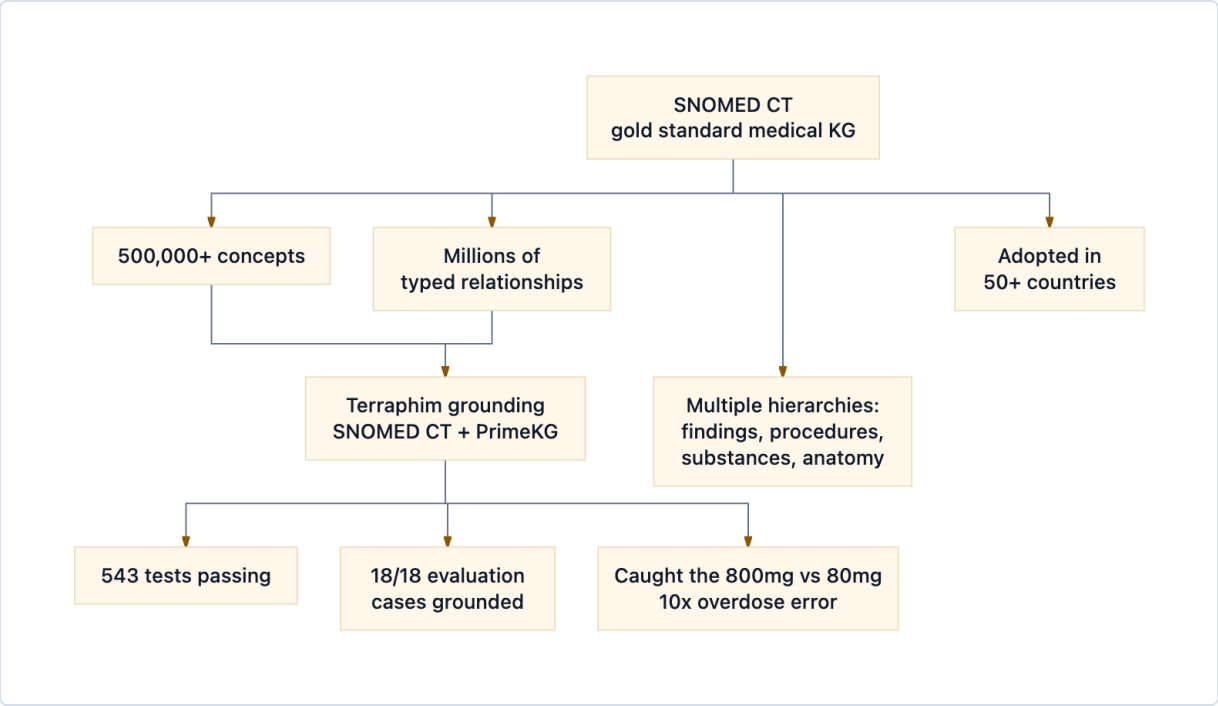


Figure 5.5: SNOMED CT at scale, and what grounding a medical model in it produced in the MedGemma competition.

SNOMED CT illustrates the scale and curation requirements of a clinical terminology.

The Scale

- **About 350,000 active concepts** in the International Edition (2024 release)
- **Millions of relationships** (edges)
- **Multiple hierarchies** (clinical findings, procedures, substances, anatomy)
- **International adoption** in 50+ countries

The MedGemma Competition

The author's MedGemma evaluation used a Terraphim knowledge graph built on SNOMED CT and PrimeKG, with these results:

- **543 automated tests passing**: component tests over extraction, graph query, guideline lookup, gate decision and trace generation, as Chapter 9 defines them
- **18 of 18 evaluation cases grounded**: every entity in each final recommendation matched the graph and the recommendation agreed with the reference; this is a result on 18 author-chosen cases, not a general hallucination rate
- Specific drug-dose recommendations instead of class-level suggestions.
- One tenfold dosing error caught: the raw model's 800mg of osimertinib was replaced by the licensed 80mg. This is one case out of eighteen, not a measured error rate.

Chapter 9 describes how the graph and grounding gate produced these outcomes, including the dosing correction and the evaluation's limits.

Inference: The Hidden Power of Graph Structure

One capability that distinguishes knowledge graphs from simpler data stores is **inference** – deriving new knowledge from existing relationships without storing it explicitly.

Transitive Reasoning

If the graph states:

Figure 5.6 distinguishes stored edges from a derived relationship.

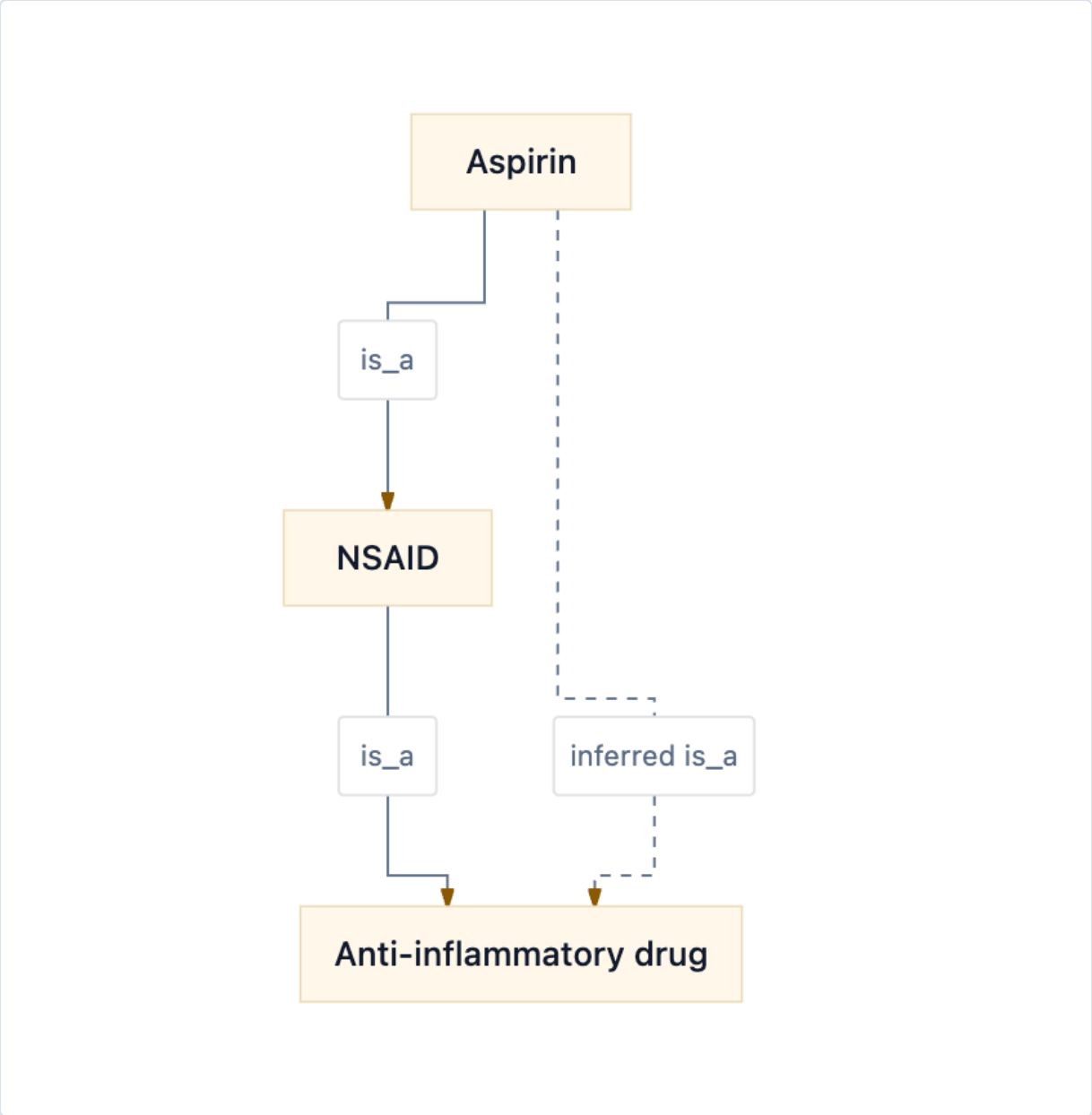


Figure 5.6: Two stored is_a edges connect Aspirin to NSAID and NSAID to Anti-inflammatory drug; their transitive consequence is shown as a dashed edge.

Then the system can infer that Aspirin is an Anti-inflammatory Drug, even though that relationship is not explicitly stored. This transitive reasoning across `is_a` hierarchies is fundamental to ontology-based systems like SNOMED CT.

Inheritance

Properties defined on a parent class propagate to children:

Figure 5.7 keeps the class-level relationship separate from clinical confirmation.

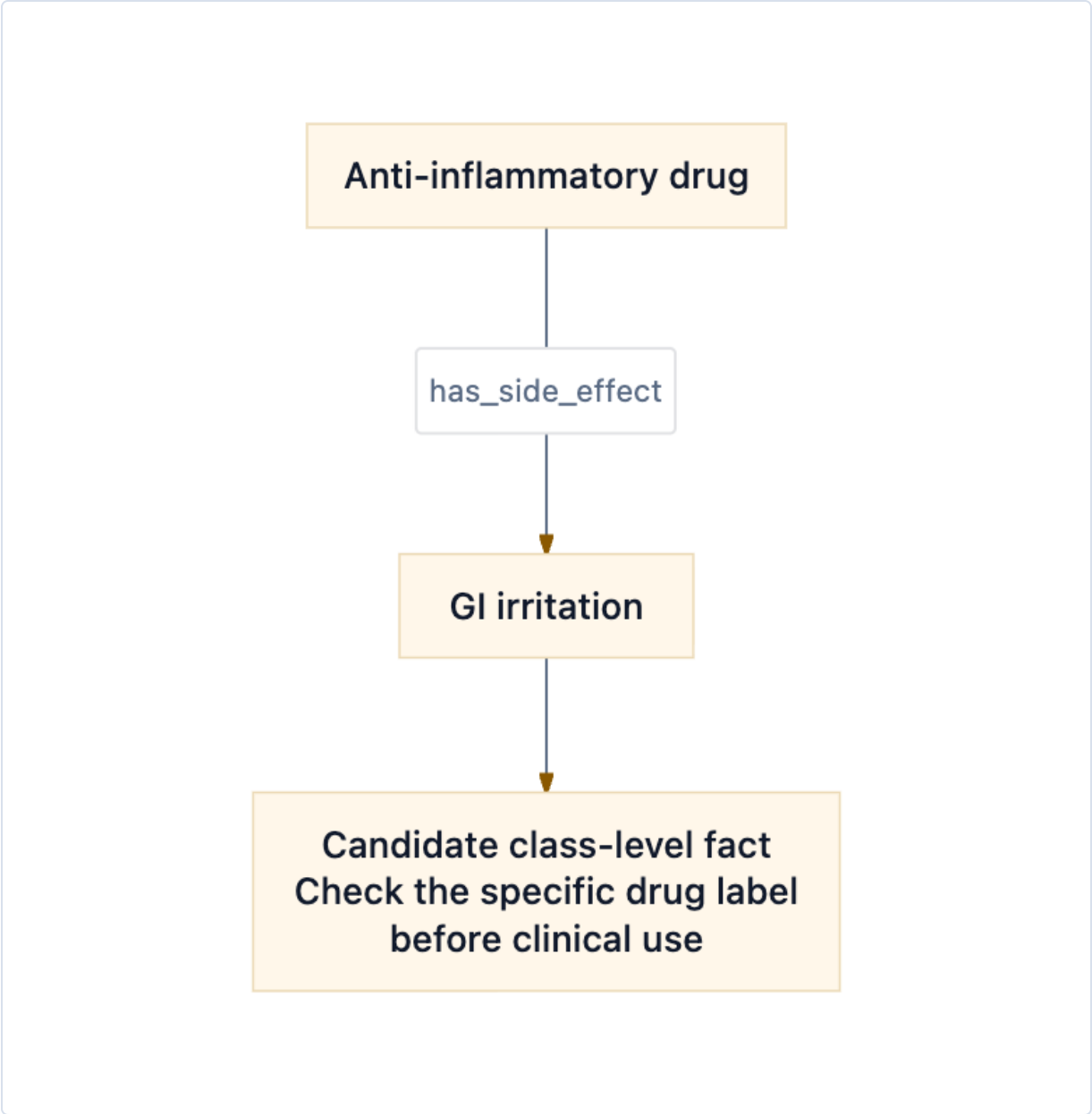


Figure 5.7: A class-level side-effect relationship supplies a candidate fact; a clinical system must check the specific drug label before using it.

Any drug classified under Anti-inflammatory Drug inherits this side effect as a *candidate*: inheritance tells the system where to look, and a clinical system must still confirm the class-level fact against the drug's own label before acting on it. Outside medicine the same mechanism is less fraught: a product taxonomy in which every item

under Perishable inherits a shelf-life rule is inheritance doing exactly what it should.

Path-Based Inference

More sophisticated inference follows multi-hop paths:

A query such as “What compounds interact with drugs that treat Disease X?” crosses treatment, ingredient and interaction relationships. Figure 5.8 shows the stored edge directions; retrieval starts at the disease and follows incoming treatment edges before visiting ingredients and their interactions.

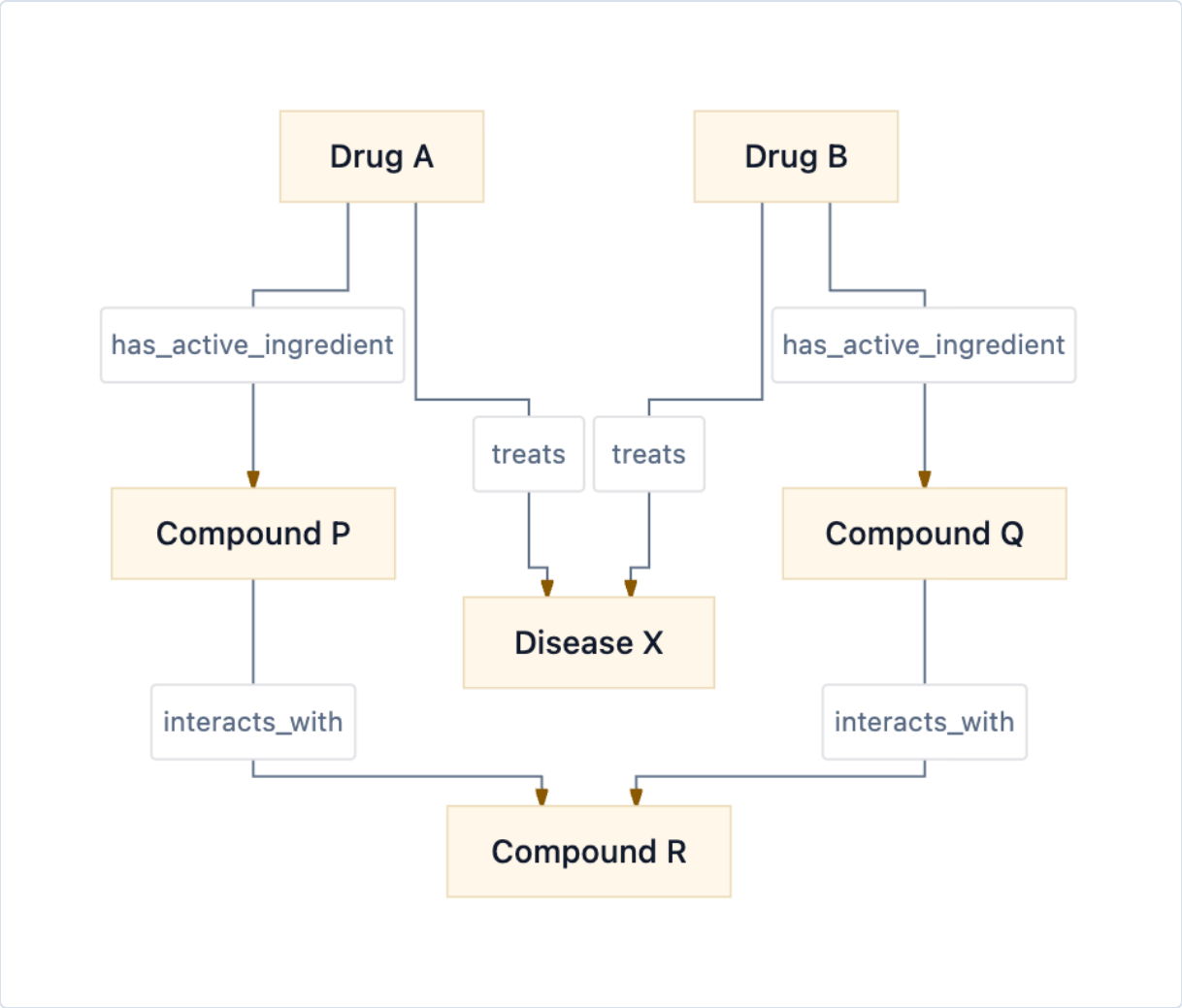


Figure 5.8: A multi-hop query finds Drug A and Drug B through their treatment links to Disease X, then visits Compounds P and Q and their interaction links to Compound R.

Each hop follows a stored edge that can be audited.

Traced step by step for a small graph: the query names Disease X; the *treats* edges into Disease X yield Drug A and Drug B; the *has_active_ingredient* edges from each yield Compounds P and Q; the *interacts_with* edges from P and Q yield Compound R. The answer is R, and the trace records the edges taken to produce it, each of which a reviewer can check against the source that asserted it. To add this to your own graph, three things are needed: typed,

directed edges (Chapter 8); a traversal depth limit per query type, because unlimited hops produce unlimited answers; and a trace that records the edges taken, which is what the grounding gate of Chapter 9 consumes.

Inference derives relationships from stored facts. The validators in Chapter 14 can check those derived relationships for contradictions that a document lookup alone would miss.

From Knowledge Graphs to Context Graphs

A **context graph** is a triples-based representation of data designed for AI consumption (TrustGraph documentation, trustgraph.ai, 2025).

The Key Insight: Structure Carries Meaning

TrustGraph reports, from its own experiments, that **passing graph results in structured formats such as Cypher and RDF improves model responses despite higher token overhead** (TrustGraph documentation, trustgraph.ai, 2025); the finding is vendor-reported and has not been reproduced for this book. When you pass a Cypher query result to an LLM, the syntax itself encodes what is a node, what is a property, and what is a relationship. The structure carries semantic information beyond the literal content.

The vendor’s finding gives a reason to compare structured results with flattened text on the intended workload, including the extra token cost.

RDF, Property Graphs, and the Non-Argument

A practical question in context graph design: should the backend be a triplestore (RDF), a property graph (Neo4j/Cypher), or something else? The answer is that **it does not matter much for the pipeline architecture**. The same information can be stored as RDF triples, property graph nodes and edges, or even joined relational tables.

TrustGraph reports that its retrieval agents behave the same on its Cassandra and Neo4j backends (TrustGraph documentation, trustgraph.ai, 2025; the vendor's own account). What matters is:

- **Ontology quality**: Well-defined classes, properties, and relationships
- **Synonym coverage**: How many ways users can refer to the same concept
- **Relationship completeness**: Whether the important connections are captured
- **Update mechanism**: How quickly the graph reflects new knowledge

The storage backend is an implementation detail. The context engineering pipeline operates at the abstraction layer above it.

Mature Graph Algorithms: The Untapped Toolbox

Decades of graph algorithm research remain only partly used in AI retrieval. Microsoft GraphRAG uses Leiden community detection at the centre of its index, so clustering is not untapped; the author's

reading of the open-source implementations is that the rest of the toolbox is:

- **Traversal optimisation**: Determining optimal depth per query type, rather than fixed depth
- **Clustering**: Grouping related documents by graph structure, not just content similarity
- **Density calculations**: Identifying densely connected subgraphs as high-confidence knowledge domains
- **Outlier detection**: Flagging entities with unusual connectivity as potential errors or discoveries
- **Community detection beyond indexing**: using the communities at query time to scope a search, not only to write summaries

These established algorithms could improve retrieval without another model call. Chapter 22 describes their use in hybrid pipelines.

Summary: Knowledge Graphs as Context Infrastructure

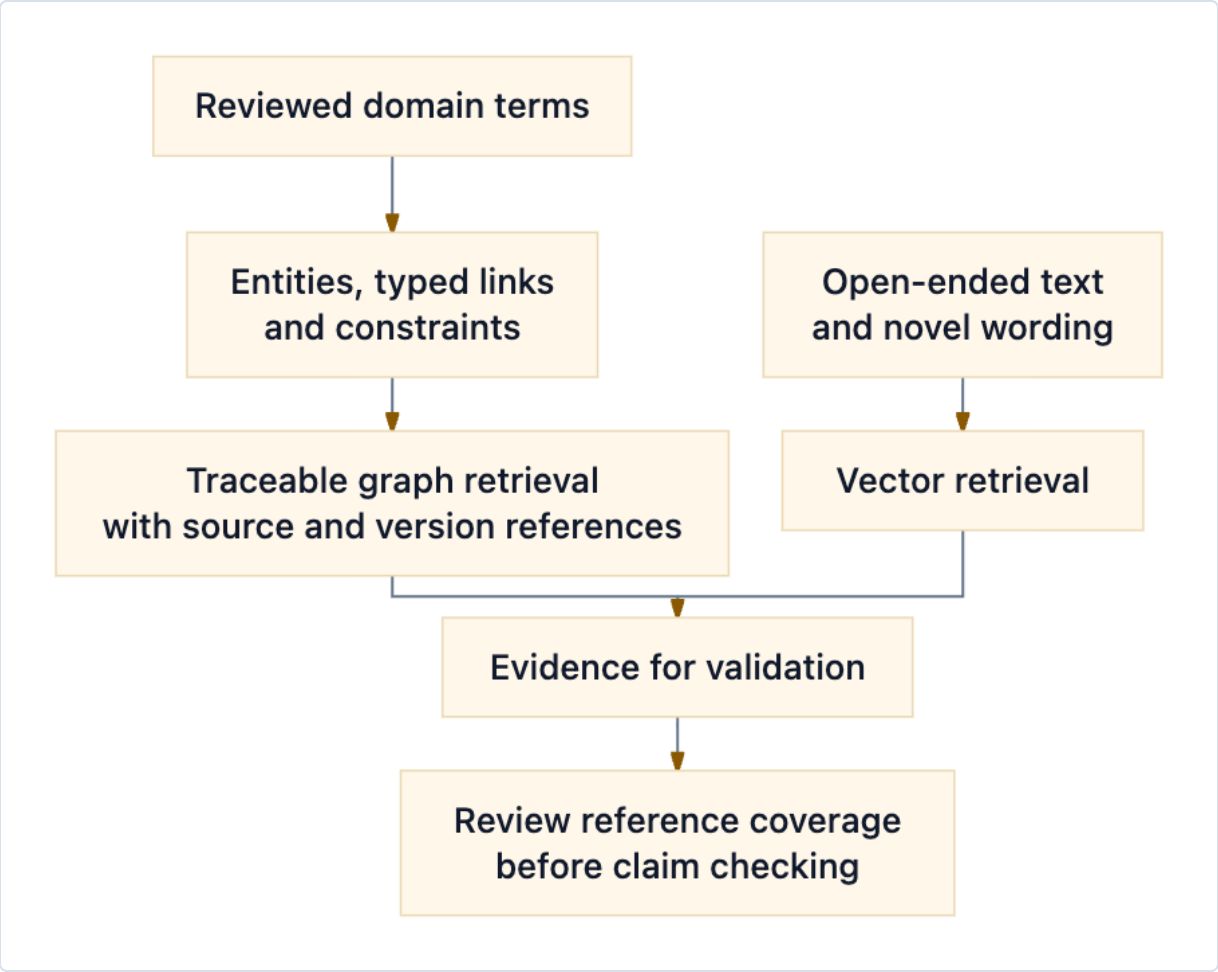


Figure 5.9: Chapter 5 recap: Typed graph knowledge supports traceable retrieval and validation, alongside vector search for uncovered text.

Use a knowledge graph for context that needs explicit entities, relationships and constraints. Its paths make retrieval traceable, and its facts provide a reference for validation. SNOMED CT illustrates curated terminology at scale; GraphRAG illustrates model-assisted extraction, whose outputs still need checking before safety-critical use. Vector search remains useful for open-ended queries and text outside the graph's vocabulary.

The result handed to retrieval is a reviewable representation with typed relationships and sources. In the maintenance case, the

assistant must still find the right rule when a user describes the failed job in different words. Chapter 6 compares retrieval methods for that task; Chapters 8 and 9 address curation and claim checking. A traceable graph path establishes how a result was selected, not whether the underlying claim is current or true.

Key Sources

- Microsoft Research, “From Local to Global: A Graph RAG Approach to Query-Focused Summarization” (Edge et al., arXiv 2404.16130, 2024) and the GraphRAG documentation
- SNOMED CT official documentation and release files
- Terraphim AI documentation and MedGemma competition results
- PrimeKG (Precision Medicine Knowledge Graph; Chandak, Huang and Zitnik, *Scientific Data*, 2023) for biomedical relationships
- TrustGraph documentation and engineering blog, trustgraph.ai (2025): context graphs, structured formats, backend equivalence and the RAG-to-autonomous-learning progression, as reported by the vendor

Chapter 6: Deterministic Search vs Vector Embeddings

Retrieval must connect a user's wording to the relevant context. This chapter compares vocabulary matching with embedding similarity and explains when to use each.

The two dominant approaches are **deterministic search** (pattern matching against known vocabulary) and **vector embeddings** (similarity search in high-dimensional space). This chapter compares them on the dimensions that decide production fit, and ends with a hybrid pipeline that uses both.

How Vector Embeddings Work

Many retrieval-augmented generation (RAG) systems fetch documents through vector embeddings before placing them in the prompt. Embeddings represent text as **vectors**, arrays of numbers in a high-dimensional space, where similar texts should occupy nearby positions.

The Process

1. **Encoding**: A model (BERT, OpenAI's text-embedding-3, etc.) converts text into a vector, typically 768 to 3072 dimensions (BERT base produces 768 dimensions; OpenAI's text-embedding-3-large produces up to 3072)

2. **Indexing**: The vectors are stored in a vector database (Pinecone, Weaviate, Chroma, Milvus)
3. **Querying**: When you ask a question, your query is also encoded into a vector
4. **Similarity Search**: The database finds the stored vectors that are most similar to your query vector

Figure 6.1 shows the encoding and comparison steps for the example query.

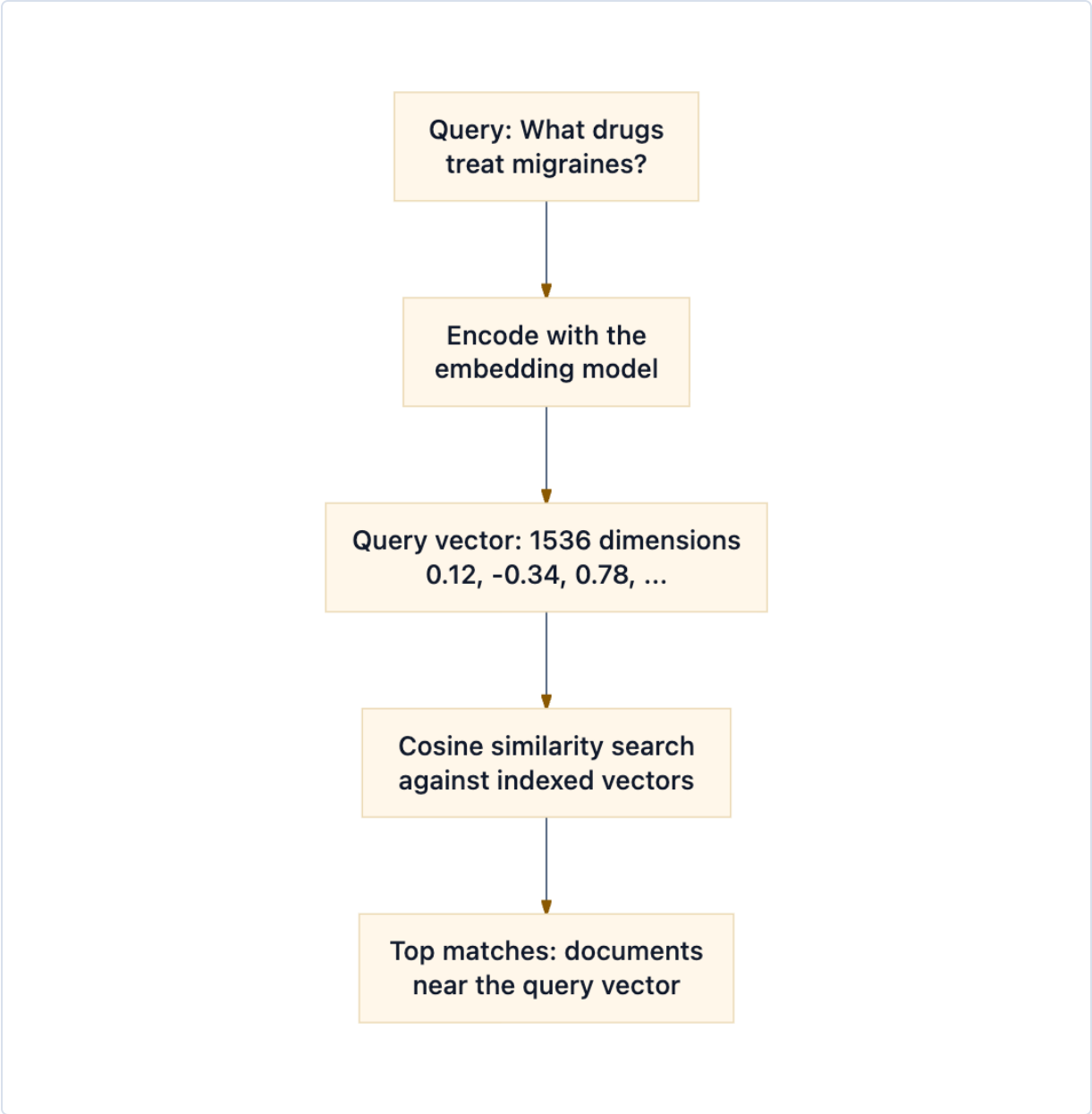


Figure 6.1: An illustrative 1536-dimensional query vector is compared with indexed vectors; proximity determines the retrieved documents.

The Embedding Problem

An embedding is a lossy semantic representation: proximity between vectors does not explicitly encode a typed relationship such as Drug A treats Disease B. The embedding model learns statistical

patterns from training data, including the contexts in which texts about the drug and disease appear.

This creates several problems:

Information loss in compression: A knowledge graph with 1,000 nodes and complex relationships cannot be fully represented even in a 768-dimensional or 1536-dimensional vector. The relationship structure is lost. Only statistical co-occurrence patterns remain.

No explainability: When a vector search returns results, you get a similarity score. But you cannot explain *why* a document matched. "The cosine similarity was 0.87" is not a satisfying answer when someone asks "why did you recommend this?"

Approximate, and unstable across changes: at a fixed model, index and seed, an embedding query is deterministic. What makes it unreliable in practice is that none of those stays fixed: the index changes with every ingest, the vendor repoints the model alias, and approximate nearest-neighbour search returns a different neighbour set after a rebuild. The result is the same query returning different documents on different days, with no change the caller can see, which is what matters for testing and audit.

The Cleora problem: Cleora (Rychalska et al., 2021) and similar graph embedding systems attempt to preserve graph structure in vectors. But they face an inherent trade-off: you cannot perfectly preserve a complex graph (thousands of nodes, millions of

relationships) in a low-dimensional vector without massive information loss. The compression is lossy, not lossless.

The Cost of Probabilistic Search

Vector search costs include:

- **Latency**: 1-3 seconds per query (encode + vector search + retrieval), the author's observation of hosted pipelines in 2025 and 2026 (see Key Sources)
- **Memory**: hundreds of megabytes to tens of gigabytes for the vector index, depending on corpus size and dimensionality (indicative, from the same observations)

Assumptions behind those ranges: a hosted embedding API with a 1,536-dimension model, a managed vector database over a corpus of a few hundred thousand chunks, float32 vectors with an approximate index, generation excluded, and no percentiles recorded. A local lookup on a small index is far cheaper and faster than this; the comparison that matters is with the automaton over the same corpus, which is the row labelled as the author's measurement in the table below. - **Compute**: Requires GPU or high-CPU instances for large indexes - **API costs**: Embedding models charge per 1K tokens - **Cold start**: Seconds to load the model and index

For a system that answers every query with an embedding call, these costs scale with query volume. A system processing a million queries a day spends $1,000 a day at $0.001 per embedding call, an

arithmetic illustration at a 2026 list price for a hosted embedding API (Key Sources), before retrieval or generation; hosted retrieval pipelines in the $0.01 to $0.10 per query range end to end, the author's observed range for the same period, are ten to a hundred times that.

The Deterministic Alternative

Deterministic search takes a different approach: pattern matching against a known vocabulary using automata.

Instead of converting text to vectors, the system:

1. **Builds an automaton**: Compiles a vocabulary of terms into an Aho-Corasick automaton, a trie of all the patterns with failure links that let a single pass over the text find every pattern occurrence without backtracking; the author's clinical build holds about 1.4 million SNOMED CT surface forms
2. **Scans input**: When you provide text or a query, the automaton scans for matches
3. **Ranks results**: Matches are ranked by connectivity in the knowledge graph

Figure 6.2 follows the same question through vocabulary matching and graph traversal.

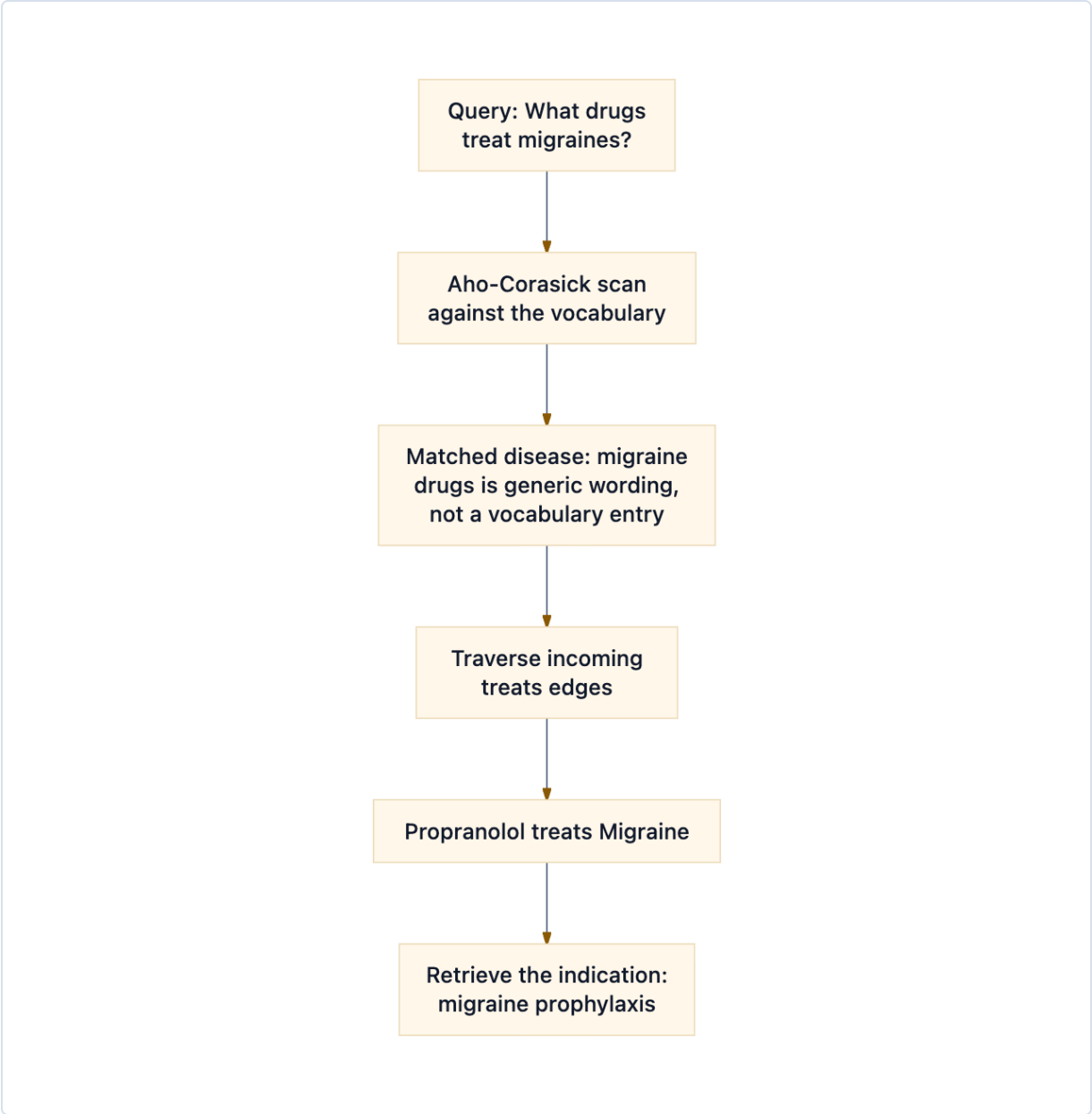


Figure 6.2: Vocabulary matching identifies migraine, then a typed treatment edge retrieves the stored prophylaxis indication; the generic word drugs is not a vocabulary match.

Key Characteristics

Exact recall for known terms: If “migraine” is in the vocabulary, the automaton finds every occurrence of it and of its listed synonyms,

and nothing else. There is no embedding drift because there is no embedding.

Explainable: The trace records the matched term, its position and the relationship used to retrieve the document.

No per-query cost: The automaton is compiled once per vocabulary and runs locally, without API calls or metered tokens.

Sub-millisecond matching: In the author's Terraphim measurements (Chapter 7), the automaton scans a clinical note against 1.4 million SNOMED CT surface forms in under a millisecond, and a full query including graph scoring returns in under 200 milliseconds. At that speed search can run on every keystroke.

Information preserved: The full graph structure is preserved. No lossy compression into vectors.

Head-to-Head Comparison

Figure 6.3 condenses the comparison table that follows.

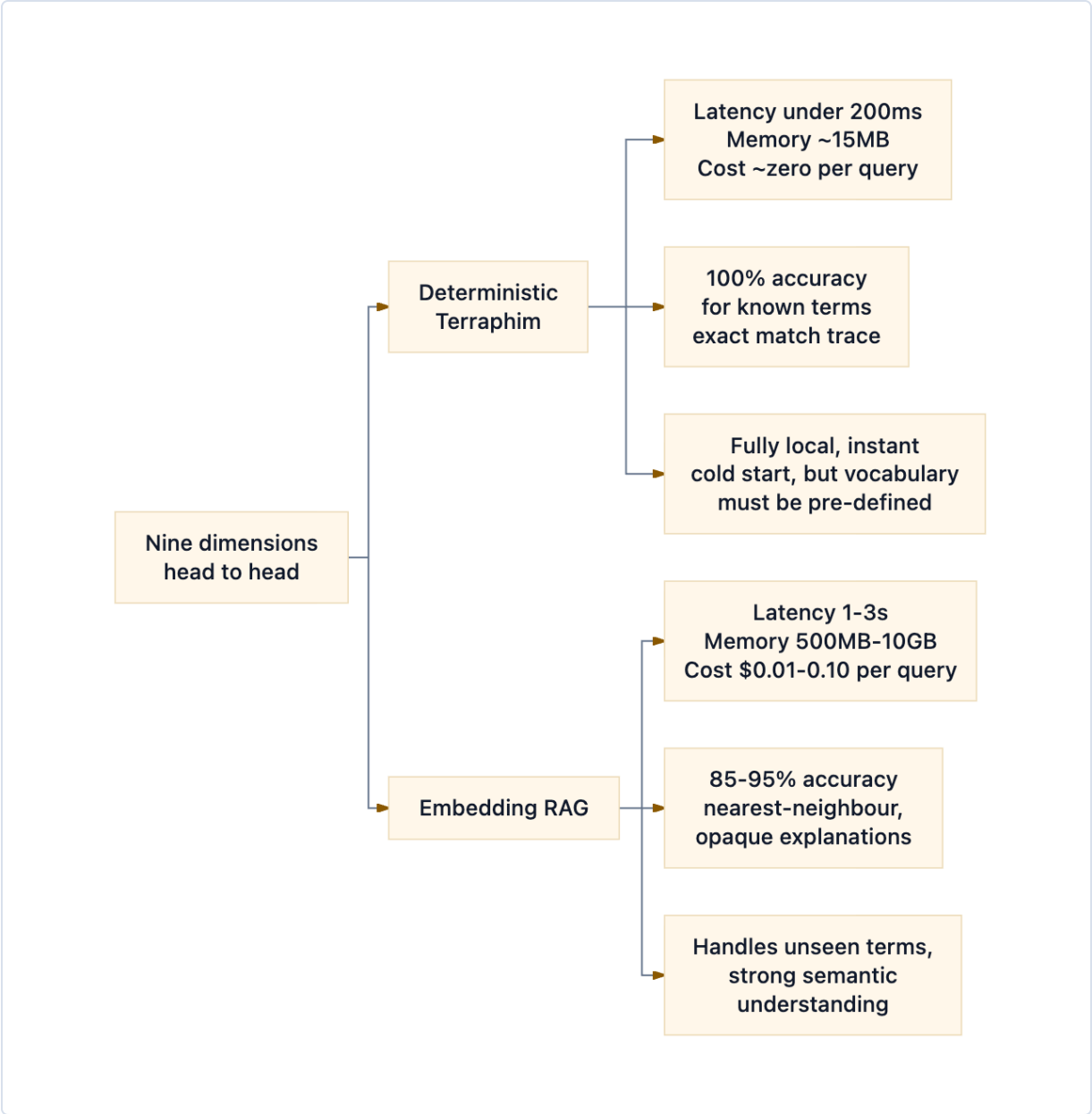


Figure 6.3: Nine-dimension comparison of deterministic (Terraphim) and embedding RAG retrieval, condensed.

Dimension	Deterministic (Terraphim)	Embedding RAG
Latency	<1ms match, <200ms query (author's measurements, Chapter 7)	Indicative 1-3s for hosted pipelines

Dimension	Deterministic (Terraphim)	Embedding RAG
Memory	~15MB (author's clinical build, Chapter 7)	500MB-10GB (author's observations, 2025-2026)
Recall	Exact over active thesaurus entries, after normalisation and word-boundary rules; nothing outside them	High but approximate; retrieval benchmarks such as MTEB (Muennighoff et al., 2023) report wide variation by model and task
Explainability	Exact match trace	"Nearest neighbour" (opaque)
Privacy	Fully local, no data leaves device	Often cloud-based
Cost	No per-query cost	Indicative $0.01-0.10 per query for hosted pipelines, author's observations and vendor list prices, 2026
Cold start	Instantaneous	Seconds (load model)
Vocabulary	Must be pre-defined	Handles unseen terms
Semantic understanding	None (literal matching)	Strong

The Terraphim column reports the author's measurements described in Chapter 7. The embedding column gives indicative ranges for hosted pipelines in 2026; they vary by vendor, model and corpus, and the reader should substitute their own figures before deciding anything on cost.

Where Deterministic Wins

Healthcare and Safety-Critical Applications

The MedGemma evaluation (Chapter 9) demonstrates the stakes. In the author's internal evaluation of 18 clinical cases (the 543 automated tests cover the pipeline's components; the 18 cases test end-to-end recommendations, as Chapter 9 defines), the raw model recommended osimertinib at 800mg, a tenfold overdose. With the grounding gate in place the output was 80mg, the dose used in the FLAURA trial (Soria et al., NEJM 2018): the gate ran in correct mode, replacing the model's dose with the licensed dose held in the graph, and logged the substitution. "Grounded" there means every entity in the final recommendation matched the graph and the recommendation agreed with the reference.

The example demonstrates a gate behaviour on an internal evaluation: the labelled source was the licensed dose in the graph, the pass criterion was agreement with the reference recommendation, and correct mode means the gate substituted the graph's value and logged it (Chapter 9 gives the protocol). It is not a clinical safety validation, the pipeline has no regulatory status, and the evaluation is not a public benchmark. The tenfold dosage error illustrates why that recommendation needed a check against the licensed dose.

Compliance and Audit

When regulators ask “why did the system recommend X?”, deterministic systems can answer:

> “Term ‘contraindicated’ matched at position 247 in document ‘drug-interactions.md’, linked to drug ‘Warfarin’ via relationship ‘contraindicated_for’, which triggers the warning you received.”

A plain similarity score offers no trace:

> “The cosine similarity between your query and this document chunk was 0.87.”

The first answer identifies a document and relationship that a reviewer can inspect. The similarity score alone supplies neither.

High-Throughput, Low-Latency

In the author’s measurements, Terraphim scans a document against 1.4M SNOMED CT (Systematized Nomenclature of Medicine, Clinical Terms) patterns in under a millisecond. A time-tracking agent using the same automaton categorises activities in similar time. This makes matching feasible on each keystroke, webhook or event in those systems.

These measurements make deterministic search worth evaluating for interactive interfaces, event pipelines and edge devices with single-digit-millisecond latency requirements.

Edge and Offline

An automaton of about 15MB (the author's SNOMED-derived build) fits where a vector index does not:

- Mobile phones (iOS, Android)
- IoT devices
- Browser-based (compiled to WebAssembly, WASM)
- Air-gapped systems

Local matching requires neither a network connection nor a metered API call, which suits privacy-sensitive and offline applications.

Where Embedding RAG Wins

Deterministic search is not a replacement for all AI search. Embedding-based RAG is the better fit for:

- **Open-ended questions**: "What are the implications of this contract clause?" requires semantic understanding beyond pattern matching
- **Unseen content**: Documents not in the vocabulary need embedding-based retrieval
- **Conversational search**: Multi-turn queries with context benefit from LLM comprehension
- **Creative tasks**: Brainstorming, summarisation, paraphrasing where semantic flexibility matters

The Hybrid Pipeline

Figure 6.4 shows the routing decision at the heart of the hybrid pipeline.

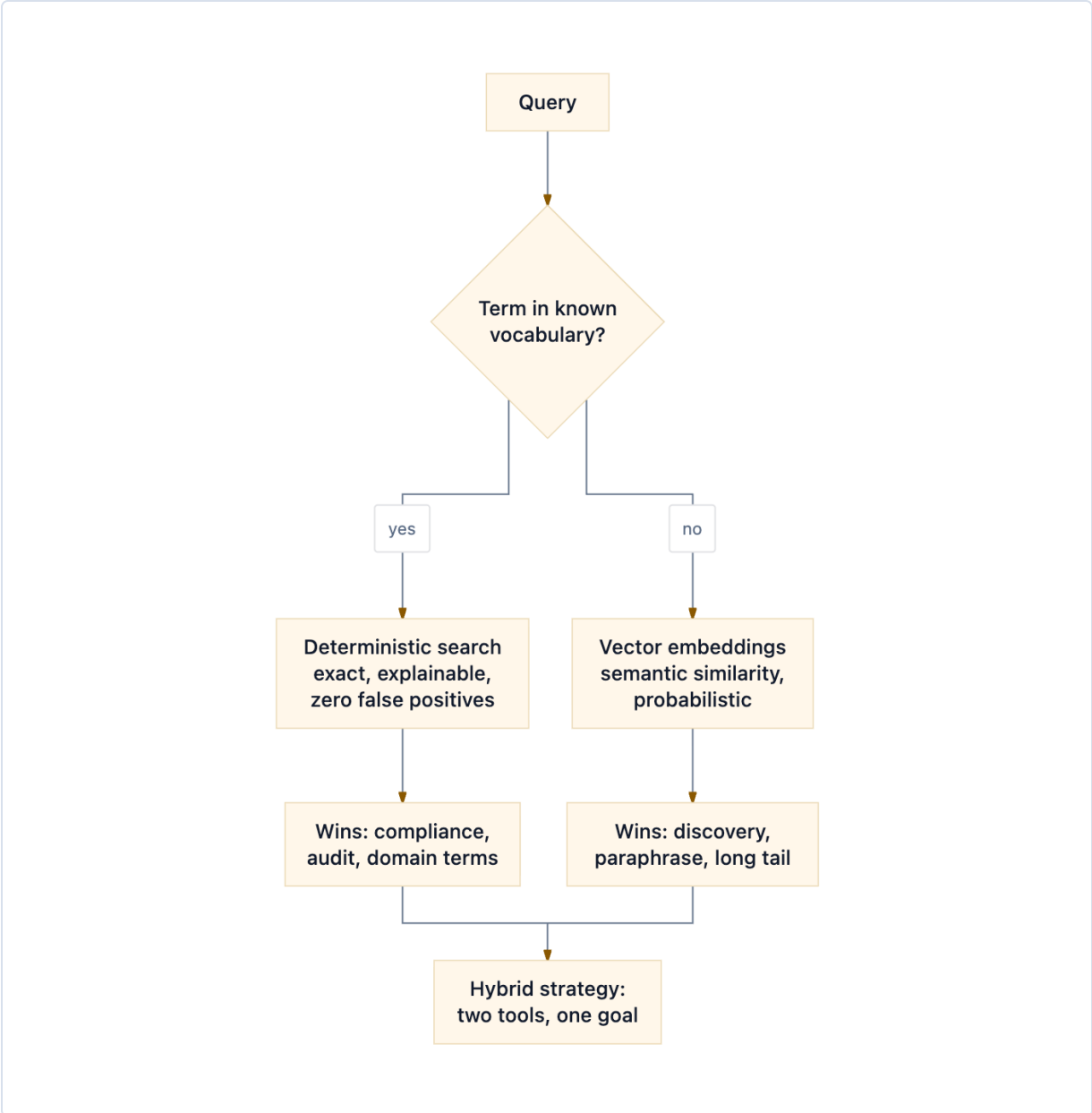


Figure 6.4: Route by vocabulary membership: known terms go to deterministic search, unknown terms to embeddings, and both feed one hybrid pipeline.

The architecture the author recommends combines both approaches:

1. **Deterministic layer** (first): Fast, exact, explainable. Handles known vocabulary, compliance terms, domain entities.
2. **LLM layer** (fallback): Routes to appropriate model for open-ended queries that don't match known patterns.
3. **Knowledge graph grounding gate**: LLM output is validated against the deterministic knowledge graph before reaching the user.

This gives you: - Deterministic speed for the queries that use known vocabulary, which in the author's deployments is the majority - Semantic capability of the LLM for the rest - A grounding gate that stops unverified claims before they reach production (Chapter 9)

Chapter 22 develops the hybrid pipeline in detail.

Getting Started

Three steps establish a deterministic lane:

1. Export your vocabulary as `(surface_form, canonical_id)` pairs from the thesaurus of Chapter 8, synonyms included.
2. Build the automaton once and load it per process:

```
import ahocorasick
A = ahocorasick.Automaton()
for surface, entity in vocabulary:
    A.add_word(surface.lower(), (entity, len(surface)))   # value
carries the length for coverage
A.make_automaton()
```

```
def route(query, threshold=0.5):
    q = query.lower(); covered = [False] * len(q); hits = set()
    for end, (entity, length) in A.iter(q):          # store (entity,
len(surface)) as the value
        hits.add(entity)
        for i in range(end - length + 1, end + 1): covered[i] = True
    alpha = [c.isalnum() for c in q]
    coverage = sum(1 for i, a in enumerate(alpha) if a and covered[i])
/ max(1, sum(alpha))
    return ("deterministic", hits) if coverage >= threshold else
("semantic", hits)

# vocabulary = [("migraine", "C0149931"), ("propranolol", "C0033497")]
# route("which drugs treat migraine")  -> ("semantic", {"C0149931"})
coverage 8/23 = 0.35
# route("migraine propranolol")        -> ("deterministic",
{"C0149931", "C0033497"})  coverage 16/16
```

3. Put the grounding gate of Chapter 9 after whichever lane answered, so a confident wrong answer from either lane is caught the same way.

Coverage here is the share of the query's letters inside matched surface forms, so a four-word question with one matched disease routes semantic, which is what you want. The threshold is a starting value for a small vocabulary; calibrate it on a labelled set of a few hundred real queries, recording three rates: false-deterministic (routed deterministic, answer wrong), semantic-rescue (routed semantic when one synonym edge would have made it deterministic) and downstream gate rejection. Feed the rescued terms back into the thesaurus, and revisit the threshold whenever the thesaurus grows by more than a tenth.

Summary: Two Tools, One Goal

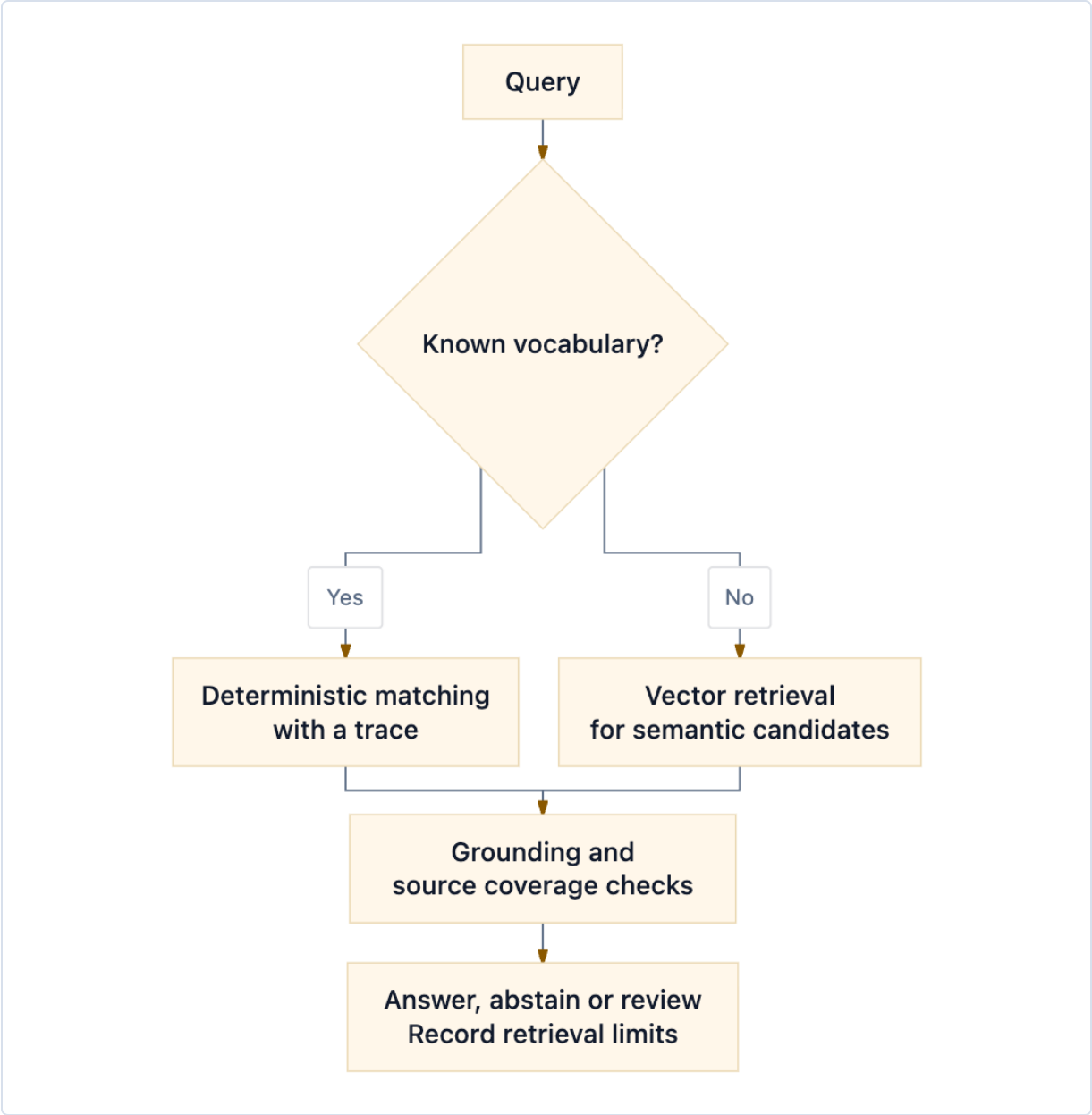


Figure 6.5: Chapter 6 recap: Route known terms and novel wording through suitable retrieval methods, then check the evidence.

Both deterministic search and vector embeddings connect queries to knowledge; the head-to-head table above is the comparison, and it reduces to one rule. Where the vocabulary is known and the cost of a

wrong match is high (safety-critical, compliance-sensitive, high-throughput work), the deterministic lane answers and the trace shows why. Where the query is open-ended or uses words the vocabulary lacks, embeddings answer and the grounding gate checks them.

The systems the author has built or reviewed benefit from **both**: deterministic for the known, embeddings for the unknown, with a grounding gate, run by the pipeline before output reaches the user, to catch hallucinations.

The handoff is a retrieval choice with known vocabulary coverage, failure cases and source references. Chapter 7 makes one implementation inspectable; Chapter 22 later combines retrieval lanes and measures selection errors. For the repair request, finding a relevant rule is still insufficient grounds to execute it.

Key Sources

- Terraphim measurements reported in Chapter 7 (author's development hardware; orders of magnitude, not benchmarks)
- Embedding-pipeline latency, memory and cost ranges: the author's observations of hosted retrieval pipelines in 2025 and 2026, and the public pricing pages of hosted embedding APIs checked October 2026; indicative ranges, not benchmarks or quotes
- Barbara Rychalska et al., "Cleora: A Simple, Strong and Scalable Graph Embedding Scheme" (2021)

- MedGemma competition results (543 tests, 18/18 grounded)
- Zestic AI framework on deterministic vs embedding approaches
- Knowledge-garden trend analysis: “Deterministic vs Embedding”
- Soria et al., “Osimertinib in Untreated EGFR-Mutated Advanced Non-Small-Cell Lung Cancer” (FLAURA), *New England Journal of Medicine*, 2018
- Niklas Muennighoff et al., “MTEB: Massive Text Embedding Benchmark” (2023), for the variation in embedding retrieval quality
- Alfred Aho and Margaret Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, *Communications of the ACM* (1975)

Chapter 7: Terraphim Graph Embeddings: Deterministic Search with Full Explainability

Terraphim calls its retrieval representation **graph embeddings**, using the term differently from the graph-learning literature. Conventional graph embeddings represent a graph as vectors. Terraphim associates documents with graph nodes through vocabulary matches and ranks them using graph structure. This chapter uses the term in Terraphim's sense and describes its matching and scoring implementation.

Terraphim keeps the graph's edges, types and properties available to the scoring layer.

Terraphim delivers exact precision and recall *against the thesaurus entries of the active role*, on exact and synonym surface forms, which we call **deterministic precision** throughout this book. That claim is bounded: it means no false positives and no missed matches for the terms the thesaurus defines, and it says nothing about terms it does not.

Timings in this chapter (under one millisecond to match, about twenty milliseconds to rebuild an automaton, under two milliseconds for file selection) are the author's measurements on development hardware against a SNOMED CT-derived vocabulary of about 1.4

million surface forms. They were taken single-threaded on the author's development laptop in 2025, without recorded run counts or percentiles, which is why they are reported as orders of magnitude and not as a benchmark; the ratio against embedding retrieval is the author's observation on that one machine and has not been measured elsewhere. A separate limitation is **coverage for unknown terms**. When a query uses terminology outside the vocabulary, Terraphim cannot match it. Chapter 22 addresses this limitation through hybrid pipelines that combine deterministic search with semantic methods for novel queries.

Explicit Relationships vs Inferred Similarity

Vector embeddings capture statistical patterns in training data. Texts about "aspirin" and "headache" may have similar vectors, but the similarity does not state the relationship between the two.

Terraphim takes a different approach: **explicit relationships defined in the knowledge graph**.

Figure 7.1 makes the three example relationship types explicit.

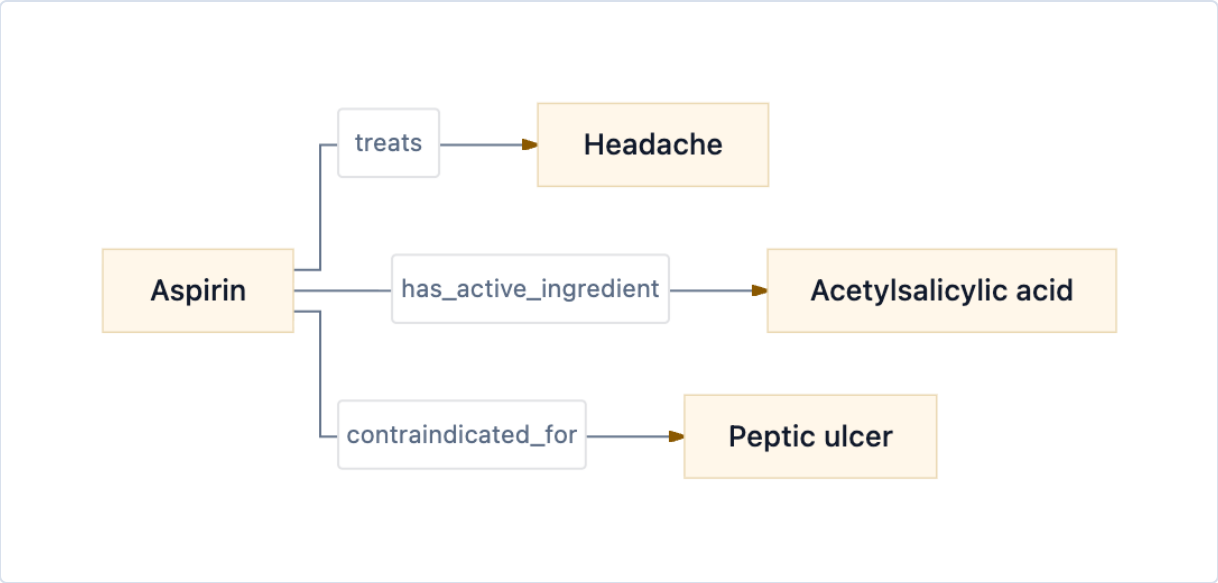


Figure 7.1: The illustrative graph distinguishes an aspirin treatment relationship, an active ingredient and a contraindication through three different edge types.

The system looks up these typed relationships directly in the graph.

Why Explicit Beats Implicit

Aspect	Explicit (Terraphim)	Inferred (Embeddings)
Relationship type	Typed (“treats”, “contraindicated_for”)	Generic similarity
Can identify	Exactly why A relates to B	“They’re similar”
Confidence	Certain for known terms	Probabilistic
New relationships	Must be added to graph	Inferred automatically

Safety-critical queries need the relationship type and its source, so that a reviewer can check what the connection means.

Deterministic Ranking: Same Query → Same Results

Figure 7.2 shows the pipeline whose stages the rest of this chapter examines in turn.

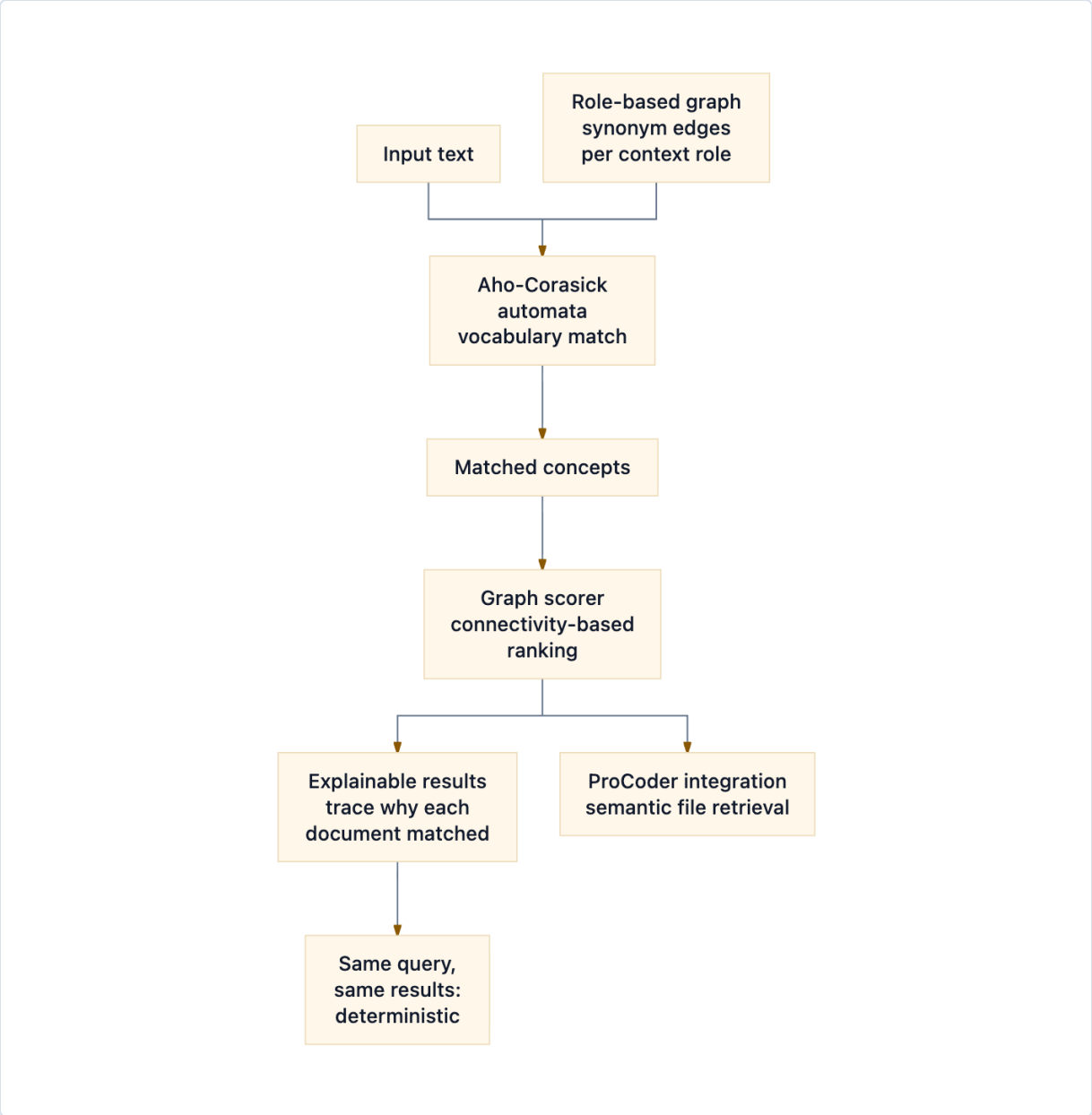


Figure 7.2: The Terraphim pipeline: role-based synonym graph, Aho-Corasick matching, connectivity scoring, and explainable, deterministic results.

Terraphim's retrieval is **deterministic** for a fixed thesaurus, graph and configuration: repeated queries return the same results.

For a fixed model, index, hardware and settings, embedding generation and exact nearest-neighbour search are deterministic too. What makes embedding retrieval pipelines non-reproducible in practice is operational: run the same query on different days and you may get different results because:

- The index changed between the two runs
- The embedding model was replaced or retrained
- Approximate nearest-neighbour search returned a different neighbour set

These operational changes can be invisible to the caller unless the system records the model and index revisions.

For production AI systems, that operational variability has three costs:

- **Testing is undermined**: a test that asserts on a result set fails when the neighbour set shifts, so teams stop asserting on results
- **Auditing is weakened**: the trace of why a result was returned is a similarity score against an index that no longer exists
- **Debugging is slow**: reproducing the user's query against the index they hit may not be possible

Terraphim solves this by using deterministic algorithms:

1. **Aho-Corasick automaton** for term matching: for a fixed thesaurus, role and document, the matcher returns identical matches
2. **Graph traversal** for relationship lookup: for a fixed graph, the same paths
3. **Connectivity-based scoring**: for the same graph state, the same scores

Run the query 100 times over the same graph, get the same results 100 times; that is the property a test suite and an audit trail can rely on.

Explainable Results: Trace Exactly Why a Document Matched

When a Terraphim query returns results, you can trace exactly why:

1. **Term match**: Which term in the query matched?
2. **Position**: At what character position in the document?
3. **Synonym expansion**: Which synonym mapped to the canonical term?
4. **Relationship path**: What path through the graph connects the matched term to the document?
5. **Score**: How was the final score calculated?

These fields provide a retrieval trace. When regulators ask “why did the system return this?”, the trace can show:

> "Term 'Warfarin' in the query matched at position 1,420 in document 'drug-interactions.md'. This document was linked to the 'contraindicated_for' relationship in the knowledge graph, connecting Warfarin to 'Bleeding risk', which your medication list includes as a condition. Score: 0.95."

Compare this to vector embeddings:

> "The cosine similarity between your query vector and this document chunk was 0.87."

The first answer identifies evidence a reviewer can inspect; the second supplies only a similarity score. Whether either satisfies an auditor depends on what surrounds it: the graph version, the validation record, the policy the decision maps to and the retention of the index state (Chapter 21).

No Loss of Represented Structure: The Graph Is Kept Whole

Vector embeddings compress information into fixed-length vectors. Embedding a graph of 10,000 entities and 100,000 typed relationships into 1,536-dimensional vectors preserves proximity but not the edge types, directions or properties: two nodes end up near each other, and the vector does not say whether one treats the other or is contraindicated for it. The graph-embedding literature treats

this as a known fidelity trade-off (Cleora, Chapter 6) rather than a defect, and it is the trade-off this chapter declines to make.

The scope of the claim is the graph. Meaning in a document that neither the thesaurus nor the graph represents is invisible to the deterministic lane; that is the coverage limit stated at the start of the chapter, and it is why gap detection exists. Within the graph, Terraphim preserves the **full structure**:

- Each entity the thesaurus defines is stored explicitly
- Each relationship keeps its type
- Each relationship keeps its direction and properties
- The graph can be queried at any depth

This means:

- **No hidden assumptions**: You see everything the system knows
- **Complete auditing**: The entire knowledge state is inspectable
- **Consistent behaviour**: Changes to the graph are predictable

Role-Based Graphs: Different Synonym Edges Per Context Role

Terraphim's **role-based graphs** give the same entities different relationships depending on the active role.

The Problem with Single Vocabularies

In a traditional knowledge base, “Client” means one thing. But in a complex organisation:

- In **sales** context, “client” means “prospect or customer”
- In **support** context, “client” means “someone with an active subscription”
- In **legal** context, “client” means “party to a contract”

Using the same vocabulary for all contexts leads to confusion and incorrect results.

Terraphim’s Solution: Role-Based Graphs

Terraphim creates **different synonym edges per context role**:

```
{
  "entity": "client",
  "roles": {
    "sales": {
      "synonyms": ["prospect", "lead", "customer", "buyer"],
      "relevant_relationships": ["opportunity", "pipeline", "deal"]
    },
    "support": {
      "synonyms": ["subscriber", "account holder", "licensee"],
      "relevant_relationships": ["ticket", "subscription",
"entitlement"]
    },
    "legal": {
      "synonyms": ["party", "counterparty", "contractor"],
      "relevant_relationships": ["contract", "obligation",
"liability"]
    }
  }
}
```

When you query the knowledge graph, you specify the role. The system uses only the synonyms and relationships relevant to that role.

The same term means different things in different contexts, and a system that serves several of them must carry that distinction.

The Formal Role Model

The role-based graph concept has a formal foundation in Medvedeva's course text *Ontologics and Communication* (School of Systems Management, 2021), which builds on the stakeholder and viewpoint concepts of ISO/IEC/IEEE 42010 and adds two components of its own. Each role is defined by three components: **concern** (what objects attract the role's attention), **preference** (what the role wants to happen), and **intention** (strategies the role plans to use).

In the sales context, the concern is revenue opportunity, the preference is deal closure, and the intention is pipeline progression. In the support context, the concern is customer satisfaction, the preference is issue resolution, and the intention is SLA compliance. These structural differences explain *why* the same entity ("client") needs different synonym edges per role – because the role's concern determines which related concepts are relevant.

ISO/IEC/IEEE 42010 formalises the *viewpoint* as the conventions that frame a set of stakeholder concerns, and Medvedeva paraphrases it as descriptions composed from a role's position with the role's objects of interest highlighted. When Terraphim rebuilds the Aho-

Corasick automaton for a new role (in ~20ms), it changes which descriptions are active, which concerns are foregrounded, and which synonym edges are in play. This implements the vocabulary aspect of a viewpoint switch.

The Aho-Corasick Automata Approach

Terraphim uses the **Aho-Corasick algorithm** to find occurrences of many patterns in one scan through the text.

Why Aho-Corasick?

Traditional approaches scan through each pattern separately:

Figure 7.3 shows the repeated work in separate pattern scans.

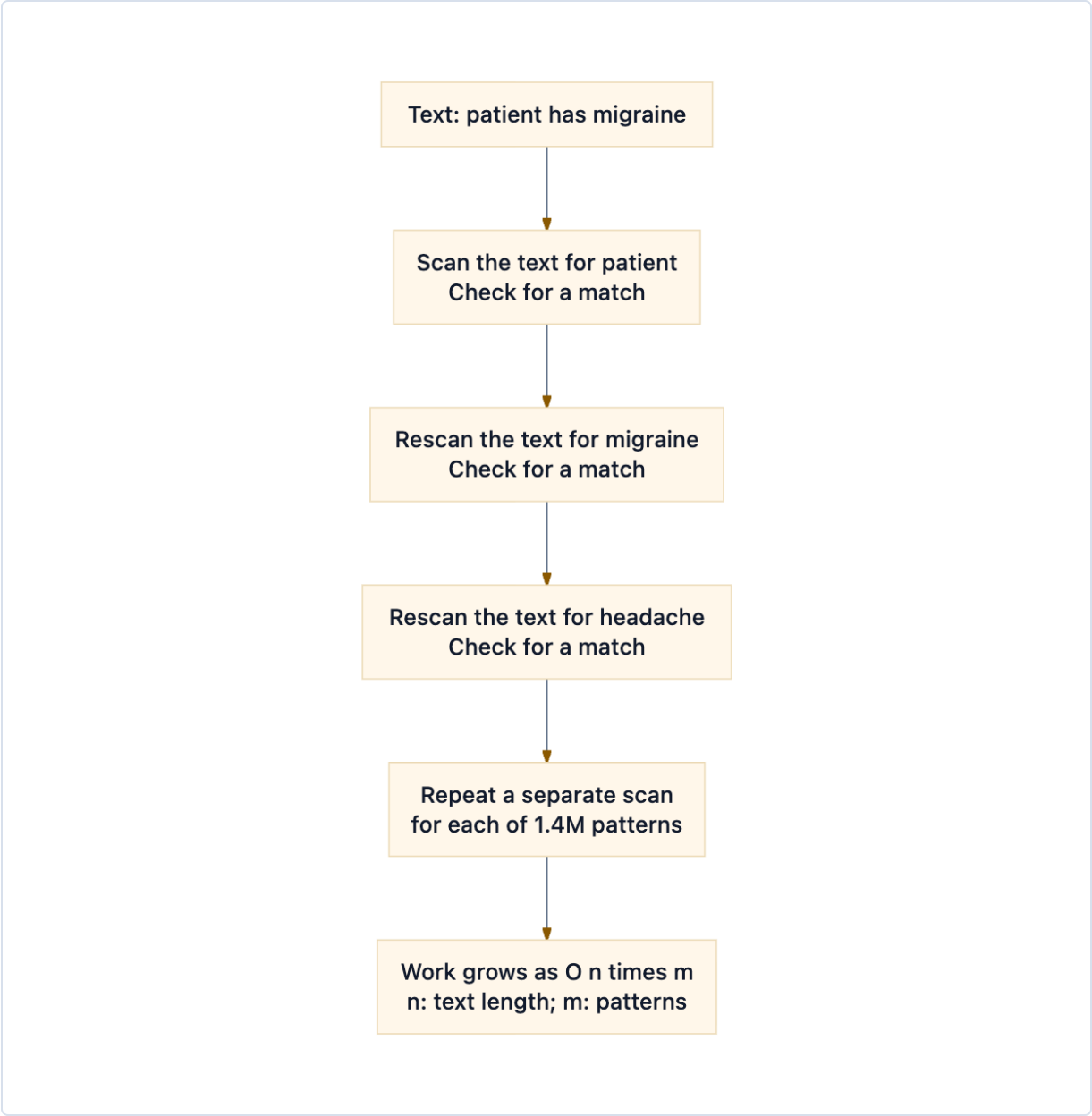


Figure 7.3: Separate pattern matching scans the same text once per pattern; the three shown scans stand for repetition over the full pattern set.

This requires O(n × m) work, where n is text length and m is pattern count. Scanning separately for 1.4M patterns repeats substantial work.

Aho-Corasick builds a **finite-state automaton** from all patterns at once:

1. **Build the automaton**: All 1.4M patterns are compiled into a single state machine
2. **Single scan**: The text is scanned once, transitioning through the automaton
3. **All matches found**: Every pattern that appears in the text is identified in O(n)

This gives:

- **Under 1ms** to scan a clinical note against 1.4M patterns (author's measurement on development hardware, SNOMED CT vocabulary, single thread)
- **Matching cost O(n + z)** after construction, for text length n and z reported matches, independent of the pattern count; construction is O(total pattern length) and is paid once per thesaurus release, and the automaton occupies about 15MB serialised for the clinical build
- **No per-query construction cost**: the automaton is compiled once per role; switching roles loads a different precompiled automaton, or rebuilds one in about 20ms when the thesaurus has changed

Precompiled Automata

Terraphim compiles the automaton at build time, not runtime:

- The vocabulary is defined in JSON/YAML
- A build script compiles it into a binary automaton
- At runtime, the automaton is loaded from disk
- No parsing, no construction, no setup cost

Start-up is the time to load the precompiled automaton from disk; on the author's development machine that is tens of milliseconds for the 15MB clinical build, against seconds for a model and index load.

How the Graph Scorer Works: Connectivity-Based Ranking

Once the Aho-Corasick automaton finds matching terms, how does Terraphim rank the results?

The Scorer Algorithm

1. **Find all matching terms** in the document (via Aho-Corasick)
2. **Look up each term** in the knowledge graph
3. **Traverse relationships** to find connections to the query
4. **Score by connectivity**: More connections = higher score
5. **Return ranked results**

Concretely, the author's scorer uses: edge weight 1.0 for curated edges; confidence 1.0 for curated edges and 0.6 for edges extracted automatically and not yet reviewed; a depth penalty of 0.5 per hop beyond the first; a maximum depth of 3; each (term, target) pair

counted once, so duplicate paths do not inflate the score; no normalisation by document length; ties broken by fewest hops, then by the earlier match position. The point is not these particular constants but that every one of them is visible and can be changed in one place. The sketch below is schematic: it shows the shape of the traversal and omits the depth penalty, the duplicate suppression and the tie-break, which the production scorer applies as described.

```
def score_document(query, document, knowledge_graph):
    # Find terms in document that match query terms
    matched_terms = find_matches(query.terms, document.text)

    total_score = 0
    for term in matched_terms:
        # Look up term in knowledge graph
        term_node = knowledge_graph.get(term)

        # Find paths to query entity
        paths = graph.find_paths(term_node, query.entity, max_depth=3)

        # Score by path strength
        for path in paths:
            total_score += path.weight * path.confidence

    return total_score
```

What “Connectivity” Means

The scorer distinguishes these connection types:

- **Direct relationship**: Drug treats Disease → high score
- **Indirect relationship**: Drug affects Gene, Gene associated with Disease → medium score

- **No relationship**: Drug unrelated to query → zero score

The score trace records which connections contributed and by how much.

Why This Beats Vector Similarity

Figure 7.4 works the example this section argues from.

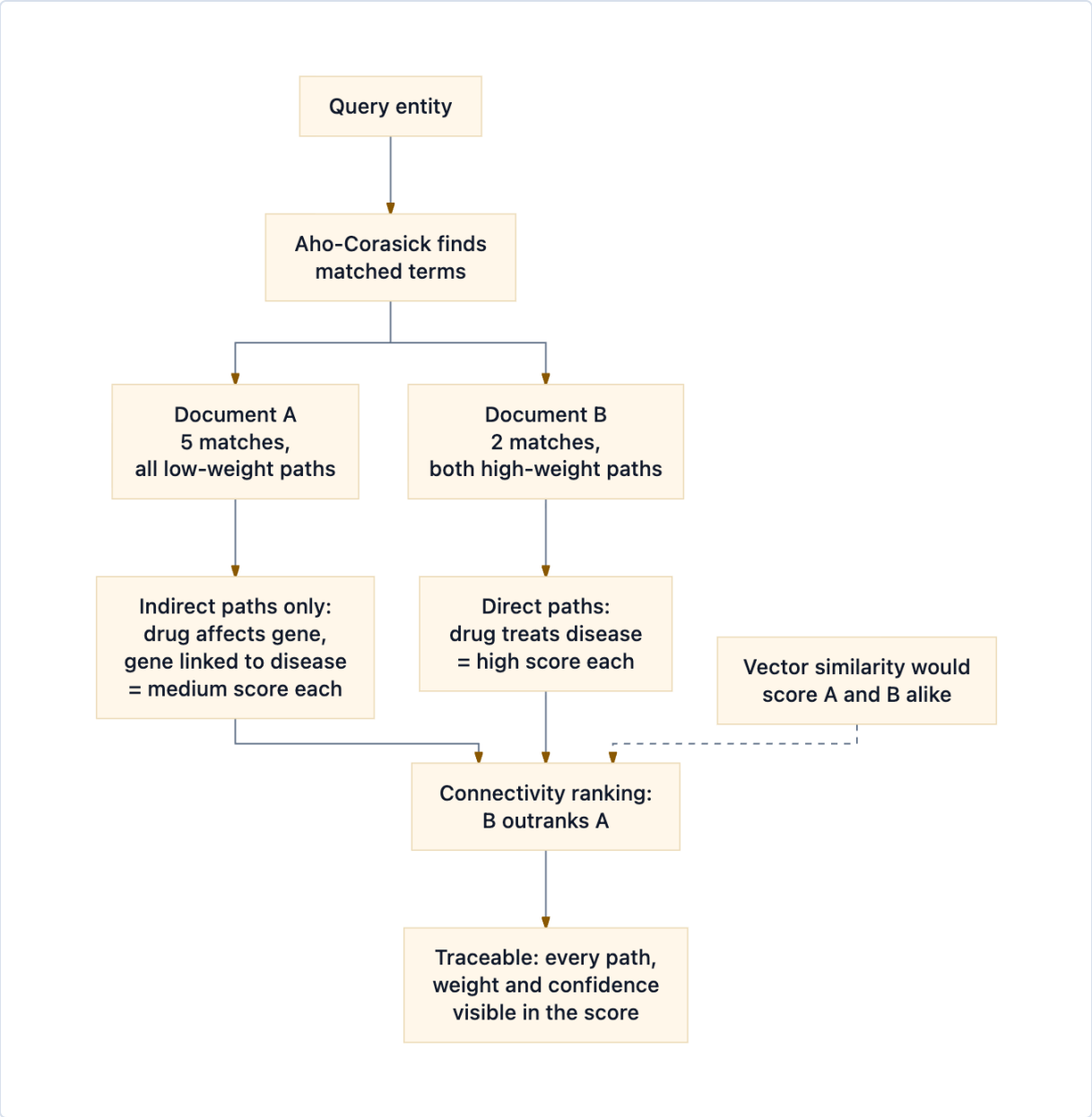


Figure 7.4: Connectivity ranking: a document with two direct high-weight paths outranks one with five indirect low-weight matches, and every path is visible in the score.

Vector similarity tries to capture “overall similarity” in a single number. But documents can be similar in some ways and different in others:

- Document A: matches on 5 terms, all low-weight

- Document B: matches on 2 terms, both high-weight

Which is more relevant? Vector similarity might give similar scores. With the constants above, Document B scores 2 × (1.0 × 1.0 × 1) = 2.0; Document A, whose five matches reach the query entity only through two-hop extracted paths, scores 5 × (1.0 × 0.6 × 0.5) = 1.5. B outranks A, and the trace shows exactly why. A reader can check the implementation against five acceptance tests: an exact surface form matches; a listed synonym resolves to the same canonical term; a term present only in another role's thesaurus does not match; a term absent from the thesaurus produces no match rather than a near miss; and the same query over the same graph returns byte-identical results on repeated runs.

Integration with ProCoder for Semantic File Retrieval

A concrete application of Terraphim's graph embeddings is integrating them with tools that manage LLM context files. ProCoder AI Pack uses manually defined glob patterns to select which files to include in LLM context:

```
# Manual file selection -- fragile and incomplete
context_globs = ["src/auth/*.rs", "src/models/user.rs"]
working_globs = ["src/api/login.rs"]
```

This works for simple tasks but breaks down when the developer does not know which files are relevant, when synonyms span

multiple directories, or when new files are added but glob patterns are not updated.

The Semantic Retrieval Pipeline

Replace manual globs with a Terraphim-driven pipeline:

```
fn semantic_file_selection(prompt: &str, knowledge_graph: &KG) ->
Vec<PathBuf> {
    // Step 1: Extract key terms from the prompt
    let key_terms = extract_terms(prompt);

    // Step 2: Expand via Terraphim (deterministic, <1ms)
    let expanded = terraphim_expand(&key_terms, knowledge_graph);
    // "authentication" -> ["auth", "login", "credential", "session"]

    // Step 3: Search the indexed codebase (deterministic, <1ms)
    let matched_files = terraphim_search(&expanded, knowledge_graph);

    // Step 4: Check for missed concepts (graph traversal)
    let missed = check_concept_gaps(&matched_files, &key_terms,
knowledge_graph);

    // Step 5: Retrieve additional files for gaps (semantic fallback)
    let gap_files = if !missed.is_empty() {
        semantic_search_files(&missed)
    } else {
        vec![]
    };

    // Combine and return
    [matched_files, gap_files].concat()
}
```

Comparison: Manual vs Semantic

Aspect	Manual Globs	Terraphim-Enhanced
File selection	User-defined patterns	Semantic via graph embeddings
Synonym handling	None (literal matches)	Automatic via KG synonym edges
Related terms	Manual addition	Graph traversal for expansion
Missed concepts	No detection	Graph gap analysis flags missing files
Speed	Instant config	<2ms when the deterministic lane answers; the semantic fallback, when gap detection invokes it, adds that lane's latency
Maintenance	Manual updates per task	Self-adapting via KG

The semantic approach is not always better. For a simple, well-understood task, manual globs are faster to set up. For complex queries that span multiple concerns, the design hypothesis is that the pipeline retrieves more complete context than hand-written globs; the author's ProCoder trials were informal and no comparison with a baseline, trial count or scoring criterion is reported here, so the hypothesis is stated as one.

Role-Based Configuration in Practice

The role configuration files in the terraphim-ai repository (github.com/terraphim/terraphim-ai, read October 2026) define six roles:

Role	Focus	Synonym Groups

Role	Focus	Synonym Groups
Terraphim AI Developer	Rust-based AI assistant development	Deep: all 20 groups
OpenClaw Developer	TypeScript gateway/agent platform	Standard: 15 groups
Last30Days Skill Developer	Web research skill development	Basic: 8 groups
System Operator	System administration and operations	Minimal: 5 groups
Learning Capture	Personal knowledge management	Deep: all 20 groups
Terraphim Engineer (Default)	General engineering	Deep: all 20 groups

Each role has its own thesaurus – a JSON file mapping terms to synonym groups:

```
{
  "name": "Terraphim AI Developer",
  "data": {
    "terraphim": { "id": 4, "nterm": "ai_assistant" },
    "claude": { "id": 4, "nterm": "ai_assistant" },
    "kimi": { "id": 4, "nterm": "ai_assistant" },
    "agent": { "id": 4, "nterm": "ai_assistant" },
    "rust": { "id": 11, "nterm": "rust_language" },
    "cargo": { "id": 11, "nterm": "rust_language" },
    "search": { "id": 8, "nterm": "retrieval" },
    "query": { "id": 8, "nterm": "retrieval" },
    "lookup": { "id": 8, "nterm": "retrieval" }
```

```
    }
}
```

When the role switches from “Terraphim AI Developer” to “System Operator”, the Aho-Corasick automaton is rebuilt (in ~20ms) with the new role’s synonym groups. The same query produces different expansions depending on which role is active – exactly the context-sensitive behaviour that enterprise AI systems require.

The 20 synonym groups cover: Knowledge, Configuration, Git, AI Assistant, Tool, Storage, Context Management, Retrieval, Failure, Documentation, Rust, JavaScript, Python, Testing, Security, Performance, Build, Execution, Debugging, and Repair. Each group maps multiple surface forms to a single canonical concept, enabling the deterministic precision described in this chapter. Appendix D (Knowledge Graph Schema) provides the full schema reference.

Summary: The Terraphim Approach

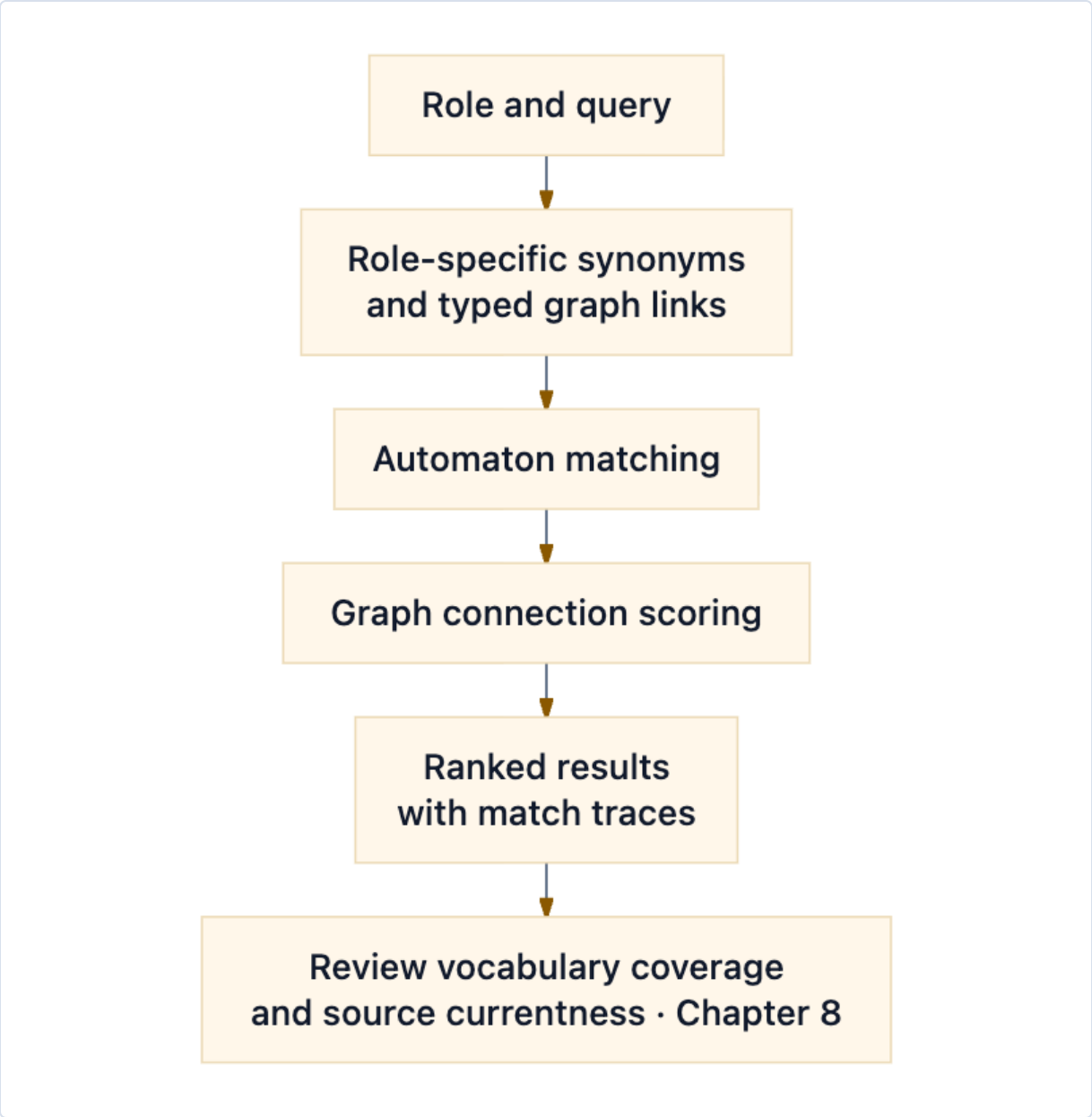


Figure 7.5: Chapter 7 recap: Terraphim combines role-specific vocabulary, deterministic matching, graph ranking and result traces.

Terraphim's retrieval pipeline has seven properties:

1. Relationships retain their types and directions.
2. Queries over a fixed graph and configuration return the same results.

3. Result traces record matches, expansions and graph paths.
4. The scoring layer retains access to the graph's full structure.
5. Each role has its own synonym mappings and relationships.
6. Matching against 1.4M patterns took under 1ms in the author's measurements.
7. Ranking uses the strength of graph connections.

For queries within the reviewed vocabulary, this implementation supplies repeatable matching and inspectable ranking. Its coverage and the author's timing measurements remain bounded by the configurations described above. Chapter 8 takes responsibility for the definitions and relationships that make the trace meaningful; an implementation cannot curate its reference merely by returning the same answer twice.

Key Sources

- Terraphim AI implementation and the author's measurements described above
- Alfred Aho and Margaret Corasick, "Efficient String Matching: An Aid to Bibliographic Search", *Communications of the ACM* 18(6), 1975
- MedGemma evaluation results (Chapter 9), where the automaton described in this chapter is the entity extractor; the results themselves are discussed there, not here

- Terraphim role-based graph configuration patterns (kimiko-terraphim-kg-config)
- ProCoder AI Pack documentation and integration patterns
- Medvedeva, “Complex Descriptions and Meta-Languages” (Ontologics and Communication, School of Systems Management, 2021) – formal role model (concern/preference/intention), ISO 42010 viewpoints

Chapter 8: Building a Knowledge Graph for Context

Chapters 5 to 7 made the case for knowledge graphs and for deterministic retrieval over them. This chapter explains how to build a graph for your own context. It argues for **vocabulary-first construction, from a curated list of terms, rather than extraction-first construction, in which a model pulls entities out of documents**. Curating the vocabulary at build time makes its definitions reviewable at query time. Extraction-first is faster to start: the indexing stage of Microsoft GraphRAG begins with model-driven entity and relationship extraction (Edge et al., 2024), and the other GraphRAG tools the author has reviewed follow that pattern for their first graph. The sections below examine the later review and maintenance costs.

The running example is the author's own knowledge graph for a CTO's working repository, about 500 terms across personal, code and research roles. It illustrates the curation work at a scale one person maintains, well below the size of SNOMED CT.

Defining Nodes

Figure 8.1 shows the sequence this chapter follows, section by section.

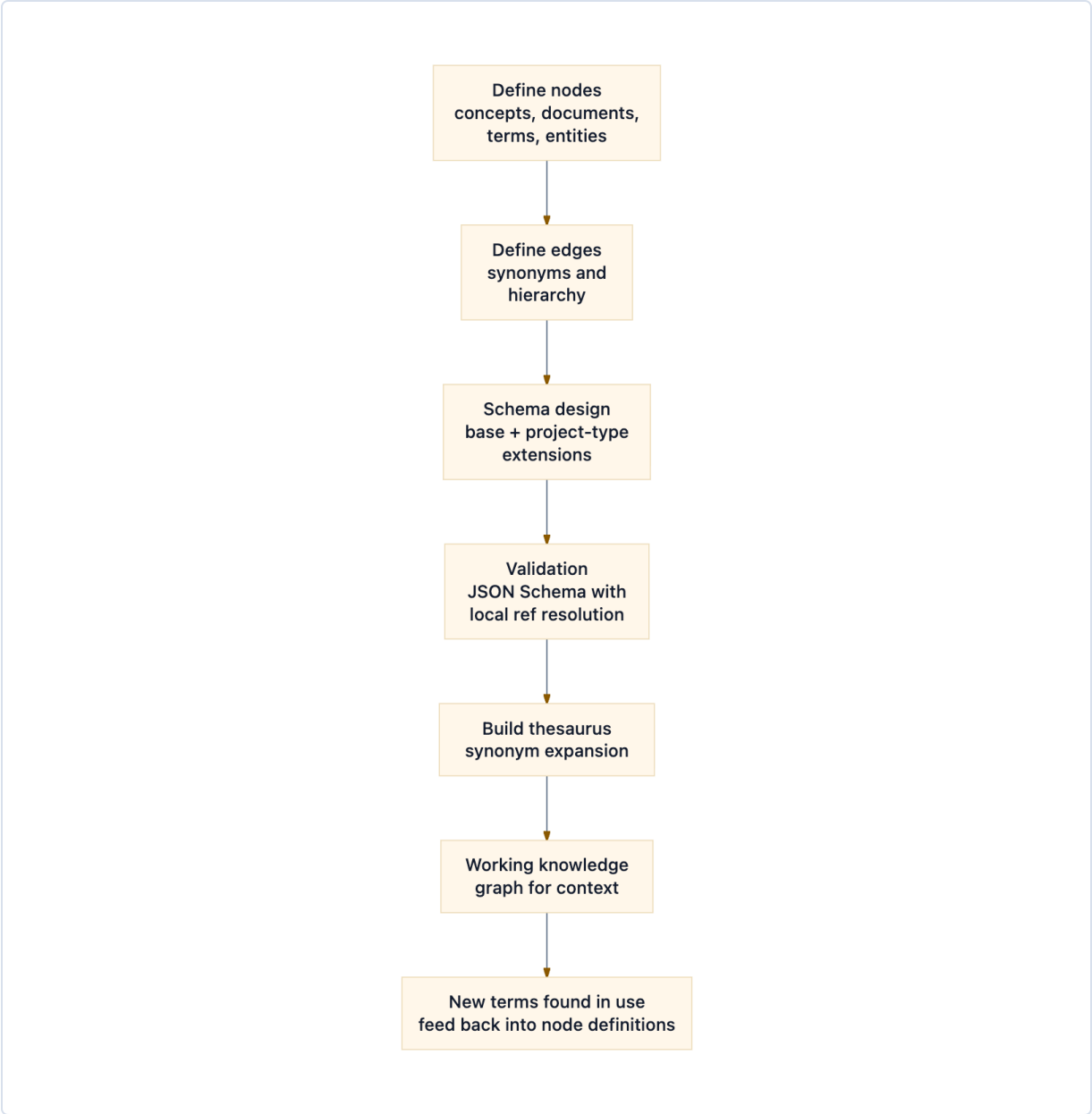


Figure 8.1: The build sequence for a context knowledge graph, from node definitions to a working graph that feeds new terms back into the vocabulary.

A knowledge graph starts with nodes, and the first decision is what kinds of node to allow. The five that recur across domains:

Node type	What it stands for	Example

Node type	What it stands for	Example
Entity	A real-world object the system reasons about	Drug, Patient, Order
Concept	An abstract idea that organises other nodes	Compliance, Security, Quality
Term	A unit of vocabulary, with its surface forms	“myocardial infarction”
Document	A container of content the graph indexes	Specification, Runbook, API doc
Event	Something that happened	OrderPlaced, PaymentReceived

The distinction that matters most in practice is between terms and everything else. A term is a node that exists so that text can be matched to it; it carries synonyms and spelling variants, and the automaton of Chapters 6 and 7 is built from terms. Entities, concepts and events are what the terms point at. A graph that conflates the two ends up with “heart attack” and “myocardial infarction” as separate entities with a relationship between them, when they are one entity with two surface forms.

Vocabulary-first against extraction-first

There are two ways to decide which nodes exist. Extraction-first hands documents to a model and keeps the entities and relationships it returns. Vocabulary-first defines the terms explicitly, then maps documents onto them.

Extraction-first is faster to a first graph, and for a corpus nobody understands it may be the only way to start. Its cost, in the systems the author has reviewed, is that the vocabulary is whatever the model decided it was, which can vary between runs and model versions and can be audited only by reading the output. The author's position, and the practice the rest of the chapter describes, is vocabulary-first: the team controls what the terms mean, changes to meaning are explicit commits, and the graph documents itself because every node was put there by someone who could say why. The cost is the curation work, which never ends, and a blind spot for terms nobody thought to define, which Chapter 22's hybrid pipelines exist to cover.

In the author's repository the vocabulary is small enough to list by kind: entities are people and systems (the author, OpenClaw, Terraphim, the Git hosts); concepts are the architectural patterns the repository is about (knowledge graph, meta-cortex, concept mining, skill, agent, multi-agent); terms are the technical vocabulary that appears in notes (pagerank, memory, session, superposition). Five hundred of them cover the operating context of one role. That number is not a target; it is what fell out of curating one repository for a year.

Defining Edges

Edges are typed relationships between nodes, and the types are few:

Edge type	Meaning	Example

Edge type	Meaning	Example
synonym	Two surface forms of one thing	“heart attack” and “myocardial infarction”
broader / narrower	Hierarchy	migraine is narrower than headache disorder
related	Associated but neither synonym nor hierarchy	drug and side effect
causes	Causal	input and output
precedes	Temporal order	OrderPlaced then OrderShipped

Synonym edges are the foundation, because they are what deterministic search runs on. The thesaurus pattern of Chapter 5 groups them per term:

```
{
  "term": "myocardial infarction",
  "synonyms": ["heart attack", "MI", "cardiac infarction"],
  "related": ["coronary artery disease", "angina"],
  "broader": "cardiovascular disease",
  "narrower": ["STEMI", "NSTEMI"]
}
```

A note on representation, because the chapter uses the word synonym three ways. In storage, a synonym is a surface form listed inside its term’s thesaurus entry, as above; in the graph, each listed surface form is itself a term node (surface forms are not a separate node type) with a synonym edge to the canonical term; at retrieval, every surface form is compiled into the automaton and resolves to

the canonical term. One entry, three views of it. `related` does not: it is a hint for expansion and for the graph scorer, not a matching rule. The most common mistake in a young thesaurus is putting related terms under synonyms because the author wanted a query for one to find the other; the result is a graph that cannot tell a migraine from stress.

Hierarchy edges are what make “show every kind of migraine” a traversal rather than a list someone maintains. In the author’s research vocabulary the hierarchy is shallow and the synonym groups do most of the work: *research, investigation, study, analysis, inquiry* resolve to one concept, *compare, contrast, evaluate, assess, review* to another, so that a note filed under any of those words is found by a query using any other.

Schema: a Base and Extensions

A graph that will outlive its first author needs a schema, and the schema should have two layers: a base that every project shares, and extensions per project type.

The base schema defines what a node and an edge are. In JSON Schema terms, a node requires an identifier, a name and a type drawn from the five above, and may carry a description and metadata; an edge requires a source, a target and a relationship type drawn from the list above, and may carry a confidence between zero and one. That is the whole base. Everything specific to a domain

goes in an extension, which adds entity types and relationship types without changing the base:

```
{ "project_type": "healthcare",
  "extensions": { "entity_types": ["drug", "disease", "procedure",
"anatomy"],
                  "relationship_types": ["treats",
"contraindicated_for", "has_active_ingredient"] } }
```

An e-commerce extension adds product, order, customer and inventory, with contains, ships_to and purchased. The two extensions share nothing but the base, which is what lets a validator written once check both.

Why the layers must stay separate

Medvedeva (School of Systems Management, 2021) gives the layering a formal justification through the idea of a meta-language: a language introduced to describe another language. The chain here runs from the object in the world (a drug), to its description in the graph (a node), to the schema that says what a valid node is (JSON Schema, a meta-language for the graph), to the specification that says what a valid schema is (the JSON Schema specification, a meta-language for the meta-language). Each level describes the one below it.

Keep these levels separate. Data belongs in the graph, validation rules in the schema or validator, and project-specific entity types in extensions to the base schema. If graph data can alter its own validation rules, an edit can bypass those rules. If the base schema

contains one domain's vocabulary, sharing it with another domain becomes harder. Separate checks for each level support the validation pipeline of Chapter 14.

Validation

A graph with invalid structure gives confident answers from broken data, which is the failure a graph was meant to prevent. The author validates with `ajv` against the base schema and the active extension, with local `$ref` resolution so that the schema can be split across files. Two further checks are schema-expressible and live there: type consistency (each node type's required fields, by conditional schema) and the allowed edge types. Two are graph-wide and need a procedure, because a document validator sees one node at a time:

1. No orphan nodes: every node has at least one edge, or it is a term nobody uses
2. No synonym cycles: a chain of synonym edges never returns to its start with a different canonical term

A third procedural rule, explicit directionality, records when A relates to B but B deliberately does not relate to A, so that asymmetry is a decision rather than an omission.

The author's repository graph passes all of these as of this writing: no orphans, consistent types, bidirectional synonyms where intended. Those results come from running the checks on the graph described

here, with the schema in the repository, and they are reported here so that a reader who copies the pattern knows what “valid” meant.

Building the Thesaurus

The thesaurus is where most of the ongoing work lives. Five patterns cover nearly every entry:

Pattern	Example	Where it goes
Exact synonym	myocardial infarction, heart attack	`synonyms`
Acronym	MI, MVP	`synonyms`, with the expansion as canonical
Spelling variant	colour, color; organise, organize	`synonyms`
Role-specific sense	client (sales), subscriber (support), party (legal)	per-role thesaurus, Chapter 7
Related, not synonymous	migraine and stress	`related`

The build process repeats on a cadence (the author runs it monthly and after every incident that involved a missed term): list the fifty to a hundred most important concepts; add the synonyms you know; validate; run the queries people actually ask and read what comes back; add the terms that were missing. The last step is the feedback edge in Figure 8.1, and it is where the vocabulary-first approach earns back the time it cost: each miss is a term someone adds with a reason, and the reason is in the commit.

From Domain Model to Graph Schema

Chapters 4 and 5 introduced entities, relationships and constraints, and how a graph carries them. Mapping a domain model onto a graph schema is a five-step exercise, shown here for a small commerce domain.

1. **Entities** come from the domain model: Product, Customer, Order, LineItem, Payment, Shipment.
2. **Attributes** attach to each: a Product has a name, a SKU, a price, a category and a stock level; an Order has an identifier, a customer, a status and a total.
3. **Relationships** connect them: a Customer places an Order, an Order contains LineItems, a LineItem references a Product, an Order has a Payment and a Shipment.
4. **Constraints** are the rules that must always hold: an order's total is the sum of its line items, an order's status is one of a fixed set, a payment's amount is positive.
5. **Mapping** turns the first four into graph structure:

```
{ "nodes": [ {"id": "Product",  "name": "Product",   "type":
"entity"},
             {"id": "Customer", "name": "Customer",  "type":
"entity"},
             {"id": "Order",    "name": "Order",     "type": "entity",
"metadata": {"attributes": ["id", "customer_id", "status", "total"]}},
             {"id": "LineItem", "name": "Line item", "type": "entity",
"metadata": {"attributes": ["quantity", "unit_price"]}},
             {"id": "Payment",  "name": "Payment",   "type":
"entity"},
```

```
            {"id": "Shipment", "name": "Shipment",  "type": "entity"}
],
  "edges": [ {"from": "Customer", "type": "places",     "to": "Order",
"cardinality": "one-to-many"},
            {"from": "Order",    "type": "contains",   "to":
"LineItem", "cardinality": "one-to-many"},
            {"from": "LineItem", "type": "references", "to":
"Product",  "cardinality": "many-to-one"},
            {"from": "Order",    "type": "paid_by",    "to":
"Payment",  "cardinality": "one-to-one"},
            {"from": "Order",    "type": "shipped_by", "to":
"Shipment", "cardinality": "one-to-many"} ],
  "constraints": [ {"type": "sum", "attribute": "Order.total",
                    "parts": "LineItem.quantity *
LineItem.unit_price", "over": "Order contains LineItem"} ] }
```

Attributes are carried as metadata on the entity node rather than as nodes of their own, which keeps the five node types of the base schema intact. The grounding gate of Chapter 9 also needs the constraints. Recording that an order contains line items does not let it check the total; the graph must also record that the total equals their sum.

Summary

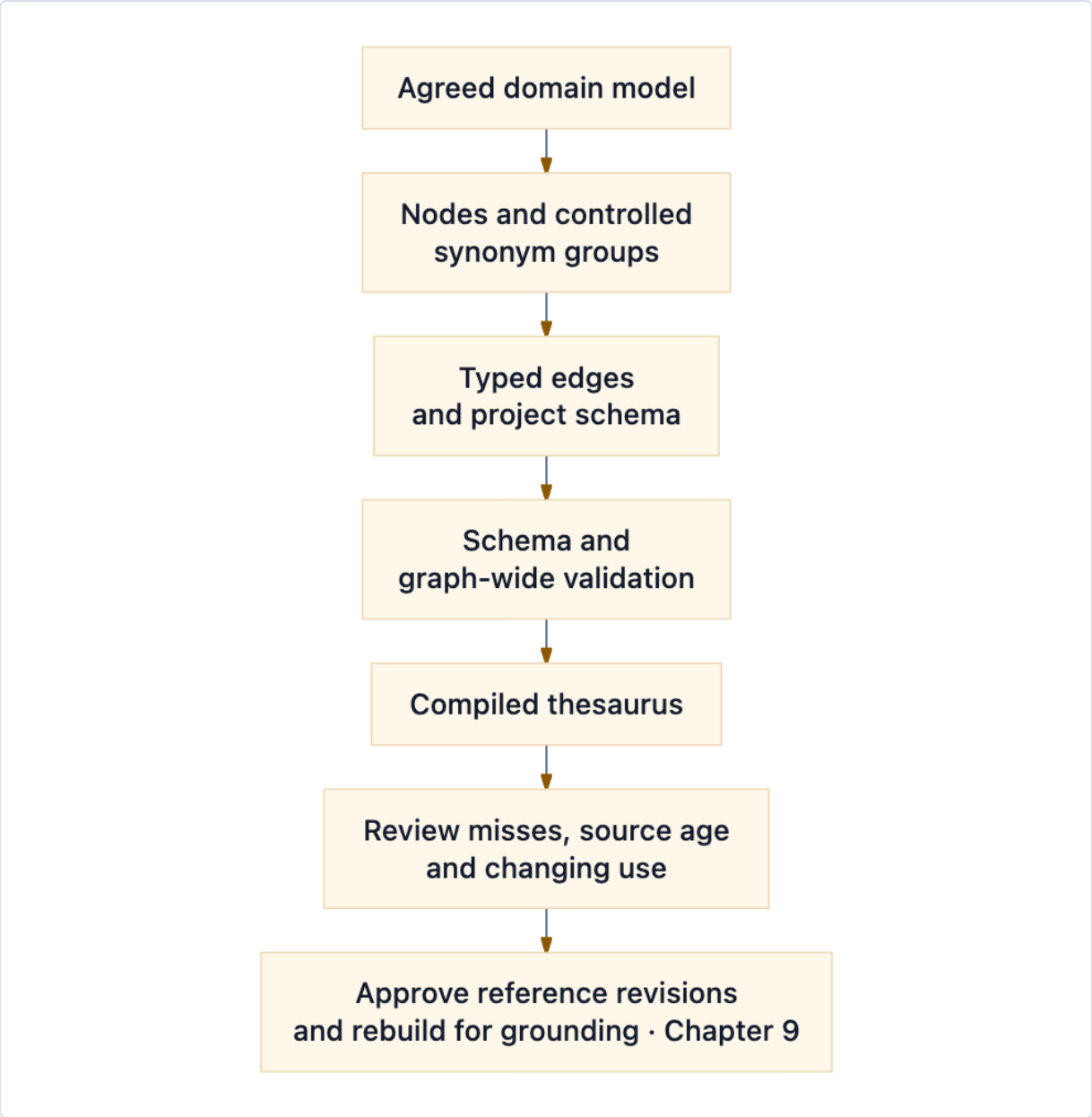


Figure 8.2: Chapter 8 recap: Graph construction is a repeatable cycle of modelling, validation and vocabulary maintenance.

Building a knowledge graph for context is a structured, repeating process:

1. Define nodes vocabulary-first, keeping terms distinct from the things they name

2. Define edges with a small set of types, and keep related terms out of the synonym set
3. Design a base schema and per-project extensions, and never mix the levels
4. Validate with a schema validator, then with the graph-wide checks the schema cannot express: orphans, synonym cycles and deliberate asymmetry
5. Build the thesaurus from five patterns and feed misses back into it
6. Map the domain model onto the graph, constraints included

The resulting graph has reviewed definitions, typed edges and validation checks. Its usefulness still depends on maintaining those definitions as the domain changes.

In the maintenance case, adding a synonym for “restored” must not erase the distinction between attempted repair, completed execution and delivered result. Hand the versioned reference and its coverage limits to Chapter 9’s grounding gate. A newly extracted claim enters review before it can support release.

Key Sources

- The author’s knowledge graph for a CTO working repository (about 500 terms across personal, code and research roles); the validation results reported are from it
- JSON Schema, draft-07, and the `ajv` validator

- Edge et al., “From Local to Global: A Graph RAG Approach to Query-Focused Summarization” (arXiv 2404.16130, 2024), for the extraction-first indexing stage
- Terraphim AI thesaurus and role configuration (github.com/terraphim/terraphim-ai)
- SNOMED CT concept model, for the scale the patterns here must reach in a clinical domain
- Medvedeva, “Complex Descriptions and Meta-Languages” (Ontologics and Communication, School of Systems Management, 2021): the meta-language principle behind schema layering
- Chapters 4, 5, 7, 9 and 22 of this book

Chapter 9: Knowledge Graph Grounding as Safety Gate

Chapters 5 to 8 described knowledge graphs as precise, explainable context. This chapter uses the graph as a **grounding gate**: before releasing a model's output, the gate checks parsed claims whose entities the vocabulary recognises. Its coverage depends on that vocabulary and the claim parser. The graph supplies the reference; the gate applies the release policy.

By the end of the chapter you should be able to describe the gate pattern, choose between its four modes for a given domain, read a grounding trace, and sketch the pipeline for your own vocabulary. The running example is the author's entry in the MedGemma evaluation, where the gate caught a tenfold dosing error.

The MedGemma Case Study

In 2025 Google released MedGemma, a medical language model. The Terraphim team built a knowledge-graph grounding pipeline around it and ran MedGemma's outputs through it. The results are the author's own, from an internal evaluation rather than an independently judged competition, and they should be read as a proof of concept with a complete evidence trail, not as clinical validation.

What the raw model did

Unconstrained, MedGemma produced four kinds of dangerous output in the evaluation cases: a dosage error, osimertinib at 800mg where the licensed dose is 80mg, a tenfold overdose; vague class-level recommendations ("a BRAF inhibitor") where a specific approved drug was called for; substitutions that were contraindicated by the patient's other conditions; and recommendations with no evidence trail, no trial or guideline the clinician could check.

The pipeline

Figure 9.1 shows the six stages and where the gate caught the dosage error.

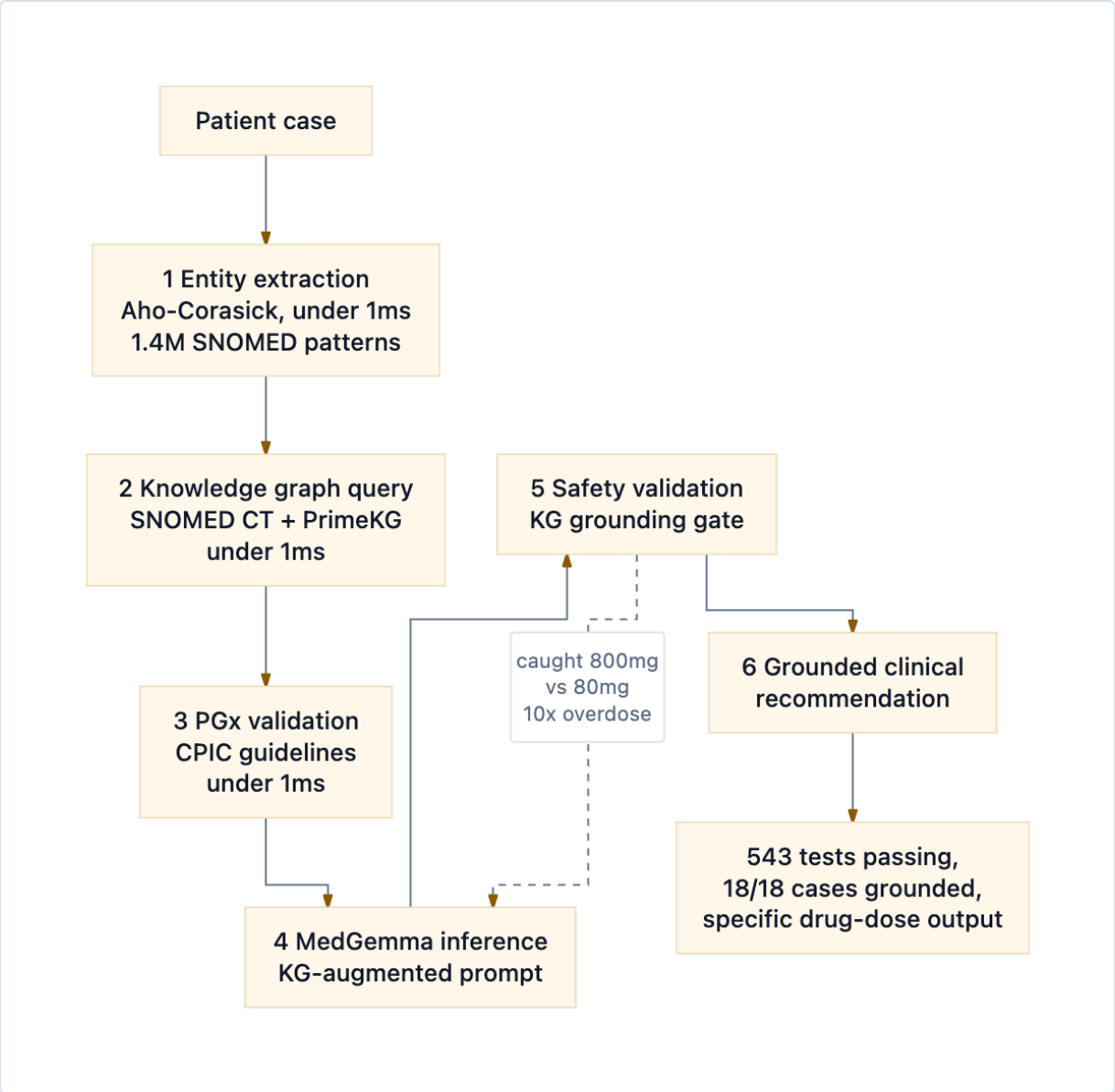


Figure 9.1: The six-stage grounded pipeline from the MedGemma evaluation: deterministic extraction, graph query and guideline validation before the model, a grounding gate after it.

Three deterministic stages run before the model and one after it. Entity extraction uses an Aho-Corasick automaton (Chapters 6 and 7) built from about 1.4 million SNOMED CT surface forms (the author's count for the 2024 International Edition with its description synonyms). The knowledge-graph query resolves the extracted entities against SNOMED CT and PrimeKG, the Precision Medicine

Knowledge Graph, which is where drug-disease and drug-gene relationships come from; SNOMED CT itself is a clinical terminology with is-a and attribute relationships, not a treatment database. Pharmacogenomic validation checks the candidate drugs against CPIC guidelines. MedGemma then answers a prompt augmented with the grounded context. Its draft passes through the grounding gate, and only a grounded recommendation leaves.

The results, defined once

The evaluation has two parts, and the numbers in the rest of the chapter refer back to this paragraph.

- **The test suite**: 543 automated tests over the pipeline's components (extraction, graph query, guideline lookup, gate decision, trace generation), each asserting a specific input produces a specific output. "Passing" means every assertion held on the pipeline at the evaluated commit.
- **The evaluation cases**: 18 clinical scenarios with a known correct recommendation. "Grounded" means the gate matched every entity in the final recommendation to the graph and the recommendation agreed with the reference. 18 of 18 were grounded; the osimertinib case is the one where the raw model's 800mg was replaced by 80mg, the once-daily dose used in the FLAURA trial (Soria et al., *New England Journal of Medicine*, 2018).
- **Overhead**: the three deterministic stages and the gate together added about three milliseconds per case on a laptop-class

machine (author's measurement, 2025), against seconds to minutes of model inference.

The Gate Pattern

Figure 9.2 shows the gate as a loop around the model rather than a filter after it.

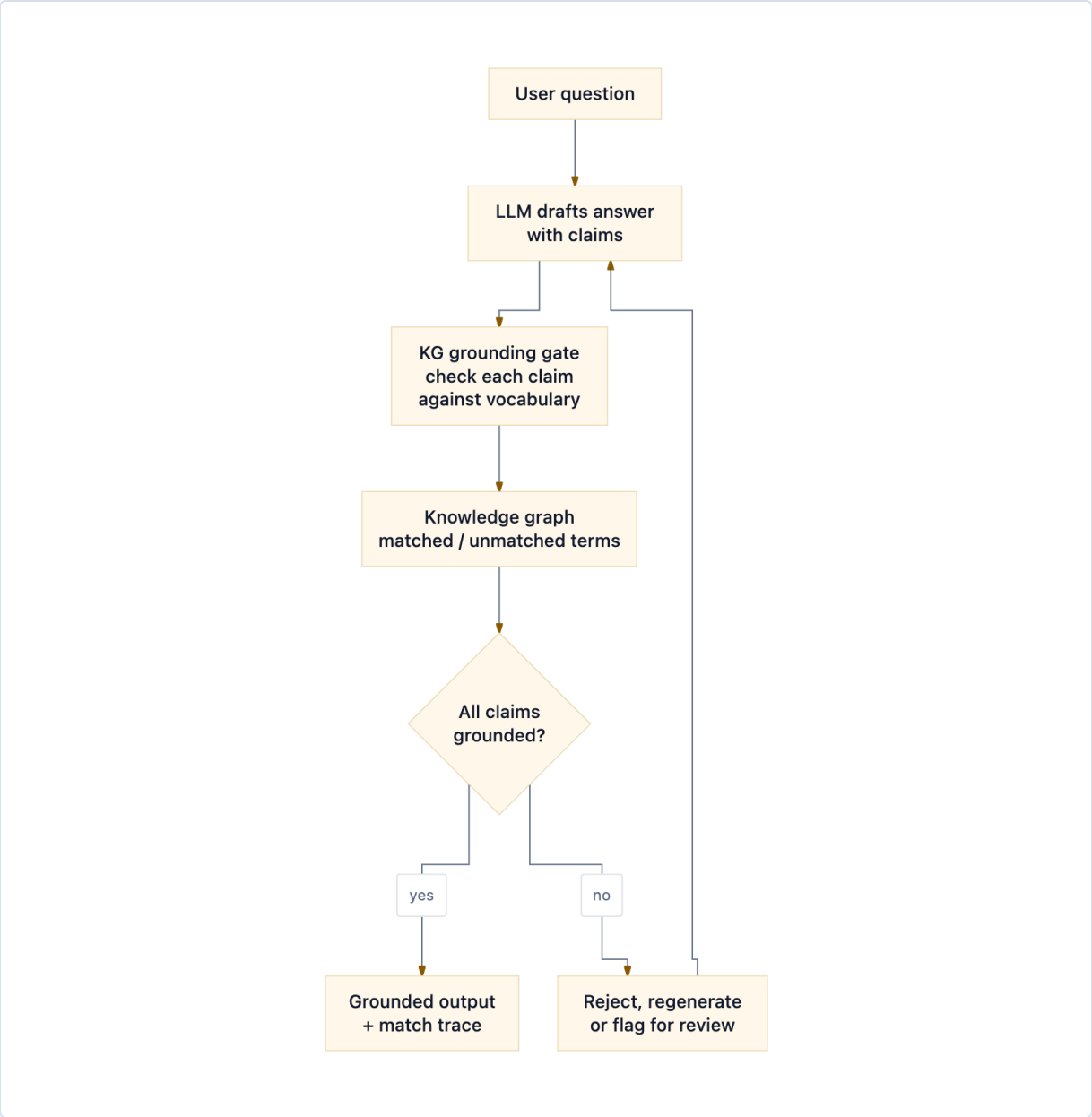


Figure 9.2: The grounding gate: every claim in a draft answer is checked against the vocabulary, and ungrounded claims send the draft back.

The pattern is: the model generates (untrusted); entities are extracted from the output; each entity and each claimed relationship is looked up in the graph; the output passes if everything resolves and is sent back, corrected, annotated or escalated if not. A **safety gate** is any check that stands between a model and a consequence; a

grounding gate is the specific safety gate that uses a knowledge graph as its reference, and it is the only kind this chapter is about.

The gate checks whether an explicit relationship is present in the graph (Chapter 6), constrains the model's draft and records the lookups behind each decision. Its default is to flag an entity the extractor recognises but the graph cannot resolve. Coverage remains the limit: a claim phrased in vocabulary the thesaurus lacks is invisible to the gate, which is why Chapter 22 considers hybrid pipelines.

Four modes, and when to use each

A gate can respond to a failed check in four ways. The choice is a design decision per domain, not a property of the gate.

Mode	What happens	Use when
Reject	The output is refused and regenerated	The error is unambiguous and a wrong answer is worse than no answer (dosage above a hard maximum)
Correct	The offending value is replaced from the graph	The graph holds the authoritative value and the rest of the output is sound (a licensed dose)
Annotate	The output passes with a warning attached	The output is valid but carries a caveat the reader must see (a contraindication for this patient)

Mode	What happens	Use when
Escalate	The output is held for a person	The graph cannot decide (an unlisted drug) or the stakes exceed what automation may settle

In the evaluation, the osimertinib case ran in correct mode; a drug absent from the approved list ran in escalate mode. A rule of thumb: correct only when the graph's value is unambiguous; otherwise reject or escalate, because a silent correction the model did not make is a context mutation of the kind Chapter 19 warns about, and it must be logged.

From the Gate to the Regulator

The four modes determine what the user sees; the trace records the checks for later review. Regulators of medical software, financial services and similar fields ask why a system made a recommendation, what evidence supported it and whether the decision path can be followed. A vector similarity score alone cannot answer those questions. The grounding trace links extracted claims to graph entries and their sources.

For every grounded output the pipeline records a trace of this shape:

```
RECOMMENDATION: Osimertinib 80mg daily
EVIDENCE TRACE
  1. Entity "osimertinib" extracted from the model output
  2. Resolved in PrimeKG; licensed dose attribute 80mg once daily
  3. Source: osimertinib prescribing information (FDA label: 80mg once
```

```
daily); trial evidence FLAURA (Soria et al., NEJM 2018)
  4. No contraindication found against the patient profile
GROUNDED: 4 of 4 claims matched
VALIDATION: PASSED
```

In the trace, PrimeKG supplies the graph relationships, the prescribing information is the authoritative source for the licensed dose, and FLAURA is the trial evidence behind the label; the gate corrects only from the authoritative source. The last two lines are counts, not confidences. The final match is deterministic: either a claim matched the graph or it did not, and the trace says which. The uncertainty lives upstream, in the extractor's coverage and the parser's recall, and those are measured separately (Chapter 14) rather than folded into a confidence on the match. That is what makes the artefact useful to a reviewer. It is designed to answer the questions regulators ask; whether a given regulator accepts it is a matter for that regulator, and the chapter does not claim more.

Building the Pipeline

The components are small. What follows is a compact Python sketch of the shape of each, written against the `pyahocorasick` library for the automaton; a production pipeline adds error handling, logging and the trace store, and it keeps the vocabulary and the graph under the same change control as the code (Chapter 21). The sketch assumes the model's draft has already been turned into structured claims (entity, attribute, value). That step is the hard one in practice: constrain the model to emit claims in a schema first; fall back to a

domain parser for drug, dose and frequency; and treat negation (“not contraindicated”), units (mg against mcg) and patient context (renal dosing) as the cases the parser must have tests for, because a claim parsed wrongly is grounded wrongly.

```
import ahocorasick

def build_automaton(vocabulary):            # vocabulary: iterable of
(surface_form, entity_id)
    a = ahocorasick.Automaton()
    for surface, entity in vocabulary:
        a.add_word(surface.lower(), entity)
    a.make_automaton()
    return a

def extract_entities(text, automaton):
    return {entity for _, entity in automaton.iter(text.lower())}

def check(entities, claims, graph):          # claims: {(entity,
attribute): value} parsed from the output
    results = []
    for entity in entities:
        if entity not in graph:
            results.append(("NOT_FOUND", entity, None)); continue
        for (e, attr), value in claims.items():
            if e != entity: continue
            ref = graph[entity].get(attr)
            results.append(("OK" if ref == value else "MISMATCH",
entity, (attr, value, ref)))
    return results

DECISION = {"NOT_FOUND": "ESCALATE", "MISMATCH": "CORRECT"}   # per-
domain policy table

def decide(results):
    for status, entity, detail in results:
        if status in DECISION:
```

```
            return DECISION[status], entity, detail
    return "PASS", None, None
```

The policy table is where the four modes live, and it is the part that differs between domains: a clinical pipeline maps a dosage mismatch to CORRECT only when the graph's dose is licensed, and to ESCALATE otherwise. Before any patient-facing use, a clinical correction also requires an authoritative source for the substituted value, a second validation pass over the corrected output, a logged trace, and either a certified clinical workflow or a clinician in the loop; the proof of concept in this chapter had the first three and not the fourth. The trace generator, omitted here, is the function that turns `results` into the evidence trace above and writes it to an append-only log with the model output and the decision.

Two practical notes. The automaton for 1.4 million surface forms builds in seconds and is about 15MB serialised (author's measurement, same build as Chapter 7); build it once per vocabulary release and load it, do not rebuild per request. And keep the extractor's vocabulary and the graph's node set in step: an entity the extractor knows but the graph lacks is the NOT_FOUND case, and if that happens often the thesaurus and the graph have drifted apart (Chapter 8).

Beyond Healthcare

Other domains suggest applications of the pattern. In finance, a gate could check a recommendation against the instruments the firm is

authorised to offer. In legal work, it could check cited statutes against the intended jurisdiction. In manufacturing, it could compare generated process parameters with equipment limits. The author has not built these applications; each would need its own vocabulary, graph, policy table and evaluation.

Applying This

- Identify the vocabulary your high-stakes outputs are expressed in; if there is no controlled vocabulary, build one first (Chapter 8), because the gate can only check what the extractor recognises
- Decide the policy table before building anything: for each failure kind, reject, correct, annotate or escalate, and who sees escalations
- Log a trace for every decision, including passes, and keep the log append-only
- Measure the NOT_FOUND rate; it is the drift signal between your thesaurus and your graph

Summary

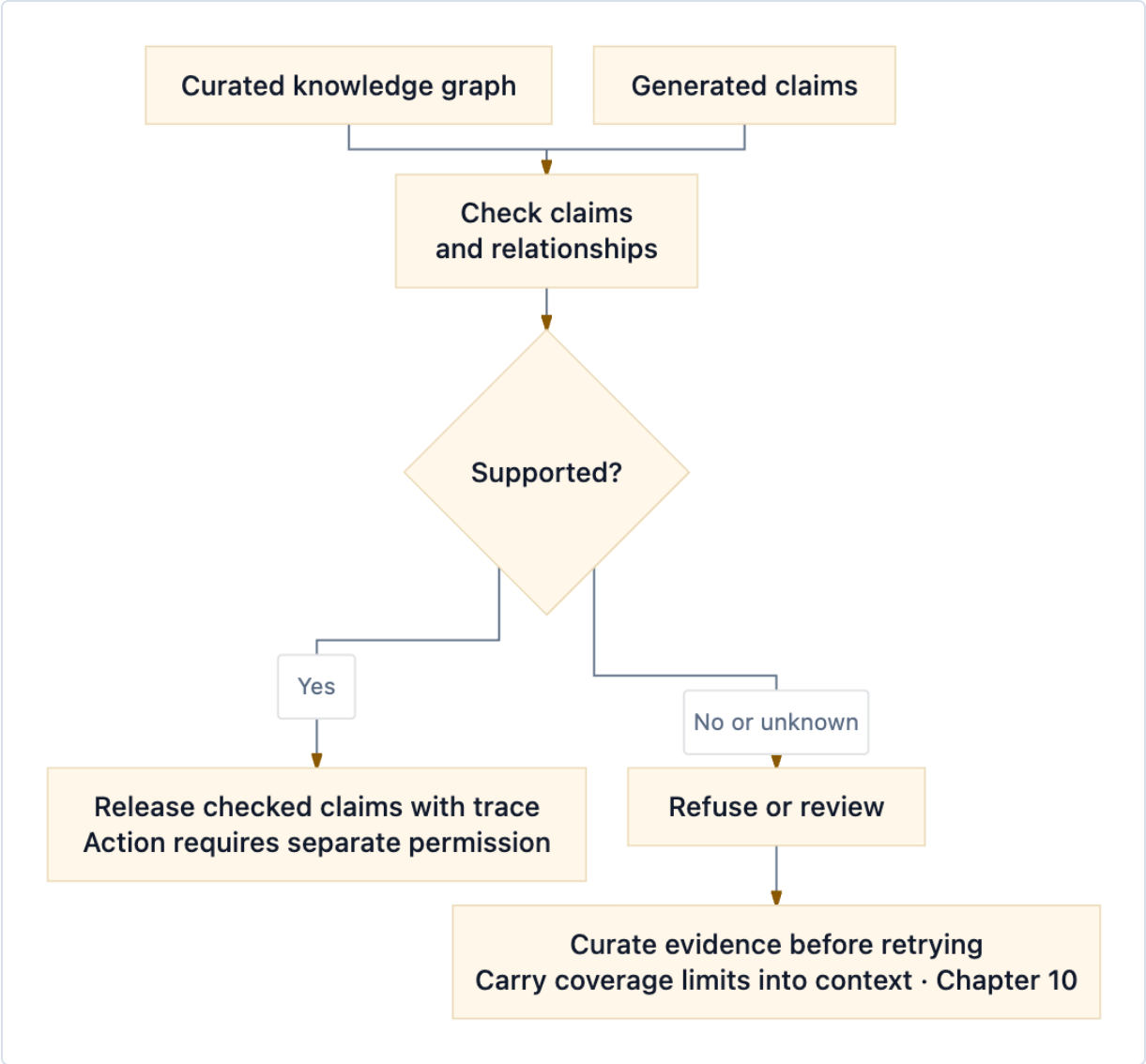


Figure 9.3: Chapter 9 recap: Grounding checks generated claims against curated knowledge and records both accepted and refused results.

The internal evaluation illustrates four properties of the grounding gate:

1. The gate catches what the model gets wrong, as the osimertinib case showed, and refuses what it cannot check
2. Every decision carries a trace of matched claims, which is what a reviewer or regulator can examine

3. The overhead is milliseconds against seconds of inference
4. The results above are from the author's internal evaluation of a proof of concept, not a production deployment; the gap between the two is the operational work of Chapters 14 and 21

We now have an explicit reference and a bounded way to check recognised claims. The maintenance assistant still cannot load every rule, observation and exception on every call. Chapter 10 selects the material needed for the current decision while retaining references to the rest. Neither a grounded claim nor a complete retrieval trace grants permission to change a job.

Key Sources

- The author's MedGemma evaluation pipeline (Terraphim, 2025): 543 automated tests, 18 evaluation cases, traces and timings as defined above; the surface-form count (about 1.4 million), the automaton size (about 15MB) and the per-case overhead (about three milliseconds) are the author's measurements on a laptop-class machine (2025)
- SNOMED CT (clinical terminology) and PrimeKG (Chandak, Huang and Zitnik, *Scientific Data*, 2023) for the graph
- CPIC pharmacogenomics guidelines
- Soria et al., "Osimertinib in Untreated EGFR-Mutated Advanced Non-Small-Cell Lung Cancer" (FLAURA), *New England Journal of Medicine*, 2018

- Google, MedGemma model release (2025); evaluation record: author's pipeline, 2025, 18 cases chosen by the author to cover the dosing and contraindication paths, timings by wall clock per case on a laptop-class machine. The 1.4 million surface forms come from the 2024 SNOMED CT International Edition including description synonyms
- Tagrisso (osimertinib) prescribing information, US FDA label, for the licensed dose
- `pyahocorasick` library documentation
- Chapters 6, 7, 8, 14, 19, 21 and 22 of this book

Chapter 10: Token Budget Architecture

Every LLM has a finite context window. This chapter describes **token budget architecture**: allocating space to instructions, skills, memory and working material, then compressing or unloading content when those allocations are exceeded.

The maintenance case now has reviewed rules, requirements and source observations. Its decision packet can refer to all of them without loading them all. For a proposed repair, retain the current requirement, owner and scope, the evidence needed for that step, and references to the protected cases. Do not trade away a refusal condition or approved operating limit merely to fit the window. If essential context cannot fit, narrow the task or stop before the call.

The 12,000 Token Rule and Progressive Disclosure

A practical rule of thumb: **allocate no more than 12,000 tokens for active context** (out of typical 128K-200K windows). This leaves headroom for the LLM's output and system overhead.

Why 12,000? Modern context windows support 128K to 200K tokens, but response quality does not scale with them. Liu et al. (2023) showed that models lose track of information placed in the middle of

long contexts, and long-context benchmarks such as RULER (Hsieh et al., 2024) show effective context lengths well below advertised ones for most models. The 12K figure is the author's working value, arrived at by measuring the OpenClaw skill set and sessions with the Anthropic tokeniser in 2026 (about twenty skills, a few hundred sessions), and it provides enough room for two or three active skills, conversation history and working context while keeping the signal-to-noise ratio high. Teams should recalibrate it against their own task set and tokeniser; start conservative.

The papers cited establish that quality degrades with context length; they do not establish 12,000 as the right ceiling, and the figure should be read as the author's working value, not a result. The challenge it sets is fitting what is needed into that budget.

The Problem: Flat Context Loading

Naive agent setups that load every skill file at startup pay for all of it on every turn:

- All skills: 50 skills at roughly 1,000 tokens each is 50,000 tokens before any work starts
- All memory: Conversation history and file reads grow throughout the session
- All knowledge: Loading the entire graph includes material unrelated to the task

The skills alone exceed the author's 12,000-token working budget. Truncating the assembled context without priorities can remove

instructions the task still needs.

The Solution: Progressive Disclosure

Progressive disclosure is a design pattern from UI design: show only what's needed at each moment, reveal more on demand.

Applied to context:

1. **Startup**: Load only skill names (~200-300 tokens)
2. **On trigger**: Load full skill content (roughly 1,000 to 2,000 tokens per skill)
3. **On compression**: Reduce to essentials before the LLM call (roughly 500 to 800 tokens per skill)
4. **On demand**: Load additional context when specifically requested

Chapter 11 gives the compression strategies in their priority order: deduplicate, summarise, truncate, drop.

Constitutional Context: What Never Gets Compressed

The loader excludes some content from compression because losing it changes the agent's behaviour in later turns. This chapter calls that content the constitution.

Example Constitutional Files (OpenClaw)

The file names below are the OpenClaw convention (Chapter 20); any agent system has equivalents, and the reader should map them to their own.

File	Purpose	Never Drop
AGENTS.md	Identity, values, operating principles	✓
SOUL.md	Core persona and personality	✓
USER.md	User preferences, context	✓
HEARTBEAT.md	The checklist an unattended agent works through on each scheduled poll	✓ on polls

These files carry the operating rules, the persona and the user facts; without them the agent cannot apply the project's conventions or address the right person. The first three are budgeted in the constitutional allocation below; HEARTBEAT.md is loaded only on autonomous polls and is budgeted under working context for those turns.

Constitutional Principles

1. Preserve the assistant's defined role and operating rules
2. Retain the user context needed for the task
3. Retain the instructions needed to operate its heartbeat and skills

This design treats them as fixed: the loader never passes them to the compression pipeline, however tight the budget.

Layered Loading: Skill Catalogue → Full SKILL.md → Compression

The loader, which the author's systems call the context optimiser, implements a **three-layer architecture**. Compression, in this chapter, means reducing a skill's content to the instructions and examples that matter for the current task while keeping its triggers intact; Chapter 11 gives the strategies.

Layer 1: Skill Catalogue (Names Only)

At startup, load only skill names and triggers:

```
# Skill catalog: ~200-300 tokens for 20 skills
skills:
  web-search:
    name: web-search
    triggers: ["search", "web", "find", "lookup"]
    tier: standard
    estimated_tokens: 800

  terraphim-memory:
    name: terraphim-memory
    triggers: ["memory", "recall", "remember", "kg"]
    tier: core
    estimated_tokens: 1200
```

Token cost: roughly 200 to 300 tokens for a 20-skill catalogue, against roughly 16,000 for the full content of the same skills. These

are the author's measurements of the OpenClaw skill set with the model's tokeniser; your skills will differ.

Layer 2: Active Skills (Full Content)

When a skill is triggered, load its full content:

```
// Activation decision
const activeSkills = skills.filter(s =>
  s.triggers.some(t => userQuery.toLowerCase().includes(t))
);

// Load full content only for activated skills
const skillContents = await Promise.all(
  activeSkills.map(s => readFile(`skills/${s.name}/SKILL.md`))
);
```

Token cost: ~1000-2000 tokens per active skill

Layer 3: Compression (Before LLM)

Before calling the LLM, compress all active skills:

```
function compressContext(skillContents, maxTokens) {
  // Priority: core skills first, then standard
  const sorted = sortByPriority(skillContents);

  let totalTokens = 0;
  const compressed = [];

  for (const skill of sorted) {
    if (totalTokens + skill.compressed_tokens <= maxTokens) {
      compressed.push(skill.compressed);
      totalTokens += skill.compressed_tokens;
    }
  }
```

```
  return compressed;
}
```

Token Tracking and Monitoring Systems

Managing a token budget requires **visibility** into what's consuming tokens.

The Token Tracker

A tracking system monitors context composition:

```
const tokenTracker = {
  constitutional: 0,
  skillCatalog: 0,
  activeSkills: 0,
  memory: 0,
  working: 0,
  buffer: 0,

  budgetFields: ["constitutional", "skillCatalog", "activeSkills",
"memory", "working", "buffer"],
  get total() {
    return this.budgetFields.reduce((sum, f) => sum + this[f], 0);
  },

  warnAt: 12000,      // the working budget: compress on the next turn
  criticalAt: 15000,  // emergency ceiling: compress now and fail
closed if still over

  check() {
    if (this.total > this.criticalAt) {
      triggerCompression();
    }
```

```
  }
};
```

Monitoring Dashboard

Key metrics to track:

- **Total context size**: Should stay under 12K
- **Active skill count**: at most five (the cap Chapter 12 derives), which fits the allocation only when the skills are compressed; two or three fit at full size
- **Memory age**: How old is the conversation history?
- **Buffer remaining**: What's left for this response?

Alerting

```
// Alert when approaching budget
if (tokenTracker.total > tokenTracker.warnAt) {
  notify("Approaching token budget", "yellow");
}

if (tokenTracker.total > tokenTracker.criticalAt) {
  notify("CRITICAL: Triggering emergency compression", "red");
  triggerCompression();
}
```

Allocation Per Context Type: The Budget Breakdown

Figure 10.1 shows the five layers the allocation below is divided among.

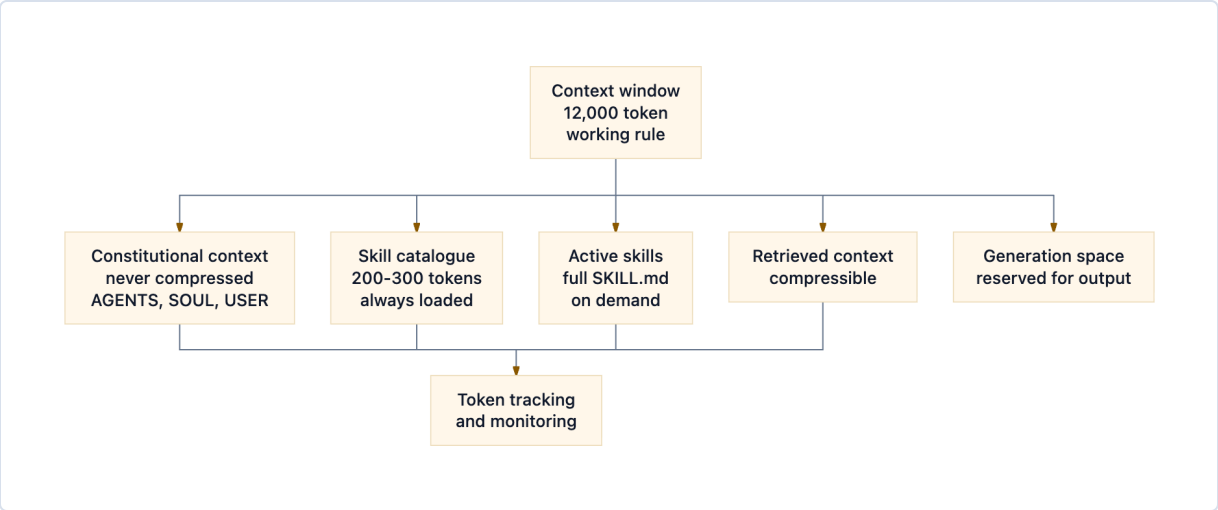


Figure 10.1: The layers inside a 12,000-token working budget and which of them are compressible.

The author's budget assigns an explicit allowance to each context type:

Recommended Allocation (12,000 Token Budget)

Context Type	Allocation	Purpose
Constitutional	2,000	AGENTS.md, SOUL.md, USER.md (never compressed)
Skill Catalogue	300	Names, triggers, tiers
Active Skills	5,000	Two skills at full size (up to 2,500 each), or three to five compressed to 500 to 800 each

Context Type	Allocation	Purpose
Memory	2,000	Recent conversation, key facts
Working	2,000	Current task context, file state
Buffer	700	Margin before the hard budget; the model's own output sits outside the working budget

When categories exceed their allocation the loader applies a fixed priority: keep the constitutional tier; keep the current working state; compress active skills; trim memory from the oldest turn; and if the total is still over the emergency ceiling, stop and ask for clarification rather than send a truncated prompt.

The allocation protects operating rules and user context, loads skills on demand and gives recent memory priority. The buffer absorbs variation in working context; output capacity is reserved separately from this budget.

Adjusting for Different Models

Window class	Working budget the author uses	Constitutional
128K	12,000 to 15,000	2,000 to 3,000
200K	15,000 to 20,000	2,000 to 3,000

The budget grows with the window, but much more slowly than the window does, because the degradation evidence above applies to what is in the context, not to what the model could hold. The constitutional allocation stays roughly constant whatever the window.

Summary: Budget as Infrastructure

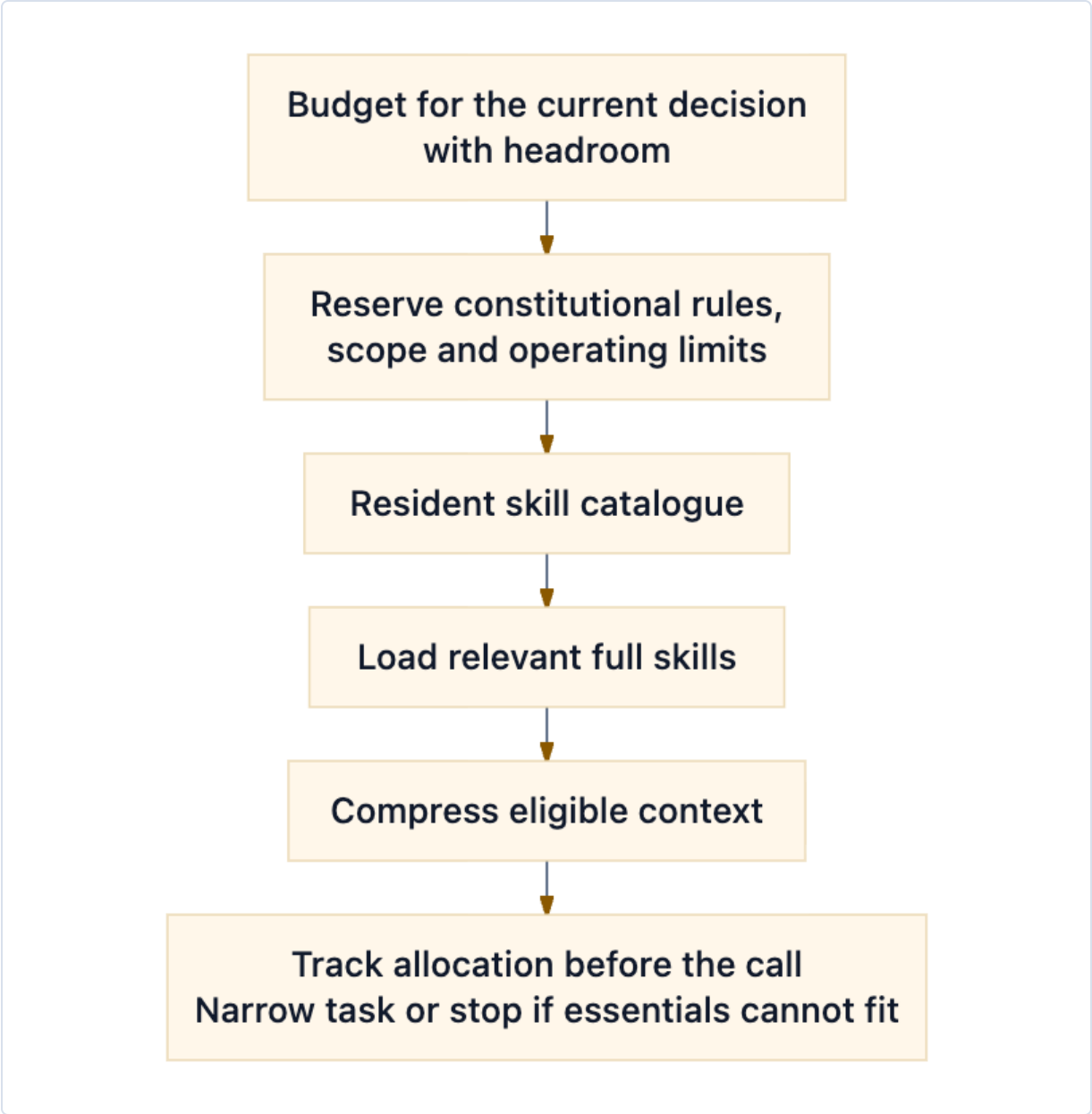


Figure 10.2: Chapter 10 recap: Allocate the context budget, protect essential rules and load other material progressively.

A token budget turns context from something that fills the window into something that is allocated:

1. **12,000 token rule**: Practical limit with headroom
2. **Progressive disclosure**: Load only what's needed, when needed

3. **Constitutional context**: Lock what must never be lost
4. **Three-layer loading**: Catalogue → Full → Compression
5. **Token tracking**: Visibility into context composition
6. **Explicit allocation**: Budget breakdown by type

The result is an explicit allocation for one decision, with required controls protected and omitted detail recoverable. Chapter 11 tests what may be reduced when the allocation is exceeded. Saving tokens is useful only if the remaining context still supports the decision the service is required to make.

Key Sources

- Context optimiser (the loader in the author's OpenClaw system; unpublished, measured with the Anthropic tokeniser in 2026)
- Anthropic, Claude Code documentation on context and compaction
- OpenClaw context management (author's system, Chapter 20)
- LLM context window specifications (OpenAI, Anthropic)
- Nelson F. Liu et al., "Lost in the Middle: How Language Models Use Long Contexts", *TACL* (2024; arXiv 2307.03172, 2023)
- Cheng-Ping Hsieh et al., "RULER: What's the Real Context Size of Your Long-Context Language Models?" (arXiv 2404.06654, 2024)

Chapter 11: Compression Strategies

Chapter 10 established the token budget. This chapter answers the question the budget raises: **when the budget is exceeded, what do you remove, and in what order?**

The four operations described here remove different kinds of information. Try deduplication and summarisation before truncation or eviction, so that repeated content goes before material the agent may still need. Several agent frameworks the author has reviewed start with truncation; this chapter explains the risks of that ordering.

The Compression Priority Order

The table orders the four strategies by what they remove and how much information they lose:

Priority	Strategy	What it removes	Token savings	Value lost
1	Deduplication	Content that appears more than once	Low	None, by construction
2	Summarisation	Detail from long sections that are not core	Medium	Some: the summary is a guess about what mattered
3	Truncation	The oldest non-constitutional content	High	Unknown: whatever was old

Priority	Strategy	What it removes	Token savings	Value lost
4	Drop	Whole inactive skills	Highest	All of the skill, until reloaded

The savings column is the author's experience with the OpenClaw context optimiser, not a benchmark. Deduplication typically recovers 5 to 15 per cent of a context assembled from several skills, because skills written by the same person repeat the same boilerplate; summarisation recovers 30 to 50 per cent of the sections it touches; truncation and dropping recover as much as you let them. Your numbers will differ, and the point is the ordering, not the percentages.

1. Deduplication: Remove Identical Content

Deduplication removes content that appears in more than one loaded skill: a shared example, a troubleshooting section pasted into three skills, an identical description of a tool. The second and later copies are replaced by a reference to the first, retaining the content while removing repeated copies.

```
function deduplicate(skills) {
  const seen = new Map();
  return skills.filter(skill => {
    const hash = hashContent(skill.content);
    if (seen.has(hash)) { skill.references = [seen.get(hash)]; return
false; }
    seen.set(hash, skill.name); return true;
  });
}
```

Whole-skill deduplication is straightforward. Section-level deduplication can recover repeated passages within otherwise distinct skills: hash each heading's body separately so that a shared "Common errors" section is kept once across five skills. Fuzzy deduplication merges similar rather than identical sections and can discard differences. Treat that operation as summarisation and assess what it removes.

2. Summarisation: Condense What Is Not Core

Summarisation replaces detail with a shorter account of it. Worked examples become one line about what the example shows; a twelve-step troubleshooting guide becomes two steps and a pointer to the full document; a list of fifteen edge cases becomes the three that matter and a note that there are more.

The OpenClaw optimiser uses a flag to identify sections that may be summarised. Core sections (the skill's purpose, its triggers, its invariants) are excluded. A non-core section over about 500 characters is summarised; that threshold was chosen by inspection rather than measured. A heuristic keeps the heading and first substantive line, then appends a pointer. This avoids the additional cost and latency of a model call for each summary.

Summarisation loses value, and the loss is not uniform. A summary of a worked example loses the example, which is what made it useful. The reason to accept that loss is that the full example is one load away: the skill body is still on disk, and the model can ask for it. Summarisation trades resident detail for retrievable detail, which is

the progressive disclosure principle of Chapter 10 applied inside a skill.

3. Truncation: Cut the Oldest Non-Constitutional Content

Truncation removes the oldest parts of the non-constitutional context when the first two strategies have not brought it under budget. The order of removal is: the oldest conversation turns first; then the least relevant of the files read during the session; then historical decision records, keeping the current state and dropping the account of how it was reached.

Truncation is where most systems start, and it is the strategy whose cost is hardest to see. What was old is not what was unimportant. A constraint stated in the second turn of a long session ("never write to the production database") is exactly the kind of thing that is old, short, and catastrophic to lose. The constitutional tier exists to protect what is known in advance to be like this; truncation is dangerous precisely for the things that were not known in advance. Two mitigations help: pin any turn that contains an instruction (a heuristic, but a cheap one), and never truncate below the last complete exchange, so the model always has the user's most recent words and its own last answer.

4. Drop: Evict Inactive Skills

Dropping is the last resort: a loaded skill is removed from the context entirely. The usual rule is least-recently-used, with the thresholds Chapter 12 describes: a skill unused for ten turns is a candidate, a

skill whose trigger relevance has fallen below a threshold is a candidate, and in an emergency anything inactive goes. Each drop recovers the whole cost of the skill, which in the author's skill set is one to two thousand tokens (Chapter 10's measurement), and loses the whole skill until it is reloaded.

Before dropping a skill, write its session state to memory: what it was doing and what it had learned. Without that state, reloading the skill ten turns later restores its instructions but loses its earlier work.

Building a Compression Pipeline

Figure 11.1 shows the decision structure the pipeline implements.

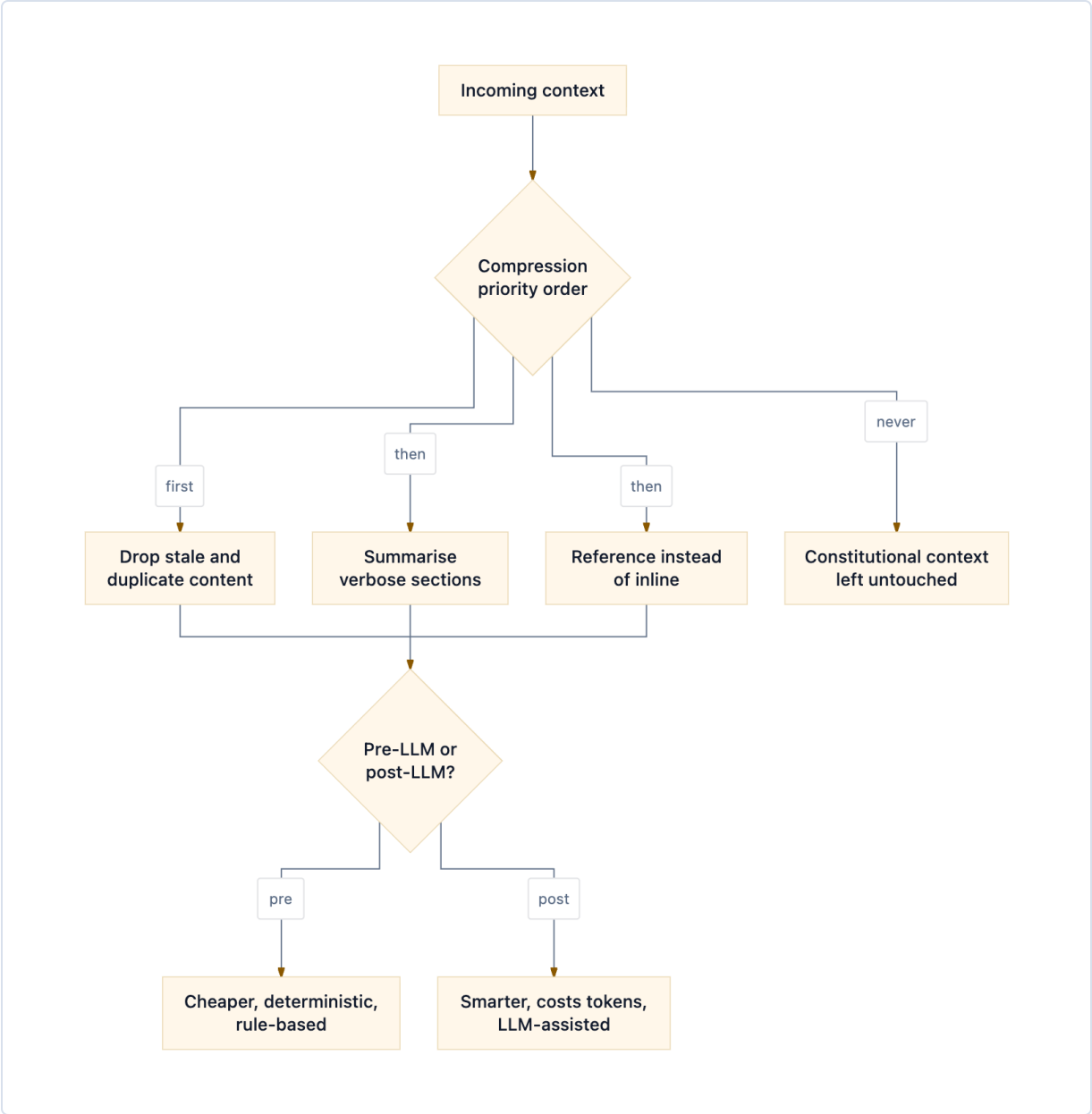


Figure 11.1: Compression priority order and the pre-LLM versus post-LLM decision.

The pipeline runs before each model call and applies the four strategies in order, stopping as soon as the context is under budget:

```
async function compressPipeline(context, budget) {
  const steps = [deduplicate, summarise, truncate,
dropInactiveSkills];
  for (const step of steps) {
```

```
    if (context.totalTokens <= budget) break;
    context = step(context, budget);
  }
  if (context.totalTokens > budget) throw new Error(`still over
budget: ${context.totalTokens} > ${budget}`);
  return context;
}
```

Two properties of this loop matter more than its code. First, it stops early: a context that deduplication alone brings under budget never sees truncation, which is the whole point of the ordering. Second, it fails loudly: if all four strategies cannot reach the budget, something is wrong with the budget or the constitutional tier, and the right response is an error rather than a silently oversized prompt that the model will truncate for you at the wrong end.

The pipeline should also emit a report: tokens before and after each step, which skills were dropped, and when. An illustrative run might show 15,000 tokens reduced to 14,000 by deduplication, 11,000 by summarisation, 10,500 by truncation and 10,000 by dropping one skill. The numbers are invented; the shape, most of the saving from the two safe strategies and a small remainder from the destructive ones, is what a healthy run looks like. A run where truncation does most of the work is a sign that the earlier strategies are misconfigured.

When to Compress: Before or After the Model

Compression can run before the model call, after it, or both, and the choice is a trade between predictability and quality.

	Compress before the call	Compress after the call
Budget	Always met	May be exceeded on this turn
Model input	Predictable size and shape	Maximum context for this answer
What is lost	Decided by heuristics before the model saw anything	Decided after seeing what the model used
Cost	Predictable	Variable
Failure mode	Loses something the model would have needed	Hits the hard window limit

Compressing before the call is budget discipline: the model always receives a context of known size, and the compression decisions are made by rules rather than by inspection. Its weakness is that the rules are guessing what the model will need. Compressing after the call gives the model the fullest context for the current answer and then compresses the exchange for the next turn, when it is known what was used. Its weakness is that it can overrun the window, and that costs vary.

Most systems the author has worked on use both: a pre-call pass to a budget slightly below the target (about 90 per cent of it, the author's setting rather than a measured optimum), and a post-call pass that compresses the latest exchange into the remaining room. The margin accommodates variation in the current turn; the post-call pass bounds the context retained for the next.

Field Evidence: Claude Code's Five-Layer Ladder

Figure 11.2 shows the ladder in the order the tiers fire.

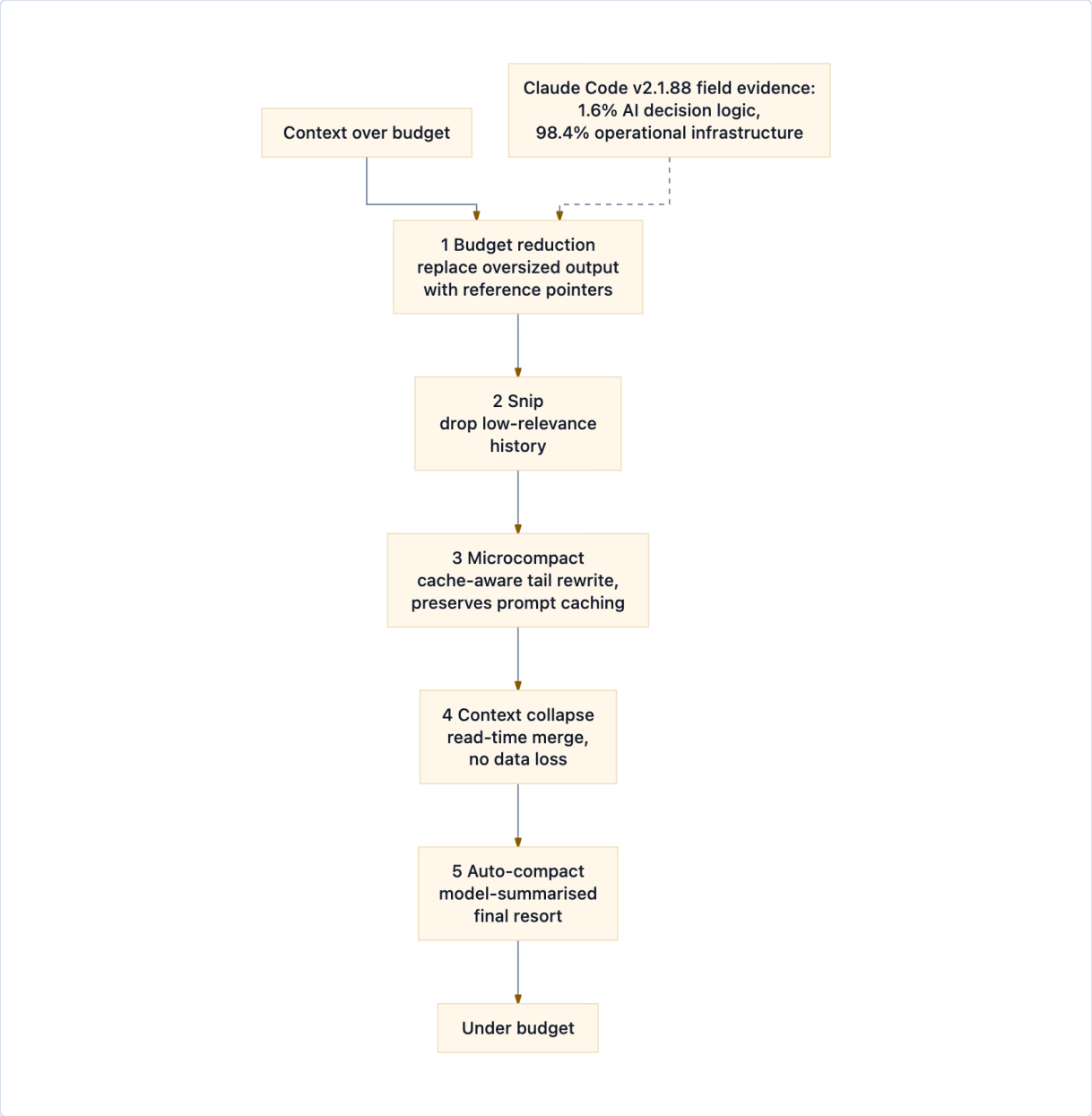


Figure 11.2: The five-tier compaction ladder documented in Claude Code, from cheap pointer substitution to model-summarised auto-compaction.

Liu et al. [@liu2026dive], reverse-engineering Claude Code v2.1.88 from its distributed bundle, document a graduated five-tier compaction pipeline that aligns closely with the priority order in this chapter: **budget reduction** (replace oversized raw tool output with reference pointers), **snip** (drop low-relevance history), **microcompact** (a cache-aware rewrite of the tail that preserves the economics of prompt caching), **context collapse** (a read-time projection that merges messages without losing data), and **auto-compact** (model-summarised compaction as the final resort). The same analysis finds that only 1.6 per cent of the Claude Code codebase is model decision logic; 98.4 per cent is operational infrastructure, of which the compaction pipeline is a significant share.

The model-driven step comes last, after cheaper operations that preserve the underlying data. This supports designing compression as a sequence with explicit loss properties, rather than adding truncation when the window fills.

When Compression Becomes the Failure Mode

A consolidation service can reduce yesterday's notes while adding another summary of the same unresolved work. Token reduction then conceals a different failure: the operator has more documents but no clearer account of what changed. The author's internal review of the consolidation pipeline (Experiment E5, Appendix I) describes repetitive daily consolidation, recorded lessons with little observed application and recurring issues without a closure mechanism. These

are reported diagnostic observations; they do not establish that a particular metric corrected the pipeline.

The stakeholder is the maintainer, whose concern is losing useful decision context among repeated reports. An illustrative requirement follows: **the consolidation service shall preserve outstanding commitments and their evidence while distinguishing new information from restatement**. The solution needs both content checks and a route for assigning and reviewing unresolved work.

Measure novelty against existing memory, then sample whether retained information supports a requirement or decision. A novel but irrelevant paragraph should not pass merely because it differs from yesterday's text. Measure lesson application through an observable changed action, with an identified lesson and opportunity to apply it. Measure issue closure against an explicit set of due issues and require closure evidence. An empty denominator is undefined, rather than zero or success.

```yaml
# Illustrative reporting fields, not measured results
consolidation_report:
  novelty_ratio: null
  novelty_method: "sentence similarity against retained memory"
  requirement_relevance_review: pending
  lesson_application:
    opportunities: 0
    observed_applications: 0
    rate: null
  issue_closure:
    due_issues: 0
    evidence_backed_closures: 0
    rate: null
```

The source PR also sketches values for a possible report, but it does not supply a measurement protocol for them. The nulls above therefore remain until a defined observation window and denominator are available. A novelty gate can merge repeated information, while application and closure checks determine whether a retained lesson changed subsequent work.

The report becomes useful when its findings change the workflow: merge a restatement into the existing record, assign an overdue commitment, or retain a decision whose evidence would otherwise be lost. Compare preserved information and subsequent use before calling the intervention an improvement. Figure 11.3 connects these checks to the compression sequence.

Summary: Systematic Compression

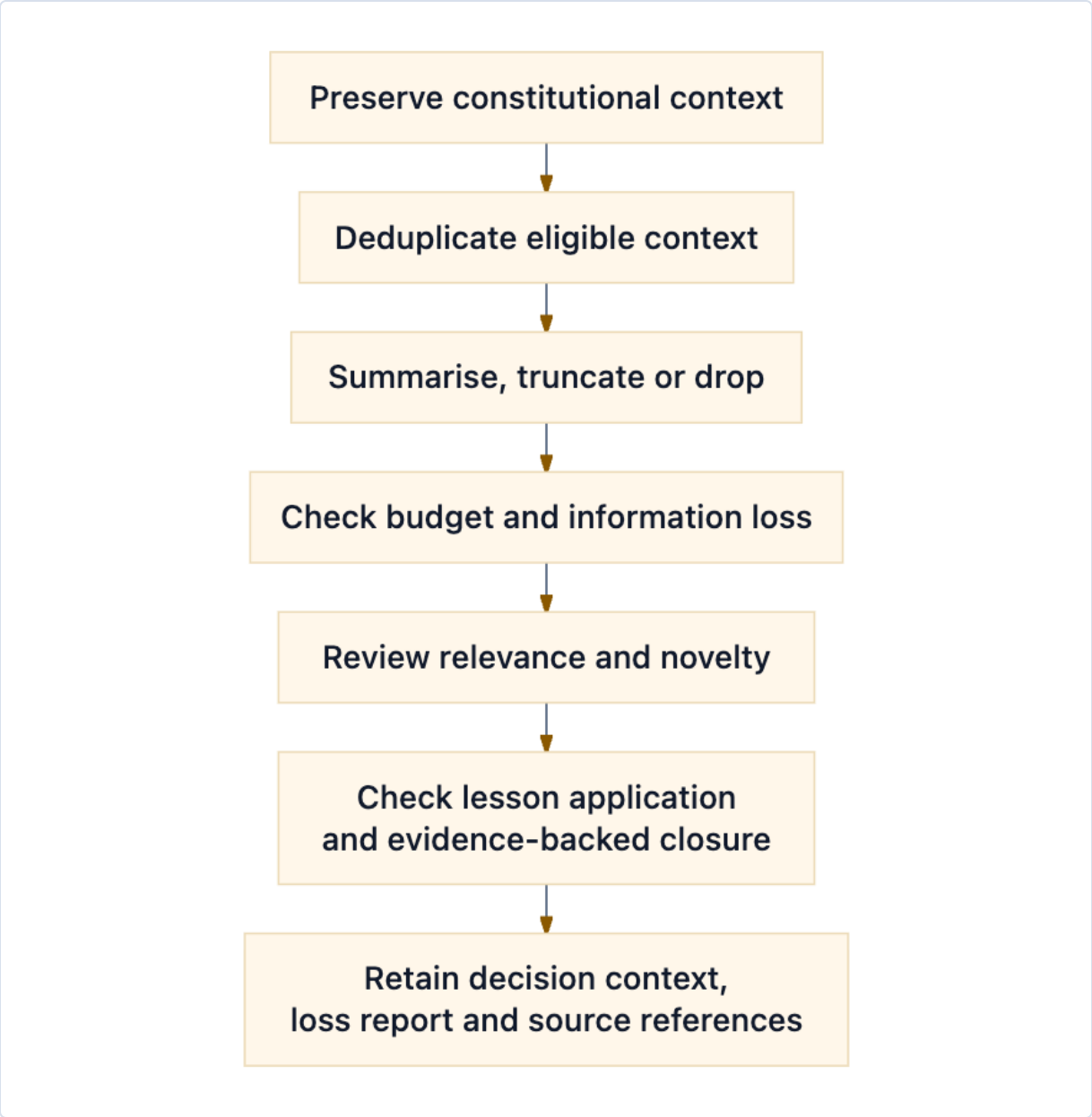


Figure 11.3: Chapter 11 recap: Compression preserves useful decision context through budget, information-loss, relevance and follow-through checks.

Compression strategies must be systematic, not ad hoc:

1. **Priority order**: deduplicate, then summarise, then truncate, then drop

2. **Pipeline**: apply the strategies in sequence and stop as soon as the budget is met; fail loudly if it cannot be
3. **Constitutional context is never compressed**: the loader keeps it out of the pipeline, as Chapter 10 describes
4. **Report**: log what was removed at each step, so that a run dominated by truncation is visible
5. **Before and after**: a pre-call pass for discipline, a post-call pass for the next turn

For the maintenance proposal, a shorter record must retain the unknown-owner exception, shared-quota limit and source identity whenever they affect the decision. The reduction report and recoverable references pass to the next step. Chapter 12 packages capabilities for selective loading and tests whether the request activates the right ones.

Consolidation also needs checks for relevant new information, observed lesson application and evidence-backed issue closure.

Source for the added operational case: Experiment E5 (Appendix I), the author's internal review of the consolidation pipeline; the observations have not been independently reproduced for this edition.

Key Sources

- OpenClaw context optimiser (author's system); the savings percentages in this chapter are observations from it, not

benchmarks. Its settings and measurements: the 500-character section-length threshold, chosen by inspection; the ten-turn inactivity rule; the 90 per cent pre-call budget; and the one-to-two-thousand-token skill size

- Liu et al., *Dive into Claude Code: The Design Space of Today's and Future AI Agent Systems*, arXiv 2604.14228 (2026): the five-tier compaction ladder and the 1.6 per cent figure
- Anthropic, Claude Code documentation on context compaction and prompt caching
- Chapters 10 and 12 of this book for the budget the pipeline serves and the skills it operates on

Chapter 12: Skill Architecture for AI Agents

Chapter 11 described how to compress context when a budget is exceeded. Skills provide much of that context. A **skill** packages a capability as a description of what the agent can do, when it should do it and how. Design skills for loading and unloading: keep a small catalogue resident, activate bodies through triggers, cap active skills and define eviction rules. Compression alone cannot compensate for loading more skill content than the budget allows.

Several frameworks load every skill at startup and rely on a large window; others retrieve skills by embedding similarity. The comparisons below identify when trigger-based loading is useful and when a larger catalogue may warrant embedding retrieval.

The Catalogue: Small Enough to Always Be Loaded

The catalogue is the minimal representation of every skill, loaded at startup and resident for the whole session. It exists to make one decision, whether a skill should be activated, and it should contain nothing that is not needed for that decision.

```
skills:
  - name: web-search
```

```
    triggers: ["search", "web", "find", "lookup", "google"]
    tier: standard
    estimated_tokens: 800
  - name: terraphim-memory
    triggers: ["memory", "recall", "remember", "kg", "knowledge"]
    tier: core
    estimated_tokens: 1200
```

For the author's OpenClaw skill set of about twenty skills, the catalogue costs 200 to 300 tokens, measured with the model's tokeniser, against roughly 16,000 tokens for the full content of the same skills. The ratio, not the absolute number, is the design target: a catalogue should cost about one or two per cent of the content it indexes. If it costs more, it is carrying description that belongs in the skill body.

The catalogue carries four things per skill: a name, the trigger words, a tier, and an estimated token cost. The estimate is what lets the loader reason about budget before loading anything; a catalogue without it forces the loader to load a skill to find out whether it can afford to.

Activation by Trigger

A skill is activated when the user's request matches enough of its triggers. The matching is deliberately crude: lower-case the query, count the triggers that appear in it, divide by the number of triggers, and activate if the fraction clears a threshold.

```
function triggerScore(skill, query) {
  const q = query.toLowerCase();
  const hits = skill.triggers.filter(t =>
q.includes(t.toLowerCase())).length;
  return hits / skill.triggers.length;
}
```

The threshold is a tuning knob with two failure modes. Set it high and skills fail to activate when the user phrases a request in words that are not in the trigger list. Set it low and skills activate on incidental matches (“pr” inside “prepare”, as Chapter 20 notes). The OpenClaw setting is a threshold of 0.7 with a floor of one matching trigger; the author arrived at it by watching a month of sessions, not by measurement, and a different skill set would need a different value.

Why substring triggers rather than embedding similarity? Embedding-based activation handles paraphrase and would solve the first failure mode. It costs an embedding call per turn, it is not explainable (Chapter 6), and it introduces a second probabilistic component into a system whose point is to keep the probabilistic component to the model itself. Trigger matching is deterministic, free, and wrong in ways that are easy to see and easy to fix by adding a trigger. For a skill set that one team maintains, that trade favours triggers. For a marketplace of thousands of third-party skills with no shared vocabulary, it probably does not, and embedding retrieval over the catalogue, with trigger matching as the deterministic fast path, would be the design.

The Cap on Active Skills

Active skills consume tokens for as long as they are resident. The design caps the number of simultaneously active skills, and the cap the author uses is five.

Active skills	Resident tokens at ~2,000 each	Effect in the author's sessions
3	6,000	Comfortable: half the 12,000 budget free for memory and work
5	10,000	Workable: room for the constitutional tier and little else
7	14,000	Over budget: compression fires on every turn
10	20,000	Unusable without dropping most of what was loaded

The table is arithmetic from Chapter 10's budget, not a quality measurement: the middle column is the token cost, and the right column is what happens to the rest of the budget. The author has not measured answer quality against skill count and does not claim a curve. The claim is narrower: with a 12,000-token working budget and skills of one to two thousand tokens, five is the most that fit alongside the constitutional tier and a useful amount of memory. A larger budget could raise the cap; a smaller one should lower it.

When more skills trigger than the cap allows, the loader must choose. The order it uses is tier first (core skills over standard, standard over optional), then trigger score, then recency of use. Tier first encodes a judgement about which failures are expensive: an

unloaded memory skill can cost a session its continuity, an unloaded weather skill costs one wrong answer about the weather.

Eviction

A skill that has been loaded does not stay loaded. Two rules evict it: non-use, and budget pressure.

The non-use rule in the author's configuration is ten turns. A skill that has not been invoked in ten consecutive turns is removed from the context and returns to its catalogue entry. Ten is a judgement, chosen so that a skill survives a short digression but not a change of topic; it is not a measured optimum. The budget rule is the LRU eviction Chapter 11 describes: when the compression pipeline reaches its last step, the least recently used skills go first.

Figure 12.1 shows the state machine the rules operate on.

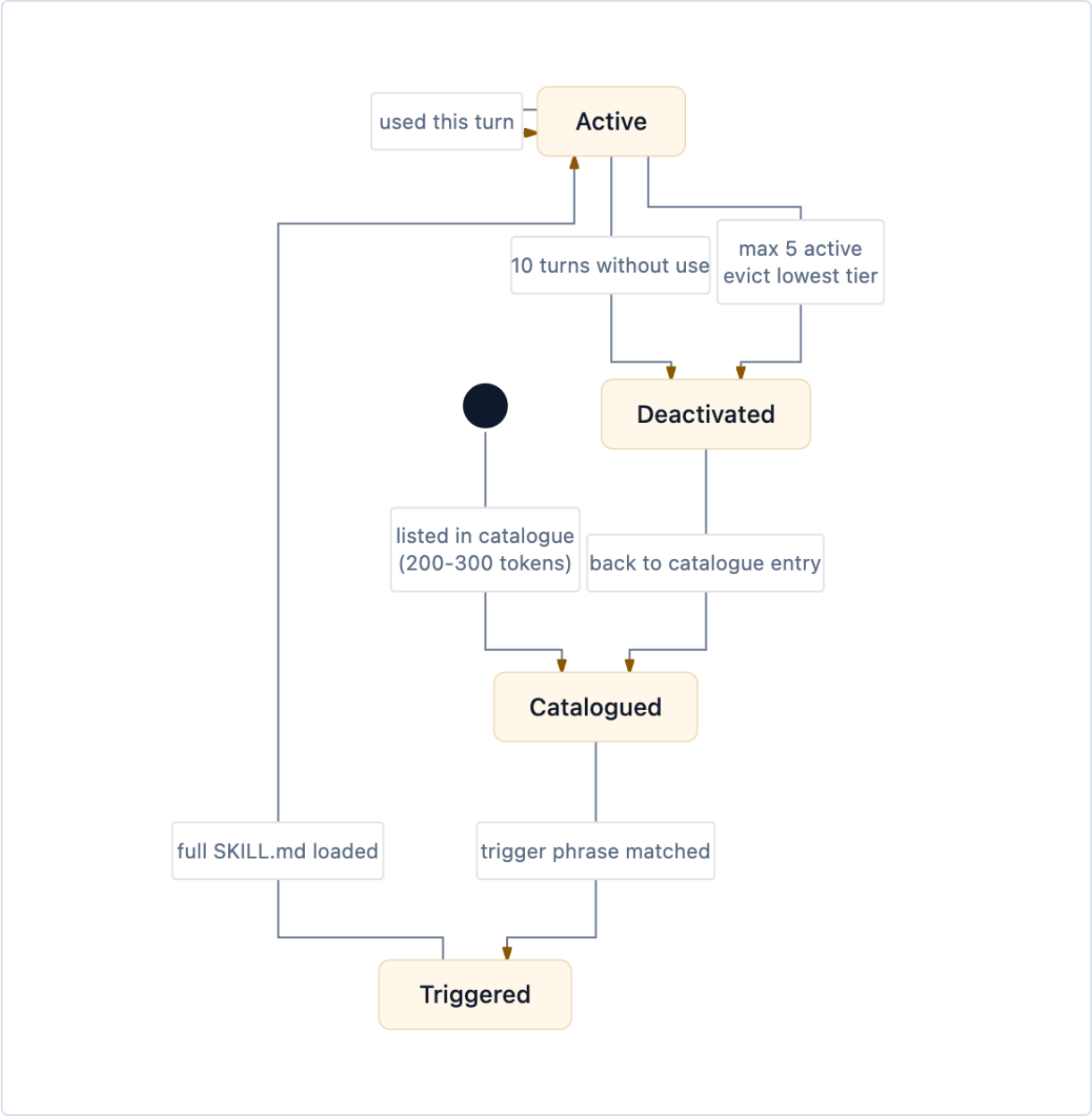


Figure 12.1: Skill lifecycle states: catalogued, triggered, active, deactivated, and back to the catalogue.

Eviction has one obligation: save the skill's session state before removing it. A skill that was tracking a multi-step task, was evicted during a digression, and is reloaded with no memory of the task has been reset, not paused. The loader writes the state to session memory on eviction and the skill reads it on activation.

Where This Design Fails

Trigger-based loading has three recurring failure modes in the author's system.

Trigger collisions. Two skills that share a trigger ("search" in both a web-search skill and a database-query skill) both activate, both consume budget, and one of them was not wanted. The fix is a vocabulary discipline over triggers: each trigger word belongs to one skill, and the catalogue is validated for overlaps when it is built. This is the thesaurus discipline of Chapter 8 applied to a very small vocabulary.

False activation. Substring matching activates skills on incidental text. It costs the budget of the skill, not a wrong action, and the cap limits the damage. A system where activation had side effects would need word-boundary matching at minimum and the thesaurus-backed matching of Chapter 7 in practice.

Thrashing. A skill evicted for non-use is needed two turns later, reloaded, evicted again. Each cycle costs a load and a state save. The symptom is a skill that appears repeatedly in the compression report; the remedy is to raise its tier or lengthen the non-use window for that skill, which the metadata below allows per skill.

Taxonomy: Names, Triggers, Metadata

A skill catalogue is a small controlled vocabulary, and it benefits from the same discipline as a large one.

Names are specific and consistent: `web-search`, `file-editor`, `code-runner`, not `web`, `editor`, `run`. A name that is ambiguous in the catalogue is ambiguous in the agent's own reasoning about what it can do.

Triggers are specific (the full word, with abbreviations listed explicitly if wanted), complete (common synonyms included), and exclusive (each trigger owned by one skill). The exclusivity rule is the one most often broken and the one the catalogue validator should enforce.

Metadata carries what the loader needs and nothing else:

```
skill:
  name: web-search
  version: 1.2.0
  tier: standard            # core | standard | optional
  triggers: [search, web, find, lookup]
  estimated_tokens: 800
  dependencies: []
  auto_deactivate_turns: 10
  compression_priority: 2   # lower is dropped first
```

The two per-skill overrides, the deactivation window and the compression priority, are how the thrashing and tier problems above are handled without changing the global rules.

Routing as Context Selection: Measuring the Decision Layer

A router decides which capabilities enter context. A capability question such as “can you parse JSON?” can match a command trigger even though the user has not requested execution. The operator’s concern is inappropriate capability selection. An illustrative requirement is: **the routing service shall reject non-action requests and preserve every requested operation in a compound request**. Its acceptance cases need negatives, capability questions and compounds alongside ordinary commands.

The author’s exp-002 routing experiment (E3, Appendix I) compares a single-shot baseline with a gated router on a reported 140-case held-out set. The figures below are author-reported: the underlying experiment directory was unavailable in the editorial checkout, so this book does not claim independent reproduction.

Metric	Single-shot baseline	Gated router
Overall accuracy	32.9%	99.3%
False positives on negative cases	50/50	0/50
Compound exact-set correctness	0/25	25/25
Mean Brier score, as reported	0.2569	0.0158
Positive-class accuracy	51.1%	98.9%

The baseline reportedly routed the JSON capability question at 0.9 confidence. An unconditional fallback can therefore create a tool-selection error before the answering model sees the task. A single-

skill response schema introduces another limitation: “fetch the URL, parse JSON and extract text” cannot be represented completely by one selected skill. An explicit operation set makes omission measurable. Exact-set correctness and per-operation recall answer different questions; report both when partial completion matters.

A fallback is also a prediction. A fixed confidence value may describe a software default rather than an estimated probability of a correct route. Before interpreting Brier, identify the scored event, confidence construction and treatment of multi-operation outputs. The reported score alone cannot establish that those quantities have the same meaning across implementations.

The account identifies one disputed label for a composed-skill input. Preserve that ambiguity and its adjudication policy rather than tuning the router to an unexplained exception. Test paraphrases and unfamiliar domains separately: Chapter 25 returns to the reported shift results. Chapter 16 explains how a typed selection becomes an input to runtime policy. Figure 12.2 shows the reject, single-operation and compound branches before budgeted activation.

Make Rejections Inspectable

The same experiment (E3, Appendix I) describes a yes/no gate followed by choice and compound tags. Its rejection reasons include question form, meta-vocabulary, chat openers, control words and terms outside the catalogue. Recording the reason lets a maintainer distinguish a missing capability from a misunderstood request. These are local signal categories, not universal intent rules: a question can

contain a genuine request, and an unfamiliar term can name a valid task.

The account says the original router had been checked against one happy-path example. The lesson is to design the acceptance set around the operator’s concern before trusting the router. Add capability questions, metaphors, unsupported operations and compounds; inspect false rejections as well as false activations. It also reports a flat 0.5 confidence on baseline fallback routes. Preserve that distribution when analysing errors instead of reporting only a mean confidence, which can conceal a software default used for unrelated cases.

Summary: Skills as Context Modules

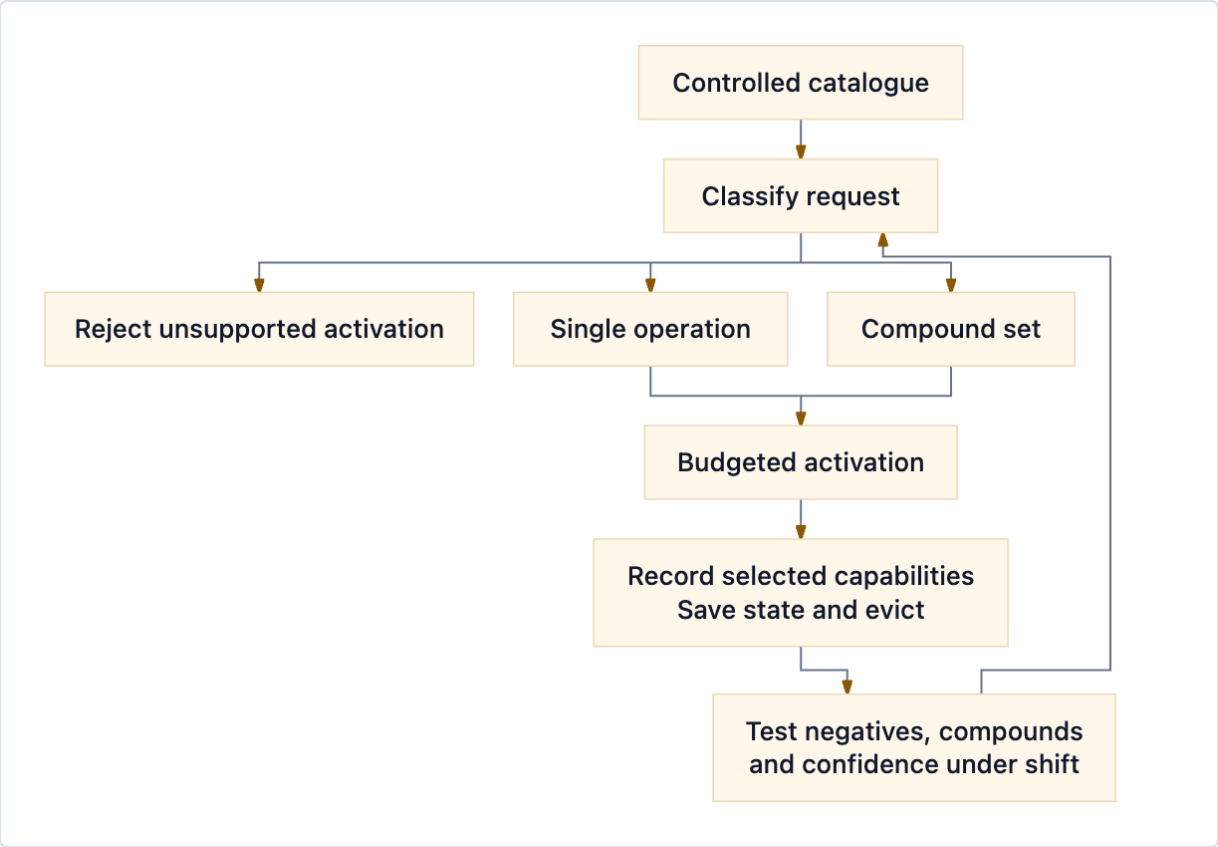


Figure 12.2: Chapter 12 recap: Routing rejects non-action requests, preserves compound operations and activates capabilities within a measured budget.

Skill architecture is what makes progressive disclosure possible:

1. **A catalogue** small enough to always be resident, carrying only what the activation decision needs
2. **Trigger-based activation**, deterministic and cheap, with its failure modes known and bounded
3. **A cap on active skills** derived from the budget, five in the author's configuration, with tier-first selection when the cap binds
4. **Eviction** after non-use and under budget pressure, with session state saved first

5. **A taxonomy** that treats triggers as a controlled vocabulary

These rules bound the material sent to Chapter 11's compression pipeline and make its token costs visible against Chapter 10's budget.

Evaluate negative requests and compound completeness before activation; confidence defaults require explicit probability semantics.

The selected capability set becomes part of the proposal's context record. A request to inspect a failed job and propose a repair needs both operations; selecting an execution skill does not authorise execution. Chapter 13 supplies persistent project rules, while Chapters 15 and 16 assess the selection mechanism and constrain how its result may be used.

Source for the added operational case: Experiment E3 (Appendix I), the exp-002 skill-router evaluation on 140 held-out cases, with its out-of-distribution follow-up (E4, 100 cases across eight domains); the figures are author-reported and have not been independently reproduced for this edition.

Key Sources

- OpenClaw skill loader and catalogue (author's system); the token figures and thresholds in this chapter are its measured costs and its chosen settings
- Anthropic, Claude Code documentation on skills and the SKILL.md format

- Chapters 6, 7, 10, 11 and 20 of this book

Chapter 13: CLAUDE.md, AGENTS.md and Project-Wide Rules

Chapter 12 described skills: modular capability loaded on demand. **Project-level context files** supply conventions that apply across tasks, whether or not a skill is active. This chapter covers `AGENTS.md` as a shared Markdown convention and `CLAUDE.md` as Claude Code's instruction file. The design applies across assistants, while each client determines which files enter its context. Writing conventions in a file gives the team something it can version, review and test, and reduces the conventions an agent must infer afresh each session.

Loaded rules occupy the context budget (Chapter 10) and can become stale or too long to review. They need a budget, an owner and updates when the toolchain changes.

The maintenance assistant now has a selected capability and a bounded context packet. It still needs the project's current rule for unknown ownership, the requirement for evidence of restoration and the approved execution limits. A task file identifies the candidate and use conditions; a project rules file points to the governing baseline. Neither file grants the assistant authority to change that baseline.

The Problem

An agent starting work on an unfamiliar codebase makes assumptions: how tests are written, what style applies, where documentation lives, whether it should ask before implementing. Across the author's Claude Code projects, recurring mistakes included missing tests, documentation in the wrong style and features built to an inferred requirement. Switching assistants adds another source of variation: the same repository can expose different instructions to different clients.

Put shared conventions in version control, then verify that each supported client loads the applicable revision before work begins.

The Solution: a Rules File

Figure 13.1 separates shared project guidance from client-specific loading and independent checks.

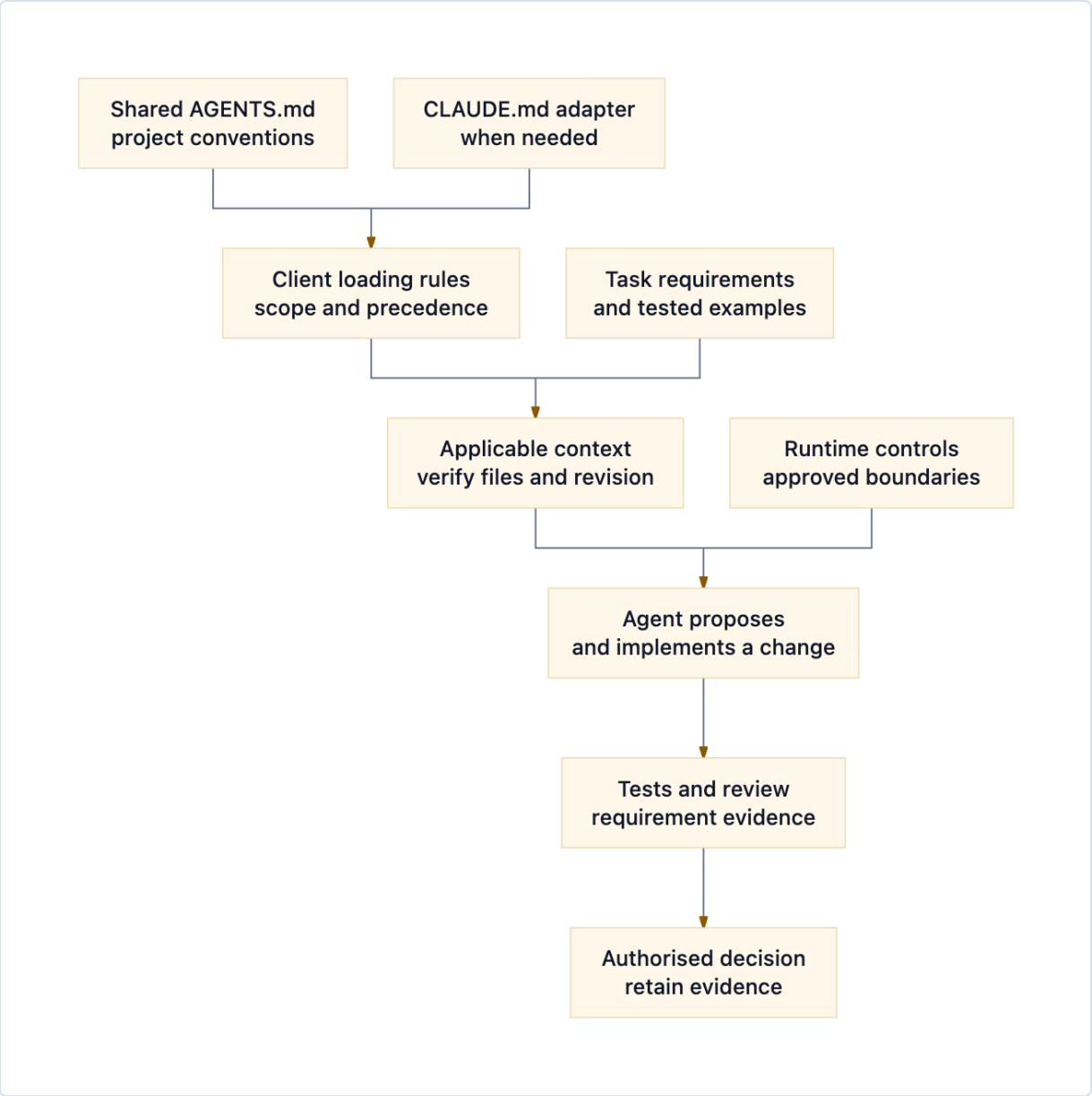


Figure 13.1: Shared project instructions reach each assistant through its loading rules; tests and runtime controls check the resulting work.

A rules file is short, declarative and specific. The following skeleton can be used in either `AGENTS.md` or `CLAUDE.md`:

```
# Project Context
## Project Overview     -- name, type, what it does, in three lines
```

```
## Testing Requirements -- framework, location, naming, the command
that must pass, coverage floor
## Code Style           -- the three or four rules that matter and the
formatter that enforces the rest
## Documentation        -- what must exist, where, and the ADR
convention
## Workflow             -- type-check and test commands, commit
convention, review rule
## Authority and Evidence -- governing requirements, operating limits,
evidence to retain
```

Each section should contain checkable rules. “Use Vitest, in `__tests__/` beside the source, named `{module}.test.ts`, run with `npm test` before any commit, 80 per cent coverage on new code” specifies a framework, location, command and threshold. “Write good tests” leaves those choices unresolved.

Keep the commands executable and the ownership clear. The style section names the formatter and leaves formatting to it, because a rules file that restates Prettier’s configuration in prose is a second source of truth that will drift from the first. And the workflow section gives commands, not descriptions, because an agent can run `npm run typecheck` and cannot run “make sure the types are right”.

Writing Checkable Rules

The principle behind every section is explicitness. Compare the two forms of the same rule:

Vague rule	Checkable rule

Vague rule	Checkable rule
Write tests for new code.	For this TypeScript package, put unit tests in `__tests__/` using Vitest's `describe` / `it` syntax. Run `npm test` before commit; new code requires 80 per cent coverage. Run service integration tests against the documented local test environment.

The detailed rule fixes the framework, location, coverage target and integration environment. These are example project conventions; select commands and thresholds that match the actual repository. Such detail adds length, so review the file as the toolchain evolves. Keep each rule concise and remove instructions already enforced by the formatter, linter or test runner.

Task Context: INITIAL.md and the PRP

A root rules file carries shared conventions; a scoped file adds conventions for its part of the repository. What is true of one task goes in a task file, created on the feature branch and read by the agent as its specification. The author's convention is a file named `INITIAL.md` with five sections: the problem in two sentences, numbered requirements, technical constraints (which database, which existing modules to reuse, which patterns to follow), what is out of scope, and success criteria as a checklist the agent can tick.

`INITIAL.md` is an author convention, not a filename every assistant discovers automatically. Pass it with the task or explicitly ask the

client to read it.

The same shape, with a user story and success metrics added, is the **Product Requirements Prompt** (PRP) pattern that circulates in the context-engineering community. Write the PRP before the code; have the agent read it and resolve questions; then implement and verify against its acceptance criteria. Retain the agreed revision so the reviewer can compare the result with the requirements set before implementation, rather than relying on the agent's account of its work.

Examples as the Source of Truth

An `examples/` directory can make conventions concrete. Include a canonical service, controller, test, REST endpoint, component and database query in the house style, and point to them from the rules file. A reference such as "follow `examples/auth/auth-service.ts`" shows the intended layering, error handling and naming in working code.

Examples go stale as fast as rules do. The author's rule is that an example file is tested in CI like any other code, so that a change to the house style that breaks the example breaks the build.

AGENTS.md as Shared Project Guidance

`AGENTS.md` is an open convention for instructions written in ordinary Markdown, with no required headings. Its published guidance recommends a root file for project conventions and nested files for

local guidance. Treat it as a useful common entry point, while checking the loading behaviour of each supported assistant. [AGENTS.md format guidance].

For a team using several assistants, keep shared instructions in `AGENTS.md`. Put client-specific settings in their supported locations and keep task requirements in their own versioned record. This lets reviewers distinguish a change to the service's requirements from a change to how an assistant finds them.

Consider this illustrative layout for the maintenance service:

```
AGENTS.md                          # Shared conventions and governing
references
CLAUDE.md                          # Claude-specific adapter, when
required
services/maintenance/AGENTS.md      # Local test commands and protected
scope
requirements/maintenance.md         # Approved requirements and
revision owner
tasks/repair-scheduled-job.md        # Candidate, intended use and
acceptance checks
examples/maintenance/                # Tested implementation examples
```

A short shared file might say:

```
# Maintenance repository instructions

## Governing baseline
- Read requirements/maintenance.md and the supplied task record before
editing.
- Preserve requirement identifiers in the change description and test
evidence.
- Propose requirement changes to the named owner; do not approve them
yourself.
```

```
## Verification
- Run the package's documented tests and retain commands, results and
commit ID.
- Report a blocked or failed check explicitly; do not describe it as
passed.

## Operating limits
- Unknown ownership requires review by the responsible service owner.
- Preserve the approved per-run and cumulative limits in the execution
service.
- A passing test does not authorise deployment or enlarge execution
scope.
```

These instructions connect stakeholder concerns to the requirements and services that address them. In the continuing example, `R-USE-01` requires evidence of successful completion and delivery before reporting restoration; `R-BOUND-01` requires established scope and resource limits before action. The rules file points to those requirements and tells the agent which evidence to retain. The execution service still needs controls that withhold actions outside its approved boundary.

A CLAUDE.md Adapter

When Claude Code needs a `CLAUDE.md` entry point, keep it small:

```
@AGENTS.md

## Claude Code workflow
- Present the planned files and verification commands before making
changes.
```

The `@AGENTS.md` line is Claude Code's import syntax; other clients need not interpret it. Anthropic's current documentation also describes direct `AGENTS.md` loading from version 2.1.277, subject to session support and settings. By default, a project or ancestor `CLAUDE.md` or `CLAUDE.local.md` changes that selection. Check the installed client before removing an adapter. Claude Code memory documentation.

An import avoids maintaining two copies of the shared text. It also avoids relying on symlinks across operating systems. Confirm that the imported file actually appears in the session's loaded context; writing "read AGENTS.md" in prose is a request for a later read, rather than the same loading mechanism.

Discovery, Scope and Precedence

The filename alone does not establish which instructions an agent receives. Working directory, client configuration and supported features can change the result. The following examples were checked against official documentation on 9 October 2026; they describe loading, not guaranteed compliance.

Client	Project entry point	Discovery and scope to check

Client	Project entry point	Discovery and scope to check
OpenAI Codex	`AGENTS.md`	Builds a chain from project root to working directory. At each directory, the override is considered before the ordinary file; nearer guidance appears later.
Claude Code	`CLAUDE.md`; supported sessions can also load `AGENTS.md`	Loads ancestor instructions and discovers local guidance during work. Existing Claude instruction files and settings affect direct `AGENTS.md` selection.
Cursor	`AGENTS.md`	Supports root and nested Markdown guidance. Structured rules have activation settings such as always included, path scoped or manually invoked.
Cascade (Devin Desktop)	`AGENTS.md`	Root guidance is always included; subdirectory guidance applies when files there are read or edited.
GitHub Copilot agents	`AGENTS.md`	Agent instructions and repository or path-specific instructions have distinct support. Check the particular Copilot feature being used.

Codex takes at most one instruction file per directory and has a configurable combined project-document budget, 32 KiB by default. Its `AGENTS.override.md` convention should therefore be treated as client-specific. Starting a run in a service directory and starting at the repository root can produce different instruction chains. Codex instruction discovery.

Cursor combines nested `AGENTS.md` guidance with parent guidance, giving more specific instructions precedence. Its structured `.mdc` files in `.cursor/rules/` offer additional activation controls. A file reference in a rule is read when needed, rather than automatically inlined. Cursor rules documentation.

Cascade derives scope from an `AGENTS.md` file's location. The Windsurf documentation now redirects to Devin Desktop, which documents `.devin/rules/` and legacy `.windsurf/rules/` for frontmatter-based rules. Cascade instruction documentation. GitHub documents nearest-file precedence for agent instructions. Repository guidance also uses `copilot-instructions.md` under `.github/`; path-specific files use the suffix `.instructions.md` under `.github/instructions/`. Support varies between Copilot features, so test the feature the team actually uses. GitHub repository instructions.

Instruction Precedence and Organisational Authority

A client's precedence rules resolve which text it presents or prioritises. Organisational authority determines who may change a requirement, approve a release or accept a risk. A nested file that

loads later cannot establish that its author has authority to weaken a service-wide limit.

For the maintenance example, a local instruction to “retry until the job succeeds” conflicts with the approved cumulative resource boundary. The team should correct the conflict and make the execution service enforce the boundary. Asking the model to choose between inconsistent instructions leaves the decision dependent on its interpretation.

Keep local guidance additive where possible: name the service’s test command, identify its examples and explain its exceptions. An exception that changes a requirement needs the applicable owner’s approval and a new baseline. A generated lesson or an agent’s personal memory may suggest such a change; it does not approve it.

Testing the Instructions Across Clients

Treat a rules-file change as a change to the context supplied to a service. Review the shared file, adapters and affected local files together. A copied adapter needs a generation or comparison check; an import needs a loading check. Merely opening the shared file in an editor proves neither.

Use a small disposable repository or an isolated branch with harmless test tasks. For each supported client, record its version, configuration, launch directory and applicable instruction files. Ask it to identify the required test command and the governing

requirement revision, then observe its actual choices. A summary of its instructions is useful diagnostic evidence, but cannot establish that it will follow them during execution.

Probe	Expected evidence
Start at the repository root	The shared convention is present and the documented root check is selected.
Work in the maintenance package	Its local test command is selected; unrelated package guidance is excluded where the client supports that scope.
Supply a task with an unknown owner	The candidate is held for the responsible owner's review.
Place contradictory guidance in a local test fixture	The contradiction is exposed for review; execution controls still withhold actions beyond the approved boundary.
Change a command in the shared file	A fresh session uses the revised command without a stale copied adapter.

These are proposed checks, not results from a cross-client experiment. Run them when adopting a client, changing the instruction layout or upgrading a version that affects discovery. Retain failures as fixtures for the judge harness in Chapter 15, and use runtime policy from Chapter 16 for actions requiring enforced limits.

Where Rules Files Fail

Rules files need maintenance in four areas.

Bloat. Every incident adds a rule, no rule is removed, and the file grows past the point where the agent reliably attends to all of it. The symptom is an agent that follows the first screen of rules and ignores the rest. The remedy is the deletion discipline above, and a budget: the author caps a project rules file at about 150 lines and moves anything longer into a skill (Chapter 12) that loads on demand.

Staleness. The rule says Vitest and the project moved to another runner. The agent follows the rule, and the rule is now the bug. The remedy is to review the file in the same pull request as any toolchain change, and to prefer rules that name a command (which fails loudly when it is wrong) over rules that name a tool (which fails silently).

Missing or conflicting instructions. A file exists but the client never loads it, or a stale adapter contradicts the shared baseline. Check the effective instruction chain after layout changes and client upgrades. Keep mandatory context small enough to fit the client's discovery budget.

Rules without enforcement. "Request review before merging" can guide the agent, but the text cannot prevent a merge. Required reviewers and branch protection can enforce the release rule; permissions and runtime policy can restrict actions. Hooks (Chapter 19) help only where the relevant execution path invokes them and honours refusal. State the convention in the file and identify the control that enforces it.

Summary

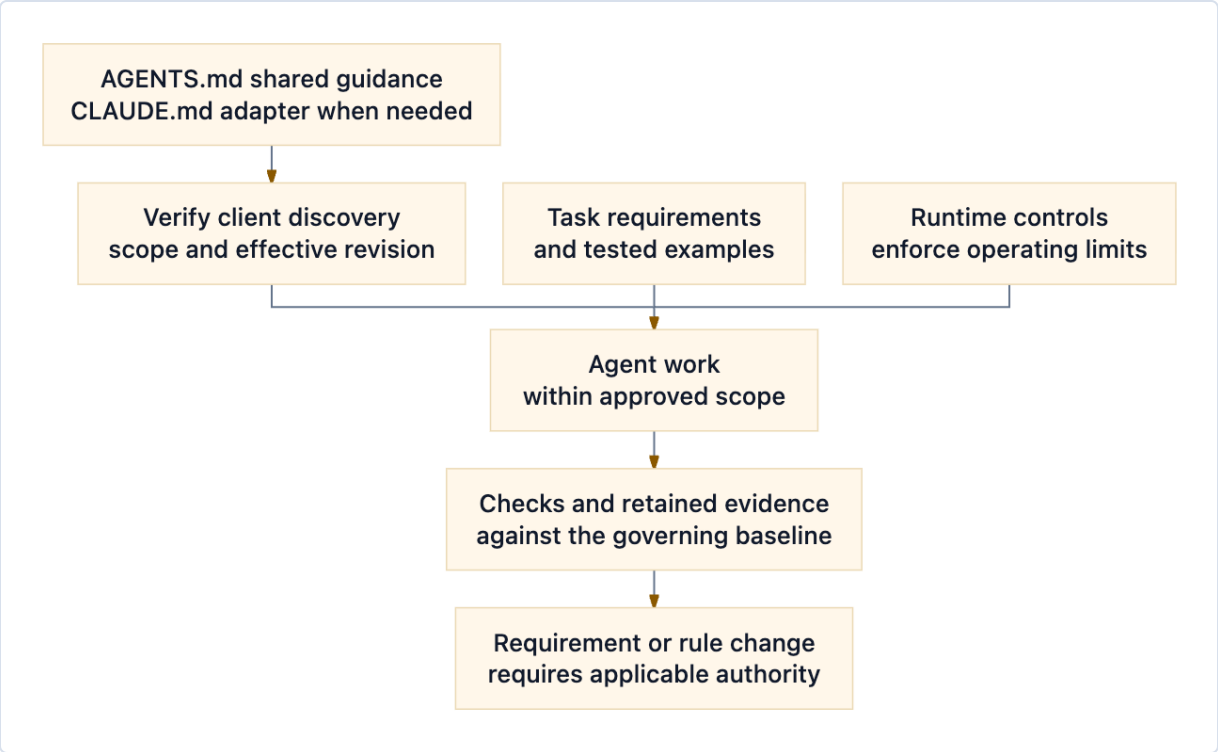


Figure 13.2: Chapter 13 recap: Shared AGENTS.md guidance and CLAUDE.md adapters supply scoped context; requirements, checks and runtime controls retain separate responsibilities.

Project-level context files complement skills:

1. **A rules file** states the testing, style, documentation and workflow conventions of the project, with governing references and explicit operating limits
2. **A task file** (INITIAL.md or a PRP) states the requirements, constraints and acceptance criteria of one piece of work, written before the work
3. **An examples directory** shows the house style in code the agent can copy, tested in CI so it cannot go stale silently
4. **Shared guidance is portable** across assistants; discovery, scope, imports and precedence require client-specific checks

5. **Rules files need review** for bloat, staleness and conflicts; tests, branch protection and runtime controls provide enforcement

The rules file belongs to Chapter 10's persistent project context. Keep it concise enough to review and current enough to guide the next task. Record which revision each client received; a common filename does not prove equivalent behaviour.

The handoff to Chapter 14 is a versioned set of requirements and examples that checks can actually use. The maintenance candidate may now be understandable and relevant, but its effect on protected work remains untested. Chapters 14–17 establish criteria, assessment, permission and retained evidence as separate responsibilities.

Key Sources

- AGENTS.md format guidance, accessed 9 October 2026
- Anthropic, Claude Code memory and instruction loading, accessed 9 October 2026
- OpenAI, Codex custom instructions with AGENTS.md, accessed 9 October 2026
- Cursor, Rules and AGENTS.md, accessed 9 October 2026
- Devin Desktop, Cascade AGENTS.md instructions, accessed 9 October 2026; the Windsurf documentation URL redirects here
- GitHub, Repository custom instructions for Copilot, accessed 9 October 2026

- The PRP (Product Requirements Prompt) pattern as circulated in the context-engineering community, 2025
- The author’s project rules files across several Claude Code repositories, for the line budget and the failure modes
- Chapters 10, 12 and 19 of this book

Chapter 14: Validation and Quality Gates

Chapter 13 described how project rules define expected behaviour. This chapter describes the checks that detect departures from those rules, incorrect claims and context drift.

Validation runs at three points: **pre-LLM validation** checks syntax, consistency and grounding before the model runs; **post-LLM validation** checks generated claims through grounding and judge evaluation; and **continuous monitoring** uses drift detection and impact analysis to assess changes over time. This chapter describes each check.

For the maintenance case, criteria must cover the failed job and protected behaviour, including unknown ownership and shared-resource limits. Link each check to the requirement and intended-use condition it supports. Passing a candidate through an evaluator establishes evidence about those cases; the separate policy boundary in Chapter 16 determines whether that evidence permits action.

KLS (Knowledge Layer Schema) Validation Protocol

Figure 14.1 shows the five dimensions and the workflow around them.

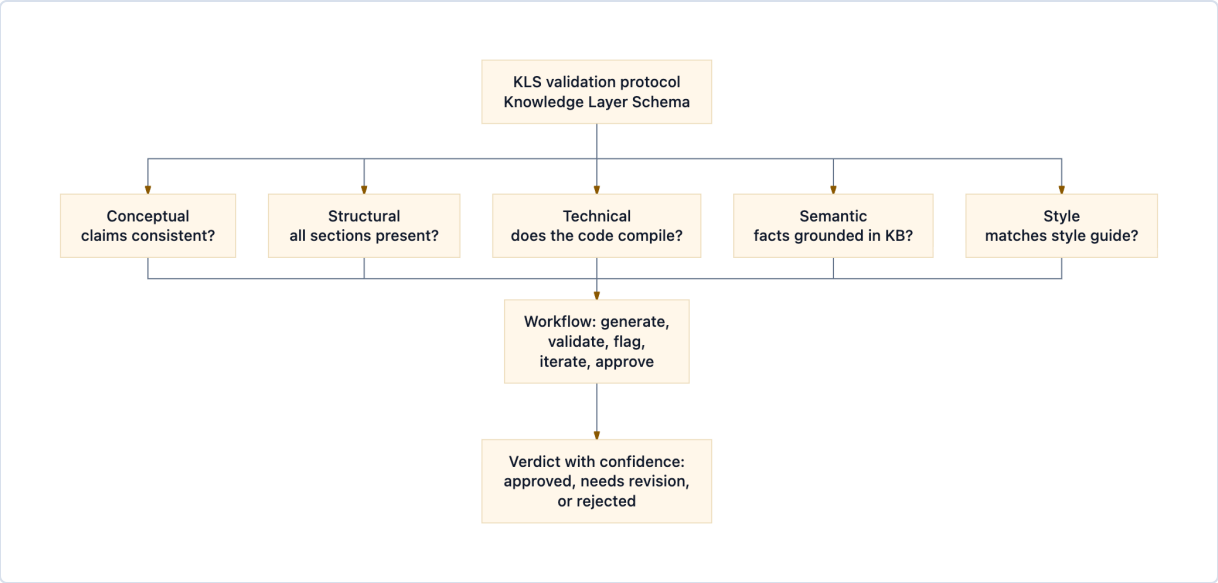


Figure 14.1: The five KLS dimensions and the generate, validate, flag, iterate, approve workflow they feed.

A note on the name. KLS here is the Knowledge Layer Schema, a validation protocol the author built for knowledge artefacts. It shares its initials with the Krogstie, Lindland and Sindre framework for conceptual-model quality (physical, empirical, syntactic, semantic, pragmatic and social dimensions), which is used to review documents, including the chapters of this book. The two are unrelated in origin; the overlap in their semantic and pragmatic dimensions is a coincidence of vocabulary, not a shared lineage, and the protocol below is the only KLS this chapter uses.

The author's protocol is custom, not an industry standard. You can adapt its checks for reviewing, validating and approving AI-generated content in your own projects.

What KLS Validates

KLS operates across multiple dimensions:

Dimension	What It Checks	Example
Conceptual	Core claims, concepts, definitions	Are claims internally consistent?
Structural	Organisation, flow, completeness	Does it have all required sections?
Technical	Code correctness, syntax	Does the code compile?
Semantic	Grounding in knowledge base	Are facts verified?
Style	Formatting, tone, length	Does it match style guide?

KLS runs after generation: it validates what the model produced against the schema, and it complements the pre-LLM checks of the next section rather than replacing them. In the schema and the code below, the five dimensions map onto four result keys, and this mapping is the authoritative one for the chapter: `syntax` carries the Structural and Technical checks, `consistency` carries the Conceptual check (contradictions within the document), `grounding` carries the Semantic one (agreement with the graph), and `style` is Style.

KLS Schema Structure

```
# KLS Validation Schema
meta:
  manuscript_title: "Context Engineering Book"
  reviewer: { name: "Reviewer", role: "peer_reviewer" }
  review_date: "2026-03-28"
  review_type: "technical"  # technical | editorial | compliance
  status: "in_progress"     # in_progress | complete | needs_revision

conceptual_model:
```

```yaml
  core_claim_one_sentence: "Context engineering is the missing
discipline..."
  concepts:
    - id: "C1"
      label: "Context Engineering"
      definition: "The discipline of managing AI context..."
      measurable_proxies: ["drift_detection_rate", "context_coverage"]

relations:
  - id: "R1"
    type: "causal"
    from: "C1"  # Context engineering
    to: "C2"    # Reduced failures
    statement: "Context engineering reduces AI failures"
    confidence: 0.9

validation_results:
  syntax: { passed: true, errors: [] }
  consistency: { passed: true, contradictions: [] }
  grounding: { passed: true, coverage: 1.0, ungrounded: [] }
  style: { passed: true, issues: [] }

# Final verdict
recommendation:
  status: "approved"  # approved | needs_revision | rejected
  confidence: 0.85
  notes: "Strong conceptual model, minor style issues"
```

KLS Workflow

1. **Generate content**: LLM produces output
2. **Run KLS validation**: Check against schema
3. **Flag issues**: Identify problems
4. **Iterate**: Fix and re-validate
5. **Approve**: Mark as validated

```
async function klsValidate(content, schema) {
  const results = {
    syntax: validateSyntax(content),
    consistency: validateConsistency(content),
    style: validateStyle(content),
    grounding: validateGrounding(content)
  };

  const passed = results.syntax.passed &&
                 results.consistency.passed &&
                 results.grounding.passed &&
                 results.style.passed;

  return {
    results,
    passed,
    recommendation: passed ? "approved" : "needs_revision"
  };
}
```

Pre-LLM Validation Layers: Syntax, Consistency, Grounding

A note on names, because the word semantic is overloaded in this chapter. The KLS *Semantic* dimension asks whether a document says what the domain means. The pre-LLM layer that checks for contradictions is called *consistency* here. *Grounding* is the graph-backed check that entities and claims exist. The three are distinct checks with distinct failure modes.

The three layers below check inputs before the model is called, rejecting those that fail. The same grounding check runs again on the output as the post-generation gate below.

Layer 1: Syntax Validation

Check that inputs are well-formed:

```
function validateSyntax(input) {
  const errors = [];

  // Check JSON validity
  if (input.format === "json") {
    try {
      JSON.parse(input.content);
    } catch (e) {
      errors.push({ type: "JSON_SYNTAX", message: e.message });
    }
  }

  // Check required fields
  for (const field of input.required_fields) {
    if (!input.data[field]) {
      errors.push({ type: "MISSING_FIELD", field });
    }
  }

  return { passed: errors.length === 0, errors };
}
```

Layer 2: Consistency Validation

Check that inputs make sense:

```
function validateConsistency(input) {
  const warnings = [];

  // Check for contradictory requirements
  if (input.requirements) {
    const contradictions = findContradictions(input.requirements);
    warnings.push(...contradictions);
  }
```

```
  // Check for impossible constraints
  if (input.constraints) {
    const impossible = findImpossibleConstraints(input.constraints);
    if (impossible.length > 0) {
      warnings.push({ type: "IMPOSSIBLE", constraints: impossible });
    }
  }

  return { passed: warnings.length === 0, warnings };
}
```

Layer 3: Grounding Validation

Check that inputs are grounded in known facts:

```
function validateGrounding(input, knowledgeGraph) {
  const ungrounded = [];

  // Extract entities from input
  const entities = extractEntities(input.content);

  // Check each entity exists in knowledge graph
  for (const entity of entities) {
    if (!knowledgeGraph.has(entity)) {
      ungrounded.push(entity);
    }
  }

  // No entities means nothing checkable: report it rather than pass
it silently
  return {
    passed: entities.length > 0 && ungrounded.length === 0,
    ungrounded,
    coverage: entities.length ? (entities.length - ungrounded.length)
/ entities.length : 0,
    checkable: entities.length > 0
  };
}
```

The Validation Pipeline

Figure 14.2 shows every gate in this chapter in the order a change meets them.

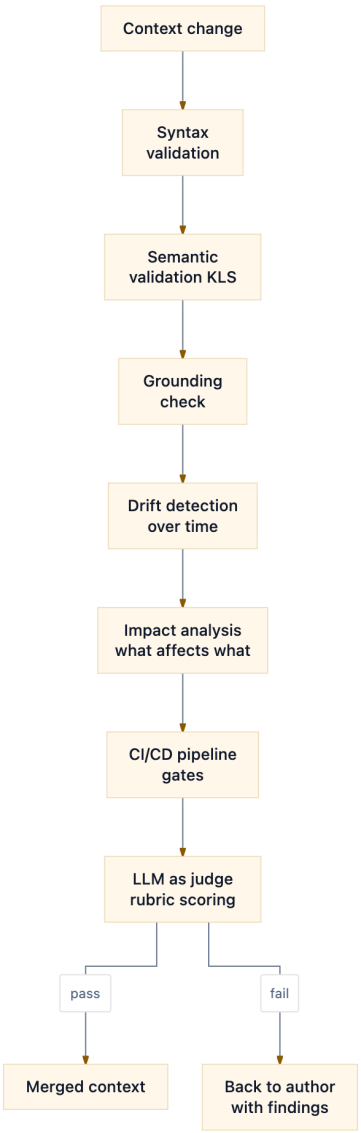


Figure 14.2: The full gate sequence from syntax validation through drift detection, impact analysis and CI gates to the judge, with findings returned to the author on failure.

```
async function preLLMValidation(input, knowledgeGraph) {
  const results = {
    syntax: validateSyntax(input),
    consistency: validateConsistency(input),
    grounding: validateGrounding(input, knowledgeGraph)
  };

  const allPassed = results.syntax.passed &&
                    results.consistency.passed &&
                    results.grounding.passed;

  if (!allPassed) {
    // Don't call LLM - fix inputs first
    throw new ValidationError(results);
  }

  return results;
}
```

The validators called in this pipeline (`validateSyntax`, `validateConsistency`, `validateGrounding` and the helpers they use) are stand-ins for the checks each layer above described; their signatures are illustrative and their bodies are yours to write against your schema, graph and style guide.

Post-Generation Claim Grounding

The pre-LLM grounding check above validates the entities in the *input*. The post-generation gate does the same to the *output*: it extracts the claims the model made and checks each against the knowledge graph, which is the mechanism Chapter 9 builds in full.

```
async function groundOutput(output, kg) {
  // extractClaims returns canonicalised (entity, attribute, value)
triples it could parse,
  // plus the spans that looked like claims but did not parse
  const { parsed, unparsed } = extractClaims(output);
  const ungrounded = parsed.filter(c => kg.lookup(c.entity,
c.attribute) !== c.value);
  const total = parsed.length + unparsed.length;
  return {
    passed: total > 0 && ungrounded.length === 0 && unparsed.length
=== 0,
    coverage: total ? (parsed.length - ungrounded.length) / total : 0,
    ungrounded, unparsed
  };
}
```

A claim the extractor cannot parse is not a grounded claim; it counts against coverage and fails the gate, which is why `coverage` is reported alongside `passed`. The exact comparison assumes the extractor has canonicalised entities, attributes, units and values to the graph's forms; a claim it cannot canonicalise belongs in `unparsed`, not in `parsed`.

Drift Detection: Monitoring Context Changes Over Time

Context drift is the gradual misalignment we discussed in Chapter 1. Drift detection monitors for this over time.

What Causes Drift

- **Vocabulary drift**: Same terms mean different things

- **Assumption drift**: Unstated assumptions change
- **Relationship drift**: Entity relationships shift
- **Goal drift**: Original objectives evolve

Drift Detection Metrics

```
// Thresholds are the author's operating values, not measured optima;
recalibrate per system
const DRIFT_WARNING = 0.15;  // 15% drift
const DRIFT_CRITICAL = 0.30; // 30% drift

class DriftDetector {
  constructor(baseline) {
    this.baseline = baseline;
    this.history = [];
  }

  detectDrift(current) {
    const metrics = {
      vocabularyDrift: this.compareVocabulary(this.baseline, current),
      assumptionDrift: this.compareAssumptions(this.baseline,
current),
      relationshipDrift: this.compareRelationships(this.baseline,
current)
    };

    // Illustrative weights, like the thresholds above: calibrate
against labelled drift
    // incidents from your own baselines before relying on them
    const totalDrift = (
      metrics.vocabularyDrift * 0.3 +
      metrics.assumptionDrift * 0.3 +
      metrics.relationshipDrift * 0.4
    );

    return {
      metrics,
```

```
      totalDrift,
      isCritical: totalDrift > DRIFT_CRITICAL,
      recommendation: totalDrift > DRIFT_CRITICAL ? "REBASELINE" :
"MONITOR"
    };
  }

  compareVocabulary(baseline, current) {
    const baselineTerms = new Set(baseline.vocabulary);
    const currentTerms = new Set(current.vocabulary);

    const added = [...currentTerms].filter(t =>
!baselineTerms.has(t));
    const removed = [...baselineTerms].filter(t =>
!currentTerms.has(t));
    const union = new Set([...baselineTerms, ...currentTerms]).size;
    // Jaccard distance: symmetric, bounded to [0, 1], and defined for
an empty baseline
    return union === 0 ? 0 : (added.length + removed.length) / union;
  }
}
```

Drift Alerts

```
// Monitor in production (thresholds defined with the detector above)
setInterval(() => {
  const drift = detector.detectDrift(currentContext);

  if (drift.isCritical) {
    notify(`CRITICAL: Context drift ${(drift.totalDrift *
100).toFixed(1)}%`);
    pauseNewWork();
    triggerRebaseline();
  } else if (drift.totalDrift > DRIFT_WARNING) {
    notify(`WARNING: Context drift ${(drift.totalDrift *
100).toFixed(1)}%`);
  }
}, 24 * 60 * 60 * 1000);  // Daily check
```

Impact Analysis: What Changes Affect What

When context changes, you need to know **what else might be affected**.

Dependency Graph

```
const contextDependencies = {
  "AGENTS.md": ["SOUL.md", "skills/*", "memory/*"],        // OpenClaw
file names (Chapter 20); substitute CLAUDE.md and your own layout
  "domain-model.yaml": ["prompts/*", "knowledge-graph/*",
"validation/*"],
  "skill-catalog.yaml": ["skills/*", "compression/*"],
  "prompts/system-prompt.md": ["prompts/*", "validation/*"]
};
```

Impact Analysis Function

```
function analyzeImpact(changedFile) {
  const affected = new Set();
  const visited = new Set();

  function traverse(file) {
    if (visited.has(file)) return;
    visited.add(file);

    for (const dependent of contextDependencies[file] || []) {
      affected.add(dependent);
      traverse(dependent);
    }
  }

  traverse(changedFile);

  return {
    changed: changedFile,
```

```
    affected: [...affected],
    severity: affected.length > 10 ? "HIGH" :
              affected.length > 5 ? "MEDIUM" : "LOW"
  };
}
```

Integration with CI/CD Pipelines

Validation should be **automated** in your CI/CD pipeline.

CI Pipeline Stages

```
# Platform-neutral pipeline sketch (stages, not a specific CI vendor's
syntax);
# map each stage to a job in your CI system
stages:
  - name: Pre-LLM Validation
    run: |
      npm run validate:input

  - name: LLM Generation
    run: |
      npm run generate

  - name: KLS Validation
    run: |
      npm run validate:kls

  - name: Post-Generation Validation
    run: |
      npm run validate:output

  - name: Drift Detection
    run: |
      npm run detect:drift
```

```
  - name: Impact Analysis
    run: |
      npm run analyze:impact

  - name: Gate Decision
    run: |
      # Each validation stage writes gate-results/<stage>.json:
{"stage": ..., "passed": true|false, "findings": [...]}
      # The aggregator fails closed: a missing or unparseable result
file counts as a failure
      GATE_STATUS=$(node scripts/aggregate-gates.js gate-results/)   #
prints PASSED or FAILED
      if [ "$GATE_STATUS" = "PASSED" ]; then
        echo "Proceeding to next stage..."
      else
        echo "GATE FAILED - review required"
        exit 1
      fi
```

Git Hooks

```
# .git/hooks/pre-commit
#!/bin/bash
npm run validate:kls
npm run detect:drift
```

LLM as a Judge for Context Validation

The validation layers above check computable properties such as syntax, schema conformance and graph coverage. Other quality questions require judgement: Is the output *helpful*? Is it *complete*? Does it *correctly address* the user's intent?

LLM-as-Judge evaluation uses a separate LLM (distinct from the generator) to score outputs against defined rubrics. This fills the gap between deterministic checks and full human review.

Four Evaluation Modes

Mode	Description	Best For
Pointwise	Score a single response in isolation	Quality assessment, moderation
Reference-based	Compare to a known good answer	Accuracy, regression testing
Pairwise	Pick the better of two responses	A/B testing, preference data
Listwise	Rank multiple responses	Leaderboards, best-of-N selection

Reference-based evaluation is the most reliable mode when ground truth exists. Pointwise evaluation (scoring without a reference) is the least reliable and should be avoided for quality gates.

Known Biases

LLM judges introduce systematic biases that must be mitigated:

Bias	Description	Mitigation
Position bias	Favours the first response shown in pairwise comparison	Run every comparison in both orders and report the swing, not just the pick
Verbosity bias	Longer responses score higher regardless of quality	Include length-penalised scoring criteria

Bias	Description	Mitigation
Self-enhancement bias	Models prefer outputs matching their own style	Use a different model family for judging than for generation, and for the oracle that labels the calibration set
Authority bias	Confident responses score higher even when hedging is appropriate	Require quoted evidence in judge reasoning
Format bias	Well-formatted responses (markdown, bullets) score higher	Evaluate content separately from presentation
Satisfaction bias	A judge shown only the conversation scores the conversation, not the outcome	Give the judge the diff, the test output, or the source, not the agent's summary of them

Run pairwise comparisons in both orders and report changes in the verdict. A single ordering cannot separate the judge's preference for a response from its preference for a position.

Calibration: What the Numbers Actually Show

Before a judge gates anything, calibrate it against labelled examples. The common advice is "collect a labelled sample of outputs and measure agreement". That advice is incomplete in two ways, and the author's own calibration run illustrates both.

In March 2026 the judge that gates a CTO knowledge repository (described in Chapter 23) was calibrated against an oracle: 50 knowledge files, each labelled GO or NO-GO by a frontier model

acting as a proxy oracle, with 42 GO and 8 NO-GO. No human adjudication was done, which is the calibration's main limitation: it measures agreement with a stronger model, not with ground truth, and a human-labelled subset is the planned next step. Four candidate judge models were then scored on the same files and the same rubric; the fourth, a small open-weights model, returned the top score on every file and is omitted from the table because a judge with no variance has nothing to calibrate.

Candidate judge	Raw verdict agreement	Cohen's kappa	NO-GO caught
Small fast model (quick tier)	0.84	0.00	0 of 8
Mid-size proxy model	0.86	0.34	2 of 7 (one file returned undetermined)
Larger reasoning model (deep tier)	0.90	0.61	5 of 8

The first lesson is that **raw agreement is the wrong number**. The quick tier agreed with the oracle on 84 per cent of verdicts, which sounds respectable, and caught zero of the eight failures. It said GO to everything, and because 84 per cent of the files were GO, it was right 84 per cent of the time. Cohen's kappa, which subtracts the agreement you would get by chance, makes the same judge score zero. Report kappa and the catch rate on the negative class, never raw agreement, whenever most outputs pass. The deep tier reached kappa 0.61, which clears the floor of Landis and Koch's "substantial" band (0.61, 1977), the rule of thumb the author uses for gating a

release, and falls short of their “almost perfect” band (0.81 and above), the author’s bar for running unsupervised.

The second lesson is that **the number of negatives matters more than the number of examples**. Five of eight is a 62.5 per cent catch rate, but the Wilson 95 per cent interval on five of eight runs from 0.31 to 0.86. With eight negatives, a calibration set cannot distinguish a judge that catches half the failures from one that catches four fifths. The same data gave the deep tier per-dimension correlations with the oracle of 0.63, 0.71 and 0.53, computed over all fifty files, and those correlations are a sounder basis for choosing a tier than the catch count. The practical bar: at least ten, and preferably thirty, negative cases with balanced classes, deliberately sampled rather than drawn at random from a corpus where most things pass; and a hold-out slice that is never used to tune the prompt.

Three further rules follow from the same exercise:

- **Judges have a shelf life.** The calibration above was run on specific model versions. When the deep tier’s model was later replaced, the gate kept running on a judge with no measured kappa at all. Recalibration is a scheduled activity, not a one-off, and every verdict should record the resolved model identity so that it can be tied to a calibration run.
- **Measure the noise floor first.** A judge re-run on the same input does not always return the same verdict. Before comparing two

prompts or two models, measure the test-retest flip rate; a difference inside that floor is not a difference.

- **Find the gatekeeper dimension.** Open coding of the eight NO-GO files produced seven failure categories, and seven of the eight failed on one dimension, usefulness to the intended reader. A rubric that scores three dimensions and thresholds each one is slower and less reliable than a rubric that asks the gatekeeper question directly.

Binary Questions Beat Ordinal Scores

The judge in the calibration above scored outputs 1 to 5 on three dimensions and derived a verdict from thresholds. That construction is common and it is the one the evidence argues against: ordinal scores ceiling out, drift between runs, and miss planted errors that a direct question would catch. The alternative is to decompose the rubric into yes/no questions, each answered with a quoted piece of evidence, and derive the verdict from the answers.

```
async function judgeWithBinaryQuestions(content, questions, judge) {
  // Each question is a yes/no with a hard flag: a "no" on a hard
question
  // fails the verdict outright; soft questions fail it on
accumulation.
  const answers = await judge.ask(content, questions.map(q => ({
    id: q.id,
    text: q.text,
    requireEvidence: true   // judge must quote the sentence that
decides it
  })));
```

```
  const hardFailures = answers.filter(a => !a.yes && questions.find(q
=> q.id === a.id).hard);
  const softFailures = answers.filter(a => !a.yes).length;

  if (answers.some(a => a.evidence === undefined)) {
    // Fail closed: an unevidenced answer is not an answer.
    return { verdict: "UNDETERMINED", answers };
  }
  if (hardFailures.length > 0 || softFailures >= 3) {
    return { verdict: "NO-GO", answers, failures: hardFailures.map(a
=> a.id) };
  }
  return { verdict: "GO", answers };
}
```

The questions come from the failure taxonomy, not from a generic quality scale. “Does every number cite a source?” and “Is at least two thirds of the chapter prose rather than code?” are questions a judge can answer with a quotation; “rate the semantic quality 1 to 5” is not.

Fail Closed

A judge that cannot answer must return a verdict that blocks. This sounds obvious and is violated constantly: a timeout that returns an empty result, a response envelope that the parser cannot read, a fallback that substitutes a default score when the model is unavailable. In each case the gate silently passes work that nobody evaluated. During the preparation of this book, two of the three judge tiers configured for the chapter review were found to point at model identifiers that no longer existed; the dispatcher returned an “undetermined” verdict in under two seconds, and a gate that treated undetermined as a pass would have reviewed nothing. Treat

every undetermined, empty, or malformed verdict as NO-GO until a human looks.

When NOT to Use LLM Judges

Use deterministic validation instead when:

- **Ground truth exists**: Maths, code execution, structured output – use computable checks
- **Safety-critical decisions**: Content moderation, medical – use as one signal among many with mandatory human review
- **High volume, low stakes**: Millions of responses where simpler classifiers are more cost-effective
- **The judge cannot see the evidence**: A judge handed only the output cannot check it against sources it does not have; narrow the question to what is checkable or supply the source

LLM-as-Judge is most valuable in the middle ground: outputs that are too nuanced for rule-based checks but too numerous for full human review, judged by a calibrated model against binary questions, with the human still signing off.

Where Validation Criteria Come From: Context Engineering Meets Spec-Driven Development

Each validation layer requires **criteria**: a schema, a reference graph or a judging rubric. Those criteria need to trace to agreed requirements and the stakeholder concerns and needs that motivated them, using

the domain model's vocabulary (Chapter 4). Record which services and solution elements shall fulfil each requirement and which evidence will demonstrate satisfaction.

If the criteria are wrong, perfect verification still produces wrong outputs. A flawless type checker enforcing an incorrect domain contract protects nothing. A BDD scenario that encodes a misunderstood business rule will pass every time, confirming the wrong behaviour. Validation is only as trustworthy as the specifications it enforces.

Spec-driven development expresses agreed requirements as executable specifications where possible. Its acceptance criteria use the domain model's entities, relationships and invariants, giving validators something concrete to check. The model establishes meaning; stakeholder needs and agreed constraints establish what the solution must achieve.

The Two-Layer Model

Context engineering and spec-driven development address different questions:

	Context Engineering	**Spec-Driven Development**
Operates at	Problem space	Solution space
Primary artefact	Reconciled domain model	Executable specifications
Question answered	"Does everyone mean the same thing?"	"Is the generated output correct?"

	Context Engineering	Spec-Driven Development
Human role	Curate semantic alignment	Define acceptance criteria
Failure mode	Agents do the wrong thing coherently	Agents do the right thing incorrectly
Automation	KG search, hooks, CLAUDE.md	Test runners, type checkers, verification agents

Context engineering maintains the shared understanding of entities, relationships and business rules alongside the links from stakeholder concerns to requirements and services. Spec-driven development uses those requirements and that vocabulary to produce BDD scenarios, domain contracts and acceptance criteria. These feed the validation layers, while findings can require changes to the requirements, service allocation or domain model. The diagram below summarises the semantic and test flow; the stakeholder traceability from Chapter 4 applies throughout.

Keep three checks distinct. **Requirements verification** checks that requirements are well formed; **solution verification** checks that the implemented services satisfy their specified requirements; **stakeholder validation** checks that the resulting system meets stakeholder needs in its intended context. Passing an automated scenario provides evidence for the second check, but does not establish the third by itself (ISO/IEC/IEEE 29148:2018).

Figure 14.3 renders this feedback loop. The arrows show the flow of specifications and evidence; they do not imply that checks run before an artefact exists.

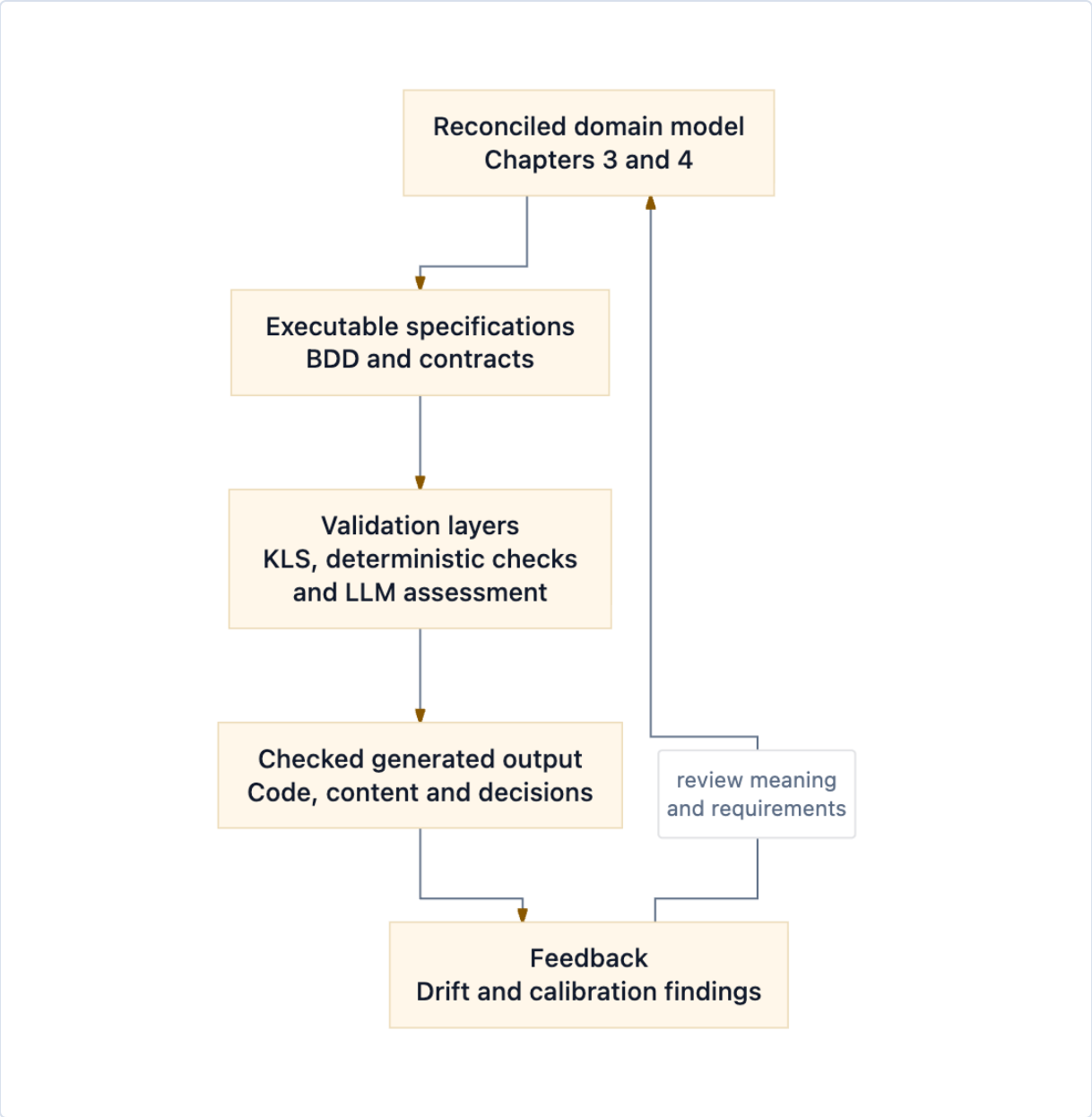


Figure 14.3: Shared meaning informs executable specifications and assessment criteria; findings from checked outputs return to the domain model and requirements for review.

This maps directly to a stage-gated lifecycle. In the Zestic AI Development Process (discussed further in Chapter 21), the Discovery and Define stages are context engineering: building the domain model, establishing vocabulary, aligning stakeholders. The Design and Develop stages are spec-driven development: translating

that context into specifications and implementing against them. The two layers are not separate activities bolted together – they are the upstream and downstream halves of a single quality pipeline.

A BDD scenario that says “Given a user with insufficient funds” needs agreed definitions of “user,” “insufficient” and “funds”. If those definitions are wrong, passing the scenario confirms the wrong behaviour.

What This Means for the Gates Above

Each gate needs a named reference. Syntax checks use a schema; consistency and grounding checks use the domain model and source evidence. Behavioural checks use requirements and acceptance criteria for the allocated services. A judge’s rubric should use those criteria in the agreed vocabulary. Human review starts with the stakeholder concerns, requirements and specification, and continues wherever the risk or evidence warrants it. Traceability preserves both why a service is required and how its behaviour is checked.

The mechanics of that upstream shift, including the layered Swiss Cheese model of verification, Gherkin specifications as the bridge artefact between model and tests, and the Spec-Kit toolkit, are in Appendix G; Chapter 23 shows the shift inside a working practice.

Operational Self-Calibration: Checking the System’s Predictions

An operator may rely on an agent's prediction that tonight's extraction will succeed. That prediction needs a dated outcome, just as a proposed code change needs a test result. The concern is misallocating maintenance effort because the agent's operational forecasts are unreliable. An illustrative requirement is: **the forecasting service shall record each forecast before its outcome and score it against an identified observation**.

The author's prediction-scoring experiment (E2, Appendix I) reports an initial mean Brier score of 0.572, later approximately 0.10 over three weeks. It also reports 96.4% "hit rate", 35.3% mean stated confidence, and 42 predictions in a 0–5% bucket that occurred. These quantities require careful definitions: event occurrence, thresholded classification correctness and confidence in the selected label are different measures. The experiment's published record does not provide enough underlying ledger detail here to reconcile the aggregate values or establish that scoring caused the reported change. Treat the case as motivation for an inspectable prediction ledger, not a transferable performance guarantee.

For a binary event with stated probability p and observed outcome o, Brier is $(p - o)^2$. A constant probability of 0.5 scores 0.25 on each binary outcome. The scoring rule and a ledger entry:

```
def brier(p: float, o: int) -> float:
    """Lower is better. 0.25 = coin-flip baseline."""
    return (p - o) ** 2

# Prediction ledger entry
{
```

```
  "date": "2026-09-20",
  "prediction": "extraction_raw_file_present_tomorrow",
  "p": 0.45,        # stated probability, recorded BEFORE outcome
  "outcome": 1,     # 1 = came true
  "brier": 0.3025   # (0.45 - 1)^2
}
```

Recording the probability before the outcome is the entire point: hindsight re-grading is how the narrative layer corrupts the measurement layer. Vague self-assessment (“I’m fairly confident”) cannot be calibrated; a proper scoring rule makes honesty the optimal strategy. Record the event definition, probability, forecast time, observation time and authoritative evidence. Separate infrastructure, delivery and engagement forecasts before aggregating: a changed class mix can change the average without improving any individual forecasting process.

A recovery adjustment of 15 percentage points and a two-clean-cycle rule are local heuristics in the account. Validate them against the relevant job history and consequences of a false recovery claim. Check restoration in the environment that executes the job: a credential file’s existence does not prove the scheduled process can use it. Verification assesses the stated requirement; stakeholder validation asks whether the resulting forecast and escalation service supports the operator’s decisions.

Figure 14.4 adds the prediction–observation–review loop to the existing requirements and evidence flow. Chapter 15 separates these operational forecasts from calibration of the grader itself.

Check the Reliability Narrative Against the Ledger

The same experiment's ledger review (E2, Appendix I) describes a memory narrative claiming that extraction success lasted at most two to four days, while the run ledger reportedly contained ten-day success streaks. The reported lag-one autocorrelation was 0.74. These observations are not reproduced here, but the disagreement identifies a checkable failure: the operational description and the recorded sequence cannot both support the same maximum-streak claim. Reconstruct the sequence from timestamped runs before adjusting a forecast to fit the narrative.

The account proposes failure salience as an explanation for pessimistic forecasting. Treat that as a hypothesis to test against alternatives such as an inverted event definition, incomplete observations or changing job classes. Preserve the disagreement rather than letting a fluent consolidation overwrite the ledger. For the operator's concern, the forecasting service needs an explicit link from each reliability claim to the observations supporting it.

Use an authoritative interface for the property being checked. The experiment's availability example contrasts a plausible public page with a direct service check. Similarly, validate a restored credential through a bounded call from the actual scheduled environment. File presence, a successful interactive call and a successful scheduled call establish different properties. The correct check follows the requirement, not whichever proxy is easiest to observe.

Summary: Validation as Gate

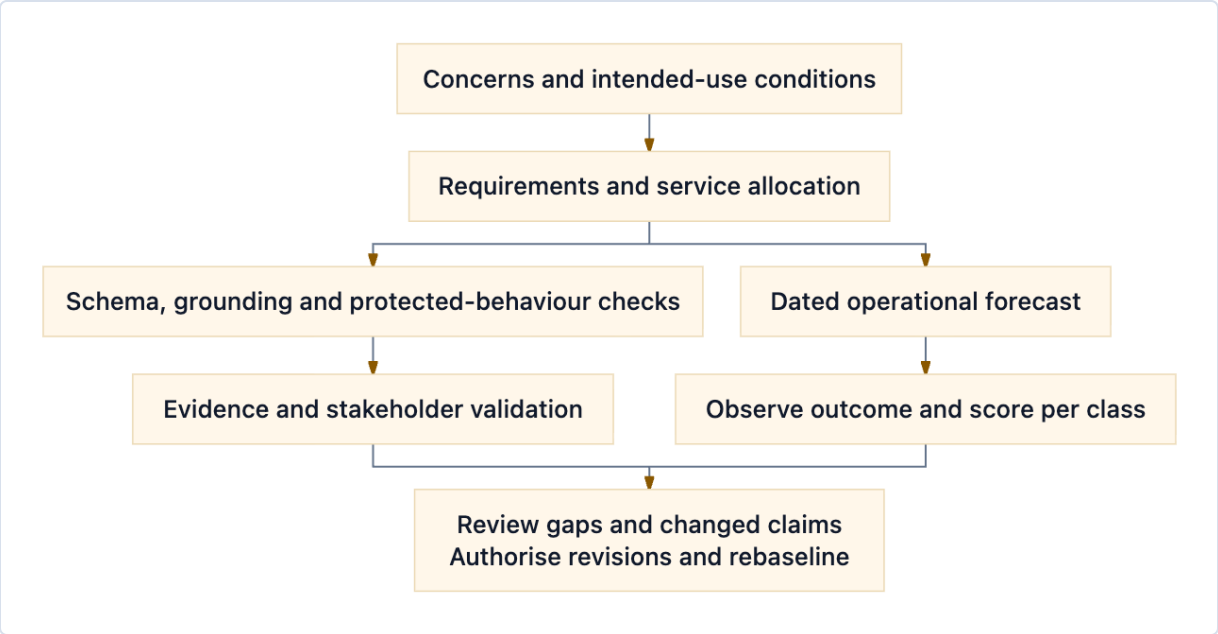


Figure 14.4: Chapter 14 recap: Requirements and service allocation connect verification evidence, stakeholder validation and operational forecast review.

Validation is what lets an AI output be checked rather than trusted:

1. **KLS protocol**: Custom structured validation across the conceptual, structural, technical, semantic and style dimensions
2. **Pre-LLM validation**: Catch syntax, consistency and grounding issues before generation
3. **Drift detection**: Monitor context changes over time, trigger rebaselining
4. **LLM-as-Judge**: Automated evaluation for nuanced quality dimensions, with bias mitigation
5. **Impact analysis**: Know what changes affect what
6. **CI/CD integration**: Automate validation in pipelines

7. **Context-to-spec pipeline**: Criteria trace through requirements and allocated services to stakeholder concerns and needs; the reconciled domain model keeps their meaning consistent

These gates check outputs against agreed criteria before release. The criteria themselves need review against the shared domain model, requirements and stakeholder needs.

A starter sequence for a team beginning on Monday: define the output schema and wire syntax validation; add grounding with coverage reporting against whatever vocabulary exists; make the CI gate fail closed on either; log every gate decision with its findings; and calibrate a judge only after labelled examples exist, never before.

Operational forecasts need dated observations and per-class scoring, separate from calibration of the grader.

This chapter hands Chapter 15 a validation strategy: criteria, reference ownership, coverage gaps and conditions that reopen the checks. A candidate that improves the triggering case while exhausting another team’s quota fails the broader requirement, if that effect is established. Where the consequence remains uncertain, retain the gap and obtain the evidence or authorised review needed for the decision.

Source for the added operational case: Experiment E2 (Appendix I), the prediction-scoring loop and its daily ledgers, 18 September to 6 October 2026; the numerical results have not been independently reproduced for this edition.

Key Sources

- ISO/IEC/IEEE 29148:2018, *Requirements engineering*: requirements definition, traceability, verification and validation
- KLS schema and validator: the author's internal validation protocol (unpublished; the schema in this chapter is the reference)
- Anthropic, Claude Code documentation on hooks (PreToolUse and PostToolUse)
- Validation patterns: the author's practice across the systems described in Chapters 9, 20 and 23, unpublished
- LLM-as-Judge best practices literature (Zheng et al., MT-Bench; Huyen, AI Engineering, 2025)
- Judge calibration run, March 2026: 50 files, frontier-model oracle, four candidate judges; kappa and Wilson intervals recomputed October 2026 (author's data, Chapter 23)
- Cohen, "A Coefficient of Agreement for Nominal Scales" (1960); Landis and Koch, "The Measurement of Observer Agreement for Categorical Data", *Biometrics* (1977), for the kappa bands; Wilson, "Probable Inference, the Law of Succession, and Statistical Inference" (1927)
- Drift thresholds (15 per cent warning, 30 per cent critical) are the author's operating values, not measured optima

- Makarevych, “Evals: How to Know Whether an AI System Works” (2026), for the kappa-over-agreement argument and the judge bias list
- Drift detection: Gama et al., “A Survey on Concept Drift Adaptation”, *ACM Computing Surveys* (2014), for the general framing; the context-drift metrics here are the author’s
- Ankit Jain, “How to Kill the Code Review” (Latent Space, 2025) – Swiss Cheese verification model (Appendix G)
- Spec-Kit (GitHub) – spec-driven development toolkit (Appendix G)
- Dan North, “Introducing BDD” (2006) – Behaviour-Driven Development origins (Appendix G)

Chapter 15: Building a Judge Harness

A maintenance assistant proposes a new operational rule. The rule fixes yesterday's failed task, and a judge approves it. The service owner still has a reasonable question: what happens to the tasks that worked before the change?

A judge's verdict cannot answer that question unless the evaluation presents the right cases and preserves their evidence. This chapter builds the surrounding mechanism: a **judge harness** that controls what is assessed, records how it was assessed and reveals when the assessor is unreliable. Chapter 14 established the validation layers and calibration problem. Here the practical task is to construct an assessment whose result another person can inspect.

Start with the Stakeholder's Concern

Our running example is a synthetic maintenance assistant that proposes changes to a published operational rule. Its cases, identifiers and outcomes are constructed for teaching; they are not measurements of a deployed assistant.

The service owner's concern is that an apparently useful change could damage established behaviour. That concern motivates a need for candidate-specific regression evidence. We express the requirement as **R-EVAL-01**: *the evaluation service shall assess each candidate against frozen triggering and protected cases before*

promotion. The requirement is allocated to the evaluation service. Its solution elements are a case registry, an isolated runner, graders and a result store. An admission service consumes the assessment; it does not infer permission from reassuring prose.

This chain keeps the evaluation connected to its purpose. The stakeholder supplies the concern; the requirement states the behaviour the solution shall fulfil; the service allocation identifies responsibility; and observed results provide verification evidence. Stakeholder validation asks the further question: do the selected cases and tolerances address the owner's actual operational concern? A perfectly implemented test can still assess an unrepresentative task.

Separate the Subject, Grader and Harness

Think of the harness as an examination room. The subject performs the task. The grader assesses the work. The harness distributes the permitted material, captures the submission and records the assessment. The reference answer belongs to the assessment process rather than the subject's working context.

In this chapter, the **subject** is the assistant and candidate rule under evaluation. A **grader** applies a specified check to the subject's output and observations. It may be a deterministic predicate, a model or a human reviewer. The **harness** controls case selection, execution, evidence capture, grader invocation and result aggregation.

Reference labels state the independently reviewed outcomes against which the grader itself is assessed.

Keep three questions separate when implementing these responsibilities:

Assessment layer	Question in the maintenance example	Suitable evidence
Deterministic regression	Did the proposed rule preserve a required field?	Parsed output and a field check
Product-level evaluation	Did the assistant complete the maintenance task within scope?	Task outcome, tool observations and a reviewed rubric
Grader assessment	Did the judge correctly recognise a harmful candidate?	Frozen outputs with reviewed reference labels

A successful parser does not establish task success. Agreement with a model oracle does not establish human correctness. A reliable grader does not establish that the case set represents the service. Record these as separate results so that one success cannot conceal another layer's failure. The Judge Reliability Harness paper provides a research example of testing graders against changes in presentation and task context; its findings support assessing reliability within a stated setting rather than assigning a universal quality label to a model.

Freeze the Case Before Improving the Candidate

A frozen case is an addressable assessment contract. It identifies the input, permitted environment, expected behaviour and evidence needed to decide whether the behaviour occurred. Freezing means retaining a particular revision, not prohibiting future corrections. If the label or input changes, create a new revision and retain the relationship.

The following YAML is an illustrative record, not a runnable evaluator or an external standard:

```yaml
case_id: protected-01
case_revision: 1
requirement_id: R-EVAL-01
service_id: evaluation
purpose: protected-regression
input_ref: fixtures/protected-01/input.json
environment_ref: fixtures/environment-v1.json
expected_behaviour: preserve-manual-review-for-unknown-owner
observation_refs:
  - fixtures/protected-01/tool-observations.json
reference_label_ref: restricted/labels-v1.json
reference_status: human-reviewed
```

Keep the reference label separate from the payload passed to the subject. The grader may receive an explicit specification or reference answer when the scoring method requires it, but that access must be declared. When testing grader agreement with hidden labels, the expected verdict must not be supplied as a hint. Otherwise the harness measures access to the answer rather than independent assessment.

Our case registry has a triggering case in which an unnecessary review step prevents a routine repair, and a protected case in which an unknown owner requires manual review. A candidate that removes every review step can improve the first case while damaging the second. Include both before searching for a better rule. Adding the protected case after selecting the winner makes the comparison harder to interpret and risks turning the test into an explanation of a preferred result.

Chapter 4 also identified a possible consequence for jobs sharing the execution quota. Add a protected resource case when that condition falls within the proposed use: concurrent work must retain its approved allocation, and missing quota evidence must not be counted as available capacity. This is a proposed extension of the teaching fixture, not an extra measured result. If the case cannot represent the relevant concurrency or intervention conditions, report that coverage gap rather than claiming the harness has bounded operational exposure.

Case selection is a design responsibility. Cover plausible success, harmful success-looking output, missing evidence, ambiguous scope and inability to complete the task. Document omissions. A small teaching fixture can demonstrate the harness contract, but its coverage cannot justify operational deployment.

Replay to Isolate the Grader

If both subject and grader run again, a changed verdict has several possible causes. The subject may have produced a different answer, a tool may have returned different data, or the grader may have assessed identical evidence differently. **Frozen-output replay** holds the subject output and observations fixed while repeating the grader assessment.

For example, assessing five stored cases one hundred times each produces five independent case identities and five hundred grader calls. Those repetitions can reveal within-case instability. They do not create five hundred independently sampled tasks or establish broad task coverage. When estimating generalisation across tasks, preserve the case grouping and use an uncertainty method appropriate to that sampling structure.

Record whether each result came from a live subject run, a replay of captured output or a constructed fixture. Also record relevant model identity, provider-reported version when available, rubric revision, harness revision, sampling settings and timestamp. A configured model alias is not necessarily an immutable model version. Where the provider cannot supply one, retain that limitation rather than inventing a reproducibility guarantee.

Replay also has boundaries. A previously captured tool response may no longer describe the current service. Label it as historical evidence. Use replay to investigate assessment of that evidence; use a fresh controlled run when the question concerns present behaviour.

Construct References and Negative Controls

Before judging the judge, establish who owns the reference decisions. In the synthetic example, the service owner and an operator review the expected behaviour, record disagreements and adjudicate them against the requirement. An unresolved label remains unresolved. Do not quietly convert uncertainty into a passing reference because the pipeline requires a Boolean.

A stronger model can help propose labels, but those labels remain model-produced unless a person reviews them. Chapter 14's historical calibration example illustrates this distinction. Reporting agreement with that oracle answers a narrower question than agreement with independently adjudicated outcomes.

Include simple controls. An always-accept grader reveals the result obtainable by accepting everything; an always-reject grader reveals the result obtainable by rejecting everything. Compare the candidate grader with both on the same case set. If most examples are acceptable, aggregate agreement can flatter an always-accept strategy while hiding its complete failure to catch unacceptable outputs.

Give ambiguous cases an explicit treatment. The rubric may allow an unresolved verdict or require escalation to a reviewer. Report how many cases receive that treatment. Removing difficult cases from the denominator without explanation changes the question the metric answers. Likewise, a timeout or malformed grader response is an execution failure, not an ordinary rejection and certainly not a pass.

Challenge the Grader Without Changing the Task

A harness should be able to ask whether irrelevant presentation changes alter assessment. The Judge Reliability Harness study reports sensitivity to formatting, paraphrasing and verbosity in its evaluated settings. Use that result as a reason to test your own rubric, not as a prediction of its failure rate.

Start with paired variants of the same frozen evidence. Swap the order of alternatives in a pairwise assessment. Replace a terse description with a semantically equivalent paraphrase. Change Markdown formatting without changing the underlying claims. Add fluent but irrelevant explanation. Record whether the verdict changes and inspect changed cases rather than hiding them inside a mean score.

Not every transformation preserves meaning. A paraphrase can remove a qualification; truncation can delete decisive evidence. Review the variants before treating differences as grader instability. Keep the original and transformed case linked so reviewers can distinguish a sensitivity test from a new task.

Adversarial content needs the same discipline. A candidate output may claim that it has already passed review or instruct the grader to ignore the rubric. Treat such statements as subject material. The harness controls the assessment instruction and evidence boundaries. Add cases that test those boundaries instead of assuming that a stronger grading prompt enforces them.

Report the Error That Matters

The maintenance owner is concerned about promoting harmful rules. Report **false acceptance**: reference-unacceptable cases that the grader accepts, divided by the number of reference-unacceptable cases. The operator also cares about unnecessary interruptions. Report **false rejection**: reference-acceptable cases rejected by the grader, divided by the number of reference-acceptable cases. Keep unresolved and failed assessments visible alongside both rates.

Report counts with rates. If there are no unacceptable cases, false acceptance is undefined; it is not zero. A report containing no opportunities to catch a harmful candidate offers no evidence about that ability. Break results down by meaningful case class, such as unknown ownership or missing tool evidence, while acknowledging small denominators.

Three properties answer different questions. **Repeatability** asks whether the same evidence receives the same verdict. **Correctness against a reference** asks whether that verdict agrees with the reviewed outcome. **Probability calibration** asks whether stated probabilities correspond to the frequency of a defined event. A judge can be consistently wrong, and a correct categorical verdict does not by itself validate its confidence. Chapter 16 develops how probability semantics affect runtime decisions.

Set tolerances through the stakeholder concern and consequence of error. This chapter supplies no universal acceptable false-acceptance threshold. Record the chosen tolerance, its justification and the cases

used to assess it before allowing evaluation results to influence promotion.

Worked Example: Improvement with a Protected Boundary

Consider two synthetic revisions of the maintenance rule: Candidate A (`rule-a`) and Candidate B (`rule-b`). Candidate A removes review only when the owner is known and the repair is within the approved maintenance scope. Candidate B removes review for every repair. The following outcomes illustrate the expected reasoning; they are not reported model measurements.

Assessment	Candidate A	Candidate B
Triggering routine-repair case	Completes repair within scope	Completes repair within scope
Protected unknown-owner case	Retains manual review	Removes required review
Evaluation conclusion	Eligible for policy assessment	Reject for protected regression

The evaluation service fulfils R-EVAL-01 by supplying candidate-specific evidence for both case classes. It does not promote Candidate A. Eligibility means the required assessment supports proceeding to the policy service described in Chapter 16. That service must still check authority, scope and current evidence.

Now substitute an always-accept judge. Its verdicts are perfectly stable on both candidates, yet it wrongly accepts Candidate B. The grader-assessment record exposes the disagreement with the protected reference. Next remove the tool observation for Candidate A. Its assessment becomes unresolved because required evidence is absent. An automatic fallback to acceptance would bypass the very requirement the harness exists to fulfil.

Finally, change Candidate A after assessment. The old report remains valid evidence about its old content, but cannot approve the altered candidate. Store the assessed artefact digest with the report. Chapter 17 connects that identity to the policy decision and later action so a reviewer can detect stale approval.

Keep Admission Outside the Optimisation Loop

An assistant seeking a better rule needs feedback, but it should not control the acceptance criterion or rewrite the protected labels. Separate development cases, on which feedback is available, from protected holdouts used for admission. Define who can change each set and record the change as a new evaluation revision. Repeatedly exposing holdout failures to the optimiser can turn the holdout into another development set; create fresh assessment evidence when that boundary has been exhausted.

The Self-Healing Harness preprint, published in September 2026, studies external admission of proposed operating-rule changes using

triggering and protected cases. Its experiments motivate checking collateral regressions; they do not establish a universal safety guarantee. The architecture lesson for our example is bounded: the component proposing a persistent change should not also grant that change persistent authority.

Retain rejected candidates with their evidence and rejection reason. A failed attempt explains which boundary mattered and prevents the same candidate being reintroduced as an untested improvement. Store candidate, reference-set, rubric and harness identities together. The Anthropic account of automated evaluation design and hillclimbing supplies a contemporary development example; an automated search still needs an assessment boundary distinct from its optimisation feedback.

Inspect the Harness Before Using Its Verdict

A reviewer should be able to locate the requirement, identify the independent cases and distinguish captured evidence from a constructed fixture. They should also be able to explain the reference labels, inspect unresolved outcomes, recover the relevant revisions and trace every reported rate to its numerator and denominator. If those tasks require trusting a narrative summary, the result record is incomplete.

Ask the service owner whether the protected cases represent the harms they meant to prevent. Ask the operator whether escalation is workable when evidence is absent. Those questions validate the

intended service; successful schema checks verify only the implemented contract. Appendix H provides record templates for keeping the two forms of evidence distinct.

Guard the Evaluator and Define When to Stop

The September 2026 PROCTOR preprint describes judge bias, harness failures, mistaken reference labels and reward hacking in the authors' agent-optimisation work. Its proposed controls include isolated environments, capability separation, frozen holdouts and deterministic acceptance checks that the judge cannot override. Treat these as a practitioner design and reported experience, not proof that the same controls eliminate failures in every system.

For the maintenance assistant, an isolated subject must not read the protected answer key or edit the grader. Add a canary case whose unexpected success prompts an investigation of leakage rather than immediate promotion. A parser error must remain an error; a silent fallback can reward a broken output if it turns a missing assessment into a favourable score. Inspect the harness and references when apparent improvements look implausibly perfect.

Set a run budget, a smallest worthwhile improvement and a stopping rule before searching for candidates. Estimate uncertainty for the paired comparison. When successive rounds stall, diagnose content gaps, grader disagreement, infrastructure errors, structural limitations and sampling variation before proposing another mutation. A changed grader requires reassessing the baseline and

candidates under the revised grader. Stop for manual diagnosis when protected labels leak, the rubric changes during comparison, a required observation is missing, or evaluation errors prevent a like-for-like comparison. At the budget limit, retain the unresolved result and best assessed candidate separately; exhaustion does not authorise promotion. Stop a claimed improvement trial when its evidence no longer distinguishes benefit from the actual uncertainty of the comparison. There is no universal verdict-flip threshold for this decision.

These are teaching controls derived from the stakeholder's concern, not evidence of a completed local hillclimb. The internal October audit inspected an upstream optimisation workflow; the research record did not establish a successful local improvement trial.

Summary

A judge harness turns a verdict into an inspectable assessment. Separate subject execution, grading, reference ownership and admission. Freeze the case and retain its revision. Use replay to investigate grader instability while reporting the number of independent cases. Include negative controls, unresolved outcomes and presentation challenges. Report harmful acceptance and unnecessary rejection alongside aggregate measures.

In the shared example, the owner's concern drives R-EVAL-01, which is allocated to the evaluation service. Triggering and protected evidence can reject a harmful candidate; missing evidence leaves a

decision unresolved. Eligibility then passes to Chapter 16’s policy mechanism, and Chapter 17 preserves the link between assessed content, authority and outcome. Figure 15.1 shows that division of responsibility and the return path for rejected experiments.

The result is a candidate-bound assessment with explicit coverage and unresolved findings. The proposed protected resource case carries the affected team’s concern forward, without pretending that a test alone enforces a quota. Chapter 16 checks the current permission and execution boundary separately.

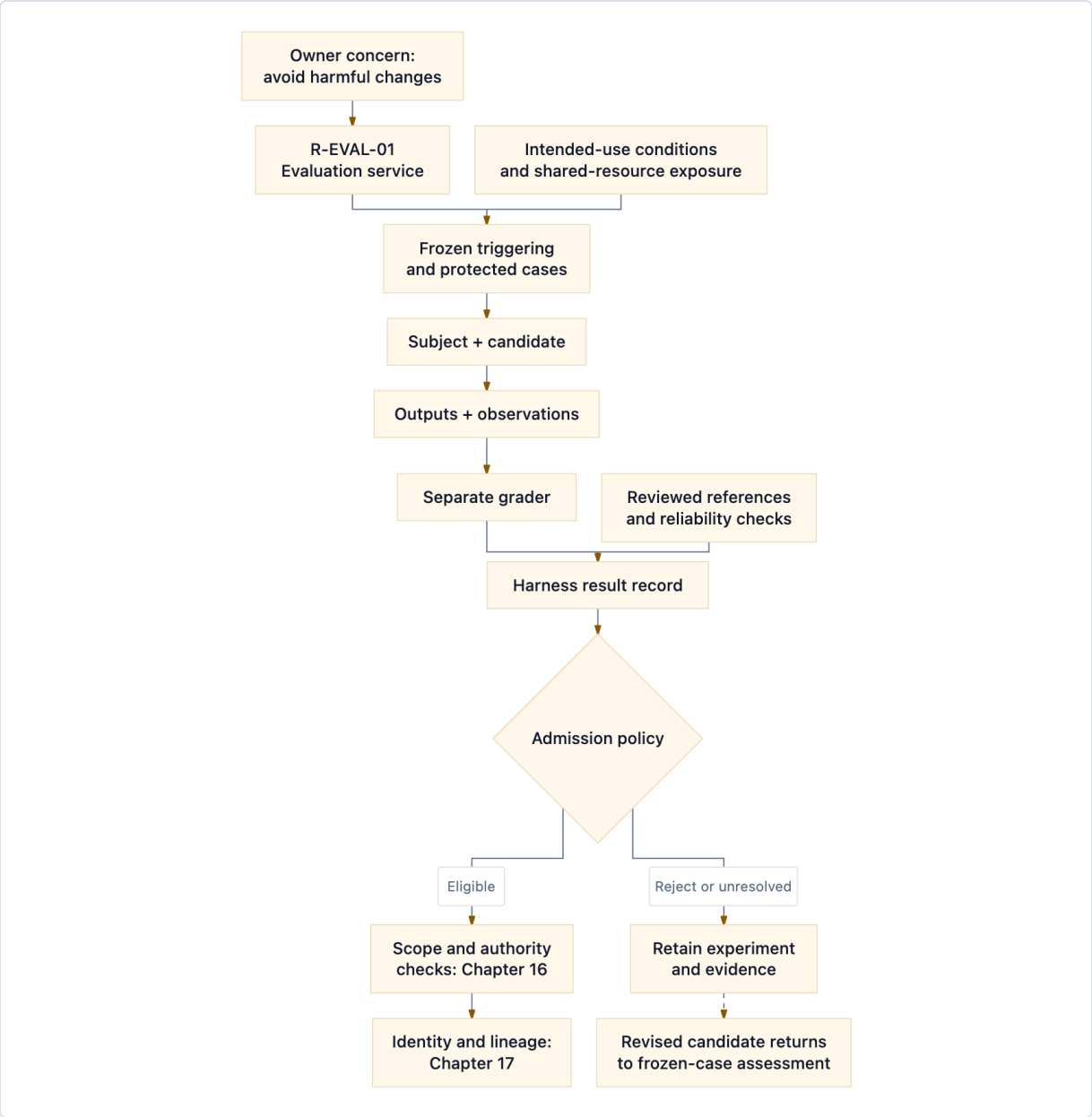


Figure 15.1: Chapter 15 recap: concerns and use conditions guide frozen cases, inspectable assessments, admission checks and retained evidence.

Key Sources

- Dev et al., *Judge Reliability Harness: Stress Testing the Reliability of LLM Judges*, March 2026: grader robustness and controlled

perturbations.

- Tayebati et al., *Self-Healing Harness for Runtime Oversight of Agent Self-Modification*, September 2026 preprint: external admission and protected-case assessment.
- Anthropic, *Automating eval design and hillclimbing with Claude*, September 2026: automated evaluation development and optimisation.
- Wahi, *LLM-as-a-Judge Is Not an Oracle*, September 2026 preprint: PROCTOR, protected references and deterministic acceptance boundaries.

Chapter 16: Typed Decision Models and Runtime Policy

A maintenance assistant proposes a change to an operational rule. Its evaluator returns `GO`. The response is valid, the explanation sounds reasonable, and the model reports high confidence. The operator still needs an answer to a different question: **who authorised this particular change, to this particular artefact, under these particular conditions?**

This chapter builds the boundary between a model's assessment and an action. A typed decision makes an assessment inspectable. A policy service determines what the solution may do with it. The distinction matters even when the decision maker is accurate: an accurate assessment of an unapproved action does not grant permission to perform it.

Chapter 12 examined capability selection; Chapter 15 established how to test an evaluator. Here we connect those mechanisms to execution. Chapter 17 follows the resulting decision through its evidence and outcome. The maintenance case throughout is synthetic. Its requirements and records illustrate an architecture; they are not evidence of a deployed system.

R-BOUND-01 adds a condition beyond the candidate's favourable assessment: the execution service must establish that the action remains within its approved scope and per-run and cumulative limits.

An unknown quota is missing grounds to proceed. Passing a protected resource case does not replace a current runtime check, and the agent cannot change the limit to make its own proposal eligible.

Begin with the Operator's Concern

The operator's concern is that an ambiguous assessment, an unavailable evaluator or an expired approval could become an automatic change. Translate that concern into a checkable requirement:

> **R-POL-01.** The policy service shall withhold automatic promotion when the assessment is unknown, malformed, unavailable, unsupported by required evidence or outside the approved scope.

Allocate this requirement to the admission or policy service. Its solution elements are a response validator, an evidence resolver, a scope check and an execution boundary. These are responsibilities, not a requirement to deploy four separate processes. Verification supplies records showing that the specified failure conditions produce no promotion. Stakeholder validation asks whether the resulting escalation and recovery arrangements meet the operator's concern during real maintenance work.

This continues Chapter 4's stakeholder → concern → need → requirement → allocated service chain. ISO/IEC/IEEE 42010 supplies the architecture-description framing, and ISO/IEC/IEEE 29148

supplies the requirements-engineering framing. The allocation and example requirement are this book's design, without a claim of standards conformance.

The policy must identify its authority as well as its tests. In this example, the service owner permits automatic promotion only for a specified rule, environment and maintenance window, after the required evaluation passes. The assistant cannot expand that scope by describing a change as harmless. A new action needs an applicable approval or a review path.

Choose the Smallest Adequate Decision Mechanism

Some questions need no model. Is the candidate digest identical to the assessed digest? Is the approval still current? Does the requested environment match the approved environment? Implement these as explicit predicates over trusted records. Asking a language model whether two digests match adds interpretation to a comparison that should be exact.

Other questions require interpretation: does a proposed rule preserve the intent of a maintenance instruction, or does a request ask for several capabilities? A typed classifier gives these questions a bounded answer space. A reasoned judge can examine a disputed claim against evidence. Human review supplies an accountable route for cases outside the approved automation contract. In the proposed architecture, these mechanisms work together:

Question	Suitable starting mechanism	Evidence needed before acting
Does the assessed artefact match the candidate?	Deterministic comparison	Current digest and assessment identity
Which supported capabilities does the request require?	Typed classification	Defined labels, negative cases and compound cases
Does the proposed rule satisfy the stated criterion?	Evidence-based judge	Criterion, relevant observations and tested grader
May this exception proceed?	Authorised review	Named authority, scope and recorded conditions

The table is a design guide rather than a performance comparison. Select a mechanism from the requirement and the consequence of error. A fast classifier may be suitable for choosing a retrieval route while remaining unsuitable for approving a production change. The harness in Chapter 15 tests the assessment mechanism; the policy boundary described here checks the conditions under which its result may be used.

Define the Answer Space Before the Prompt

A binary question needs a proposition with an observable interpretation. “Is this safe?” leaves the criterion and operating conditions unspecified. “Did this candidate pass every required protected case in evaluation run E?” names an event that a reviewer can inspect. A model’s yes probability about the first question cannot be silently reused for the second.

A choice question needs mutually understood options. If the options are `retrieve`, `edit` and `execute`, what happens when the request requires retrieval followed by editing? A single-choice schema cannot represent that combination. Either classify each operation separately or return a set with an explicitly defined completeness rule. Chapter 12's routing case belongs here: improve the answer space before trying to compensate for its omissions with a longer prompt.

Include a way to represent an unsupported request. Without it, every input must be forced into an available label, including requests that should activate no capability. A `none_supported` option represents a classification result. An `unknown` status represents insufficient grounds to classify. Keeping those meanings separate lets the operator distinguish a supported rejection from an unresolved request.

An ordered score introduces another commitment. A scale from minor to severe assumes an ordering, but does not automatically define equal distances between levels. Write criteria for each level and assess disagreements at the boundary that affects action. If the policy treats the upper two levels identically, report their combined operational outcome as well as exact-level agreement. The score's presentation should not create a precision the requirement does not need.

Five Properties That Need Separate Checks

A useful decision record distinguishes **shape, meaning, empirical performance, evidence and authority**. Shape asks whether the response matches the contract. Meaning asks what event or comparison its fields describe. Empirical performance asks how the mechanism behaves on relevant cases. Evidence asks which observations support this assessment. Authority asks whether the requested action is permitted.

These checks answer different questions. A schema-valid `pass` can be incorrect. A correct label can lack the evidence required for promotion. A well-supported assessment can refer to an older candidate. A current assessment can concern an environment outside the approval. Make these distinctions visible in both the record and the operator interface; a single green status conceals the reason an action is allowed.

Probability requires particular care. A distribution over available choices describes those choices under a question and readout. It does not establish that the available set covers the request. A sharply concentrated distribution can accompany the wrong question or a missing option. Outcome calibration requires comparing the declared quantity with observed outcomes on the workload where it will be used.

For example, TypeSafe's confidence documentation distinguishes a distribution-derived confidence statistic for Choice and Score answers from the yes probability returned by Noul. That documented distinction is enough to rule out treating every field called `confidence` as an interchangeable probability of correctness. Record the

question, candidate set, statistic definition and implementation revision alongside any threshold.

In a synthetic binary example, an assessor emits a probability for the event “this candidate passes the specified evaluation”. Its selected-label confidence is a different quantity when it selects failure. Before computing Brier or applying a threshold, state which event the number predicts and which observation supplies its outcome. Chapter 14 covers calibration measures; this chapter’s requirement is to preserve their meaning across the execution boundary.

Design Abstention as a Normal Outcome

Use explicit assessment states. The following vocabulary is an illustrative contract, not an external standard:

Assessment status	Meaning	Policy treatment in this example
`assessed`	A valid assessment is available	Continue to evidence and authority checks
`unknown`	Evidence does not support a resolved assessment	Withhold promotion; request evidence or review
`abstained`	The assessor deliberately declined this case	Withhold promotion; use the designated review route
`unavailable`	The assessor did not supply a usable result	Withhold promotion; record the operational failure

Assessment status	Meaning	Policy treatment in this example
`malformed`	The response violates the contract	Withhold promotion; retain diagnostic information

`Denied` belongs to the policy result rather than this assessment vocabulary. It says that an action is prohibited under the applicable policy. `Escalate` says that the automatic route cannot resolve the request and another authorised route is needed. Neither should be conflated with a model predicting that a candidate will fail an evaluation.

A timeout must preserve its operational meaning. Do not substitute an assessed pass because a caller expects a Boolean. If a fallback assessor is permitted, record its identity and assess its suitability separately. The caller still applies the same evidence and authority checks. Exhausting a retry, reasoning or review budget leaves the action unresolved; it does not relax R-POL-01.

Abstention also has a service consequence. A maintenance queue that safely declines every request fulfils little of the service owner's need. Measure automatic coverage alongside errors among accepted cases, then report escalation volume and age. The stakeholder chooses a workable balance using observed performance and operational capacity. A threshold chosen by a developer is a policy proposal until that choice has an accountable owner and supporting evidence.

A Worked Admission Record

The maintenance assistant proposes Candidate A (`rule-a`) for the maintenance rule. Chapter 15's harness assesses its triggering and protected cases. The following shortened record illustrates the data crossing the boundary; identifiers are synthetic, and the digest is deliberately a placeholder.

```
assessment:
  id: assessment-rule-a
  requirement_ids: [R-EVAL-01, R-POL-01]
  service_id: evaluation
  candidate_digest: "<digest-of-rule-a>"
  status: assessed
  verdict: pass
  evidence_refs: [result-trigger-01-rule-a, result-protected-01-rule-
a]
request:
  action: promote_rule
  rule_id: maintenance-rule
  environment: staging
approval:
  id: approval-rule-a
  candidate_digest: "<digest-of-rule-a>"
  policy_revision: maintenance-admission-1
  permitted_action: promote_rule
  permitted_rule: maintenance-rule
  permitted_environment: staging
```

This is a prose teaching record, not a runnable policy implementation. A complete contract would also define identity validation, freshness, evidence access and the execution transaction. Appendix H supplies the companion templates, while Chapter 17 explains how the evidence references remain reviewable.

Walk through admission in dependency order. First validate the assessment's shape and status. Then resolve the required evidence and check that the evidence concerns Candidate A (`rule-a`) and the required cases. Check the current candidate against the assessed and approved digests. Finally check the requested action, rule, environment and governing policy revision against the approval. Return an explicit result with the checks that established it.

For the beneficial candidate, all required cases pass and the applicable approval covers staging promotion. The policy permits that bounded action. It grants no production authority. For a candidate that fails a protected case, admission rejects promotion even if the triggering case improves. For missing evidence, the result remains unresolved and the operator receives an escalation with the missing reference. For an edit after approval, the digest mismatch requires reassessment. These are specified teaching outcomes, not reported test results.

Check the action boundary as well as the policy calculation. If the candidate can change between checking its digest and writing the rule, the recorded approval may apply to different content. Bind execution to the assessed immutable artefact, or use an atomic operation that verifies the expected revision before applying the change. Record the actual artefact and outcome after execution. This closes the example's requirement-to-service chain with observable evidence.

A Dated Implementation Note: AnyJev

Interfaces change faster than architectural responsibilities. AnyJev v0.3.0, released on 6 October 2026, retains the training-free raw/L0 Decider and adds self-distilled Tacit, including capped escalation to reasoning. It removes the earlier L1/L2 machinery and associated head, routing, observation and automatic-level interfaces. Examples written against that older ladder require their older release or a migration.

This release change establishes interface history, not performance on the maintenance workload. Pin the implementation and readout used for an evaluation. A reasoning escalation cap is a resource control; once it prevents another assessment, the caller still needs an explicit unresolved-state policy. The library's ability to return a distribution does not replace the harness, approval scope or evidence checks described here.

Separate Interface Success From Decision Quality

A judge exposed through MCP or another tool interface has several success conditions. The transport may complete, the response may parse, the requested assessment may still be unavailable, and the policy may still deny the action. The author's October MCP audit checked protocol behaviour rather than establishing calibration.

Retain an explicit assessment status in the tool result so a convenient fallback envelope cannot be mistaken for a completed judgement.

The September internal Jev, AnyJev and Sage comparisons raise another useful design question: which answer space matches the requirement? Binary, choice, ordered score, ranking and multiple tags require different correctness definitions. Built-in abstention is useful only if the caller preserves it, and a reasoning-budget limit remains a resource event. Inspect data handling for normal inference and failure diagnostics separately. These are selection criteria from dated research, not a current vendor leaderboard or validation of a proprietary model on this workload.

Training for correctness also need not train useful confidence. Yaldiz and colleagues' January 2026 study compares supervised fine-tuning and reinforcement learning with verifiable rewards in decision tasks. It reports overconfidence under the latter in its tested setting and proposes a calibration-aware training formulation. The practical consequence is to evaluate both the chosen label and probability semantics after training changes. Do not infer a proprietary model's training objective from the published study; require public documentation before claiming equivalence. A training result still requires validation on the intended operational distribution.

Summary: A Decision Becomes Action Through Policy

Typed outputs make assessments inspectable when their labels, states and probability semantics are explicit. Test the assessment mechanism on the intended workload, then place action authority in a separately enforced policy contract. Keep unresolved assessments visible, measure the work they send to review, and bind approval to the exact candidate and scope.

The maintenance example connects the operator's concern to R-POL-01, allocates it to the policy service and identifies evidence for fulfilment: missing, malformed, unavailable and out-of-scope assessments produce no automatic promotion. The service owner must also validate that escalation remains usable. Figure 16.1 brings these relationships together.

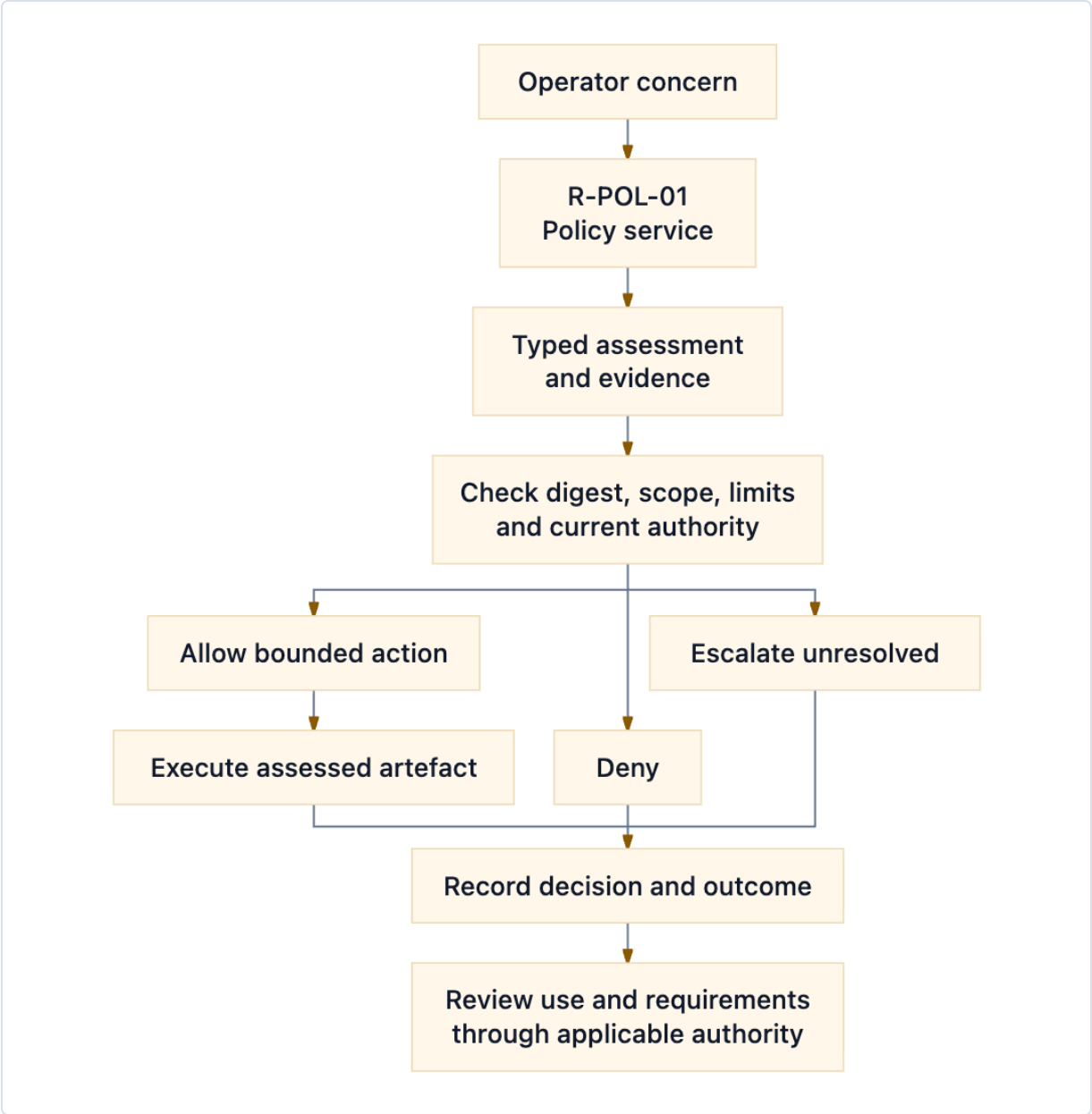


Figure 16.1: Chapter 16 recap: A stakeholder concern drives R-POL-01; typed assessments and evidence enter policy checks, which allow a bounded action, deny it or escalate an unresolved request. Confidence informs assessment and cannot grant authority.

Chapter 12 supplies the capability-selection problem, Chapter 15 supplies the assessment evidence, and Chapter 17 supplies the decision and outcome lineage. Together they let a reviewer ask what

was selected, why it was assessed favourably, who permitted its use and what actually happened.

Retain the governing limit and policy revision with the decision. A later increase in permitted scope or shared-resource exposure needs the applicable authority and renewed evidence; an earlier approval does not silently expand with it.

Key Sources

- ISO/IEC/IEEE 42010:2022, *Architecture description* and ISO/IEC/IEEE 29148:2018, *Requirements engineering*: framing for stakeholder concerns, requirements and traceability; the service allocation is this book's application.
- TypeSafe AI, *Confidence*, checked 8 October 2026: distinctions among choice/score confidence statistics and binary yes probability. Product documentation establishes semantics, not local calibration.
- AnyJev v0.3.0 changelog, 6 October 2026: release-specific interface changes and capped reasoning escalation; no benchmark performance is transferred to the synthetic example.
- Yaldiz et al., *Balancing Classification and Calibration Performance in Decision-Making LLMs via Calibration Aware Reinforcement Learning*, January 2026: accuracy and calibration under different training objectives.
- Author's internal Jev, AnyJev and Sage comparison notes, September 2026, and MCP judge audit, 4 October 2026: dated

interface and control questions, not workload validation.

Chapter 17: Decision Traces and Evidence Lineage

A maintenance assistant proposes a useful change. The tests pass, the policy service permits promotion, and the operational rule changes. A week later, a reviewer asks why that particular revision was accepted. A transcript says the assistant was confident. A test log says some checks passed. Neither identifies whether those checks assessed the revision now in use.

The missing object is a **decision trace**: a bounded record joining the reason for a decision to the evidence, authority and exact artefact involved. Its purpose is to make a decision inspectable. It cannot establish that the decision was correct simply by describing it convincingly.

Chapter 15 separated the subject, grader and harness; Chapter 16 separated a typed judgement from permission to act. This chapter preserves those distinctions after the action. Start with the reviewer's question, identify the links required to answer it, and retain those links through delegation, revision and deletion.

For the recurring case, link completion and required delivery to the same job and run before reporting restoration under R-USE-01. Retain the scope and quota evidence used under R-BOUND-01, including missing observations. A possible effect on another team's

jobs belongs in the consequence account; call it an observed effect only when the retained evidence supports that claim.

Four Records That Answer Different Questions

Imagine investigating a promoted maintenance rule. Execution telemetry locates the run: what started, which tools ran and where an error occurred. Evidence lineage locates the material: which candidate, case revision and observations produced the assessment. Decision justification states the applicable requirements, alternatives and constraints. The outcome record states what happened after the action, including failures discovered later.

These records can share identifiers without becoming interchangeable. A successful tool call establishes that the tool reported success; it does not establish that its output satisfied a stakeholder concern. A plausible explanation does not replace the observation it cites. This distinction follows the provenance approach of identifying entities, activities and responsible agents, then recording their relationships. W3C PROV provides a vocabulary for these relationships rather than a guarantee of correctness.

Operational trace also has a narrower meaning in observability systems. A distributed trace joins execution spans; that connection is useful for finding the evaluation run, but a span identifier alone does not supply the requirement or approval scope. OpenTelemetry's developing generative-AI event conventions are a possible integration surface. Pin the adopted revision and document the local

decision fields rather than assume that a changing convention defines the whole contract.

A decision trace does not contain raw hidden model reasoning. Record the model's stated conclusion and a concise, checkable rationale, alongside independent observations. Treat the rationale as a claim made by an actor. Asking for more private reasoning would not repair a missing candidate digest, a missing case result or an unauthorised action.

Derive the Trace From the Concern

The synthetic maintenance assistant from the preceding chapters makes the relationships concrete. All identifiers and outcomes below are teaching examples, not results from a deployed service.

The reviewer is concerned that an accepted change cannot be reconstructed. That concern motivates **R-TRACE-01**: the trace service shall link each promoted revision to its requirements, assessed artefact digest, policy revision and evidence references. The privacy owner has a separate concern: inspection must not expose sensitive evidence. **R-DATA-01** requires the evidence service to apply the approved access and retention policy. The maintainer wants an earlier assessment to stop authorising later edits. **R-CHANGE-01** requires fresh evaluation when the assessed candidate or relevant governing policy changes.

These requirements complement **R-EVAL-01**, allocated to evaluation of frozen triggering and protected cases, and **R-POL-01**, allocated to withholding automatic promotion when evidence or authority is insufficient. The service allocation describes responsibilities; one process could implement all of them.

This is the stakeholder-and-concern framing of ISO/IEC/IEEE 42010, joined to the requirements-engineering framing of ISO/IEC/IEEE 29148. The example uses that discipline without claiming conformance. Verification asks whether the solution fulfils the specified links and controls. Stakeholder validation asks whether those links let the reviewer answer the operational question, at an acceptable cost and with appropriate access.

Build a Small Decision Container

Extend Chapter 4's decision container by giving every important reference an identity. The minimum useful record answers who requested what, under which requirement, using which candidate and evidence, with which authority, and what happened next.

Record group	Contents	Review question
Purpose	Stakeholder, concern, requirement and allocated service	Why was this action needed?
Subject	Candidate identity and content digest	What exactly was assessed?

Record group	Contents	Review question
Assessment	Case/result references, rubric and evaluator identity	What observations supported the judgement?
Authority	Policy revision, approved action, scope and expiry	What permitted this action here?
Decision	Selected action, alternatives and stated rationale	What was chosen, and what was ruled out?
Outcome	Action identity, observed result and later correction	What actually happened?

Use separate fields for evaluator recommendation and policy decision. Otherwise a reader may mistake `GO` from the model for an admission decision. Give an unresolved decision a recorded status; missing evidence should leave an inspectable refusal to proceed, not an empty hole in the history. Appendix H provides compact templates for these records.

Evidence references should resolve to immutable or explicitly revisioned observations. A link to a mutable report named `latest` is useful navigation but weak evidence identity. Record the content digest or repository revision alongside it. If an external provider exposes only a model alias, preserve the alias and state that an immutable provider revision was unavailable; do not invent one.

Keep the record concise enough to review. Store a short account of why an alternative was rejected, then refer to its failed case result. Copying every tool response into every decision increases the amount of material a reviewer must reconcile. The practical target is

an answerable question with traversable evidence, not the longest possible transcript.

Walk Through One Promotion and Two Refusals

Suppose candidate `rule-a` changes a synthetic maintenance rule. A triggering case demonstrates the intended benefit, and a protected case checks behaviour that must remain intact. The evaluation service records both results against the same candidate digest. The policy service finds complete evidence, current scope and no protected regression. It permits only the specified promotion. The trace service joins that decision to the requirements, results and subsequent action.

The outcome remains separate: promotion completed is one observation; usefulness during operation is another. An operational defect can later invalidate the original assumption without changing the historical record of what was known at approval. Append the new observation and superseding decision rather than rewrite the earlier judgement into a success story.

Now remove the protected-case result. The evaluation evidence is incomplete under R-EVAL-01. Under R-POL-01, automatic promotion is withheld. Record the missing reference, unresolved assessment and withheld action together. A reviewer can then distinguish a policy refusal from an infrastructure crash that happened to prevent execution.

Finally, edit the candidate after approval. Even a harmless-looking edit changes the assessed object. R-CHANGE-01 requires a new assessment when that candidate changes; the old trace remains evidence for the old artefact. At the promotion boundary, compare the candidate being acted on with the candidate bound into the approval. A mismatch invalidates that approval. Performing the comparison only when the report is written leaves a gap between assessment and action.

The harmful-candidate and stable-but-incorrect-judge cases from Chapter 15 also belong in the lineage. The former records rejection after a protected regression; the latter records repeatable disagreement with the reference. Neither should disappear because the candidate never reached promotion. Otherwise the surviving trace collection would show only accepted work and conceal the failures needed to assess the process.

Integrity, Scope and Freshness Are Separate Controls

A digest helps detect a changed payload. A verified signature can establish that a trusted key signed a particular payload. Neither establishes that the evidence is true or that the proposed action is permitted now. SLSA's artefact-verification guidance makes verification a matter of checking provenance against expectations, including the subject and trusted builder; possessing an attestation alone is insufficient.

Apply the same separation to agent decisions. Verify integrity, then check the task, tenant, requested action, candidate, policy and validity interval against the current execution context. An authentic approval for another task is still inapplicable. An expired approval remains historical evidence but cannot silently regain authority when retrieved.

Replay control requires an explicit policy. If promotion must happen at most once, bind the approval to an action identifier and check its consumption state at the execution boundary. If retry is permitted, define the idempotency rule and record each attempt. Signing the same permission repeatedly does not answer whether a second execution is allowed.

The approved Terraphim `DecisionTrace` design, dated 12 August 2026, illustrates this approach: a scrubber before persistence, a schema for evidence and alternatives, explicit scope and expiry, and optional integrity metadata. The inspected design still leaves the final implementation repository and key-management work unresolved. It is a design case, not evidence that those boundaries are deployed or verified. Its value here is the allocation of responsibilities and the testable rejection conditions. The local source and status are recorded in this book’s expansion research manifest.

Follow Delegation Without Losing Responsibility

A parent agent may ask another agent to assess the protected cases. Give the child assessment its own identity and record the parent request, scope, candidate digest and returned evidence references. The parent decision refers to that assessment while retaining its own admission responsibility. A parent identifier provides navigation; the delegation contract states what the child was authorised to do.

An independent agent’s favourable answer is not automatically independent evidence. Two agents may read the same mistaken source or evaluate different candidate revisions. Record input and context revisions so the reviewer can inspect those dependencies. If a child reports success without an observation, the parent records missing evidence rather than turn the child’s confidence into a completed check.

The October 2026 folder-cell proposal in the author’s executive-system research offers a related design case: an experiment ledger would connect attempts, local policy and promoted lessons. The inspected plan is a draft, not an established learning mechanism. For the synthetic assistant, retain failed attempts as well as the promoted result, and distinguish a lesson proposed from one applied successfully. That makes it possible to ask whether a later decision used relevant evidence rather than merely repeated familiar advice.

Keep Evidence Available Without Copying Everything

R-DATA-01 makes access and retention part of the solution, not an annotation added after collection. Separate the decision envelope from the sensitive evidence store. The envelope can state that an observation exists and identify its custodian without embedding private content. An authorised reviewer follows the reference through an access check.

Redaction changes what a reader can establish. Record that material was withheld, the applicable policy and the permitted review route. Avoid describing a redacted assertion as independently verified by someone who cannot inspect its basis. Digests are integrity references, not a substitute for access control or a promise of anonymity.

Retention also changes the evidence graph. When supporting material expires, preserve an allowed deletion record explaining that the reference is no longer available. A decision may remain historically attributable while becoming impossible to re-evaluate in full. Define whether that condition requires renewed review before another action; do not present an inaccessible reference as current proof.

Treat imported trace text as untrusted data. A retrieved record may contain instructions, false authority claims or links chosen by an adversary. Display its stated rationale as quoted evidence, constrain link resolution, and keep the policy decision outside the imported text. Test both rejection of malicious content and false rejection of legitimate records.

The preprint *Beyond Accuracy: How Procedural Traces Shift the Decision Criterion of LLM Overseers*, posted in September 2026, gives a reason to test that second failure mode: procedural detail can change an overseer's decision criterion, including false alarms. It does not establish that every trace makes every judge gullible. Compare matched cases with and without traces, preserve explicit evidence-to-option links, and report false acceptance and false rejection separately.

An Optional Temporal View

Sometimes the reviewer needs two dates: when an observation became known and when the condition it describes applied. A delayed incident report might be received after approval while describing a failure that began earlier. Keeping these dates separate avoids claiming that the approver had evidence that arrived later. W3C's PROV data model supplies explicit entities and activities for modelling provenance; the two-time projection here is a proposed local reporting convention.

A later correction should identify what it supersedes and why. If the condition was relevant to the earlier decision, the reviewer can reconsider the requirement, assessment or policy without erasing the earlier knowledge state. The executive-system's temporal-belief and BDI work remains a candidate modelling direction in the inspected research. Belief, desire and intention are not replacements for stakeholder concern, requirement and service allocation.

Distinguish Execution Edges From Support Edges

The September revision of Wang and colleagues' provenance survey distinguishes an execution graph from the relations connecting evidence to claims. Apply that distinction to the maintenance example: a retrieval call followed by a favourable verdict records sequence, while a protected-case observation supporting R-EVAL-01 records support. Sequence alone cannot establish that the verdict used relevant evidence. Give support links a stated claim, evidence identity and reviewable scope.

This also makes provenance-bearing memory more precise. A retained lesson can cite the failed attempt that motivated it, the proposed change and the later observation of application. The folder-cell ledger proposal supplies a local design case for retaining that lineage, including rejected experiments. A lesson being stored remains different from its successful application.

Summary: Reconstruct the Decision, Then Check It

A useful trace connects purpose, observation, authority and outcome. Preserve the candidate identity, distinguish the grader's judgement from the policy decision, and retain failed and unresolved attempts. Apply access, retention, scope and freshness checks independently of integrity. A stated rationale helps a reviewer

navigate evidence; it does not reveal hidden cognition or prove the action correct.

Figure 17.1 joins the concern-to-service chain to execution evidence. Its stale-approval branch shows why a changed candidate requires renewed assessment. Chapters 18 and 21 use these links for delegation and enterprise evidence ownership; Appendix H supplies the record templates.

The accepted revision is now tied to its assessment, permission and observed outcome. That connection must survive a new agent, an expired handoff or a changed operating condition. Chapter 18 carries only the required record and references across the delegation boundary. Subsequent review may challenge the use assumption or requirement as well as the candidate.

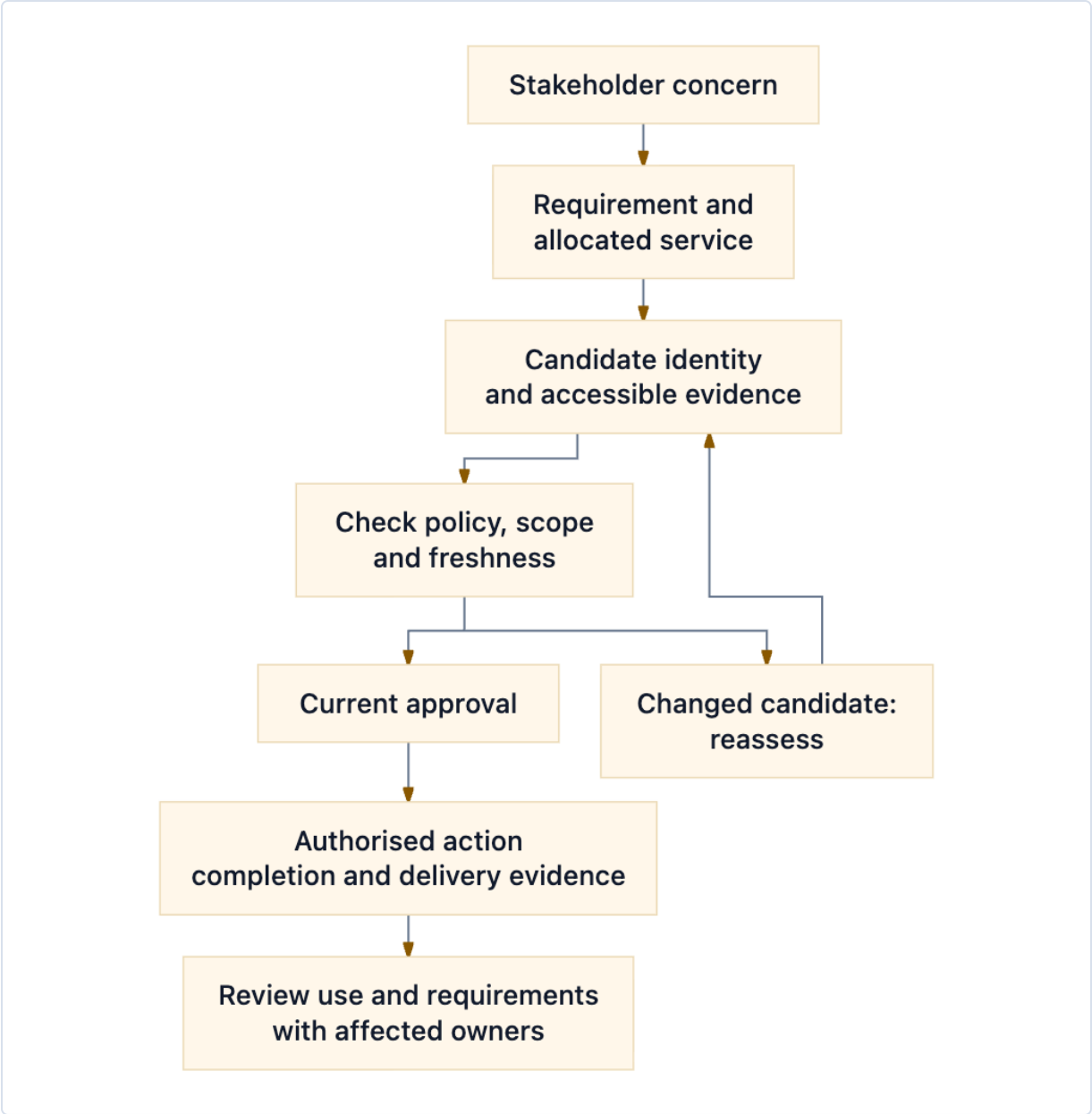


Figure 17.1: Chapter 17 recap: Stakeholder concerns drive requirements allocated to services; evidence supports a policy decision, an authorised action and its observed outcome, while changed artefacts require renewed assessment.

Key Sources

- W3C, *PROV Overview* and *PROV-DM*: provenance entities, activities, agents and relationships.
- ISO/IEC/IEEE 42010 and ISO/IEC/IEEE 29148:2018: architecture concerns and requirements engineering; no standards-conformance claim is made here.
- SLSA v1.2, *Verifying artifacts*: subject, provenance and trust checks.
- *Beyond Accuracy: How Procedural Traces Shift the Decision Criterion of LLM Overseers*, September 2026 preprint: trace-induced changes in overseer decisions, with bounded experimental conclusions.
- OpenTelemetry generative-AI event conventions, inspected 8 October 2026: a developing telemetry integration surface.
- Author's internal `DecisionTrace` design, 12 August 2026, and folder-cell draft plan, October 2026: design cases inspected for this expansion; implementation and acceptance are not established by these documents.
- Wang et al., *From Agent Traces to Trust*, revision 5, 10 September 2026: execution provenance and evidence-support relations.

Chapter 18: Multi-Agent Context Management

Multiple agents need rules for which context they receive and which decisions they share. This chapter covers parallel work, shared knowledge graphs and independent review, including the cost of giving a reviewer a fresh context separate from the builder's.

Chapter 17 supplied a trace that distinguishes the assessed revision, permission and actual outcome. Delegating the maintenance proposal must preserve those references and the receiving agent's authority boundary. A concise brief can omit unrelated conversation while retaining the unknown-owner exception and current execution limits.

Why Spawn Sub-Agents at All

An orchestrating agent spawns sub-agents for three reasons: to work in parallel on independent parts of a task; to give a sub-task a fresh context that is not polluted by everything the orchestrator has accumulated; and to delegate work to a cheaper or more specialised model. The three patterns that follow from those reasons are sequential (A analyses, B implements from A's result), parallel (A, B and C work at once and the results are merged), and hierarchical (a manager agent spawns and coordinates specialists). Claude Code exposes all three through its sub-agent facility; the patterns predate

it and apply to any runtime that can start an agent with a chosen context.

When spawning a sub-agent, decide which context it needs. Choosing by capability uses the information in Chapter 12's skill catalogue: what each skill can do and roughly what it costs. Passing the orchestrator's whole context consumes the sub-agent's budget and carries accumulated drift into its new session. This chapter uses a minimal brief instead: the task, relevant files, constraints and a short summary of prior work, within a token ceiling (the author uses about 8,000).

Memory Across Turns

Memory for an agent that works across turns comes in four kinds, distinguished by how long each lives:

Kind	Holds	Lives for
Working	The current turn, inside the budget of Chapter 10	One turn
Session	Recent turns, the author keeps the last fifty	One session
Long-term	Facts and learnings that survived a review	Indefinitely
Shared	What every agent in an orchestration may see	One orchestration

The two operations that matter are promotion and decay. Working memory is promoted to session memory at the end of each turn, and

session memory is trimmed to its window. Retrieval from session memory decays with age: a turn older than about ten is only retrieved if it is strongly relevant to the current query, which is a crude form of the compression priority in Chapter 11. Long-term memory is written deliberately, by summarising a session into its key decisions and learned patterns, rather than by promoting everything, because a long-term store that holds everything is a session log with a misleading name. Chapter 20 shows the same split in a running system.

Isolation, Sharing, and the Hybrid

Figure 18.1 shows the topology the tradeoffs in this section are about.

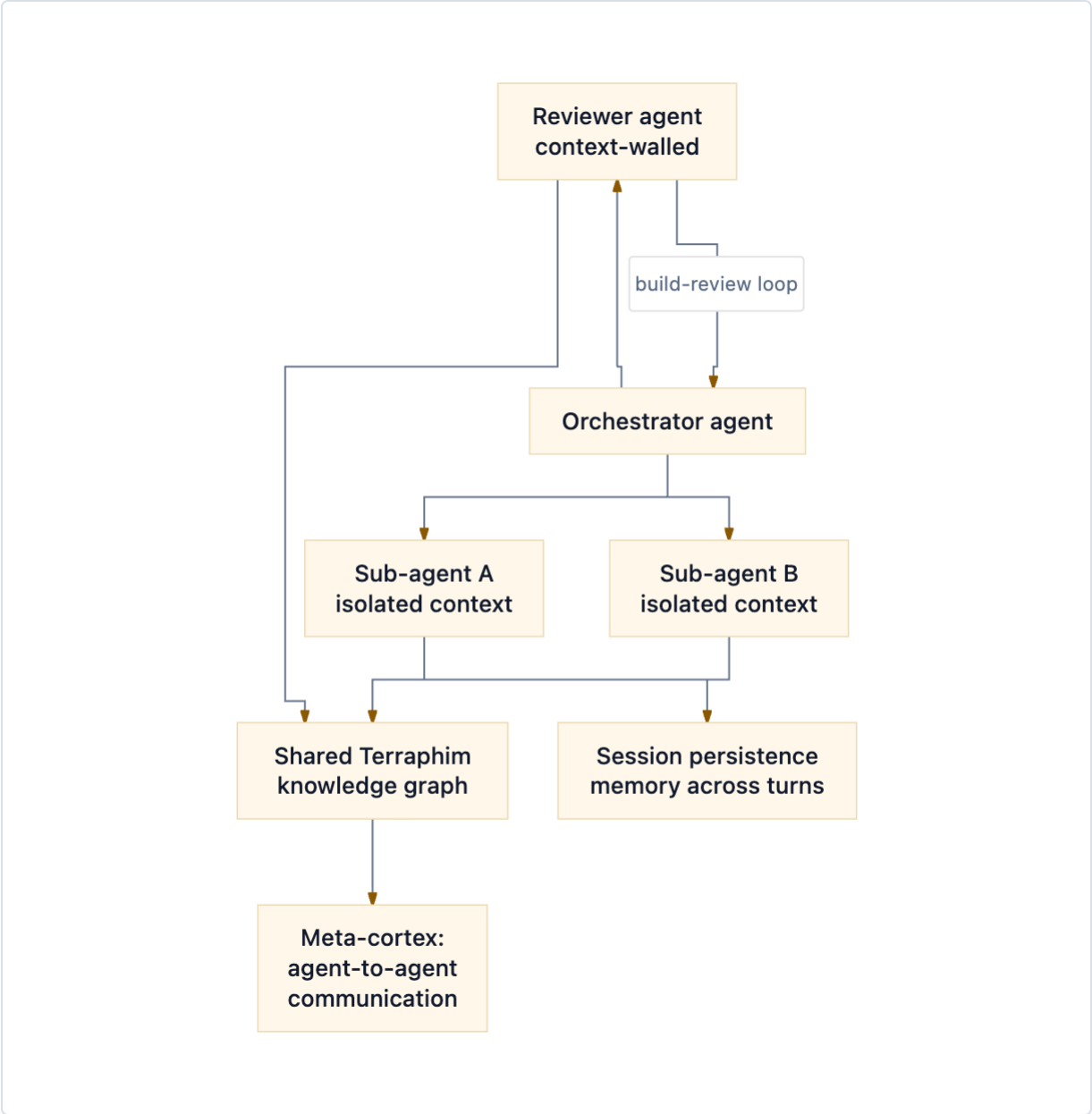


Figure 18.1: Isolated sub-agent contexts sharing one knowledge graph and one persistence layer, with a context-walled reviewer closing the build-review loop.

When several agents work together, each can have its own context, or all can see one shared context, or something in between.

	Isolated	Shared	Hybrid

	Isolated	Shared	Hybrid
What each agent sees	Only its own files and brief	Everything every agent has	A small shared core plus its own files
Cross-contamination	None	Likely	Contained to the core
Token cost	Duplicated per agent	One copy, but large	Small shared core, modest per-agent
Coherence	Hard: agents cannot see each other's decisions	Natural	Through the shared decision log
Debugging	Simple per agent	Hard: who changed what?	Moderate

The author's hybrid design shares the knowledge graph, the current task and its constraints, and an append-only decision log. Other context stays with each agent. An agent that decides to use JWT tokens records that decision in the log; other agents can then follow it without loading the files and conversation that led to it.

The Knowledge Graph as Shared Backbone

The author's hybrid design uses the Terraphim knowledge graph of Chapters 5 to 9 as its shared vocabulary and reference. Agents query it for context, with each query logged under the agent's identity to extend Chapter 9's audit trail. Before handing off, they check the entities and relationships in their results against the graph; this can expose an invented component or API when the extractor recognises

it. When an agent needs an absent term, it proposes an addition through Chapter 8's curation process.

Unrestricted writes would allow unsupported additions into the reference used by every agent. Curation must therefore apply to agent-proposed changes as well as human edits.

Build-Review Loops and the Context Wall

Hrishi of Anthropic (2025) describes the **build-review codon loop** to address favourable self-review: the builder's accumulated context can influence its assessment of its own work. A separate reviewer receives only the output across a context wall. That separation addresses the shared-context problem, while the reviewer's judgement still needs the calibration described in Chapter 14.

A codon is a sealed step with its own fresh context. The basic form is three codons, build, review, update, each starting clean and receiving only the previous step's output. The example below is the four-codon variant the author runs, which splits the work into a plan and a build so that each can have the critic its size calls for:

```
const buildReviewLoop = {
  iterations: 3,
  codons: [
    { id: "plan",        model: "sonnet", contextMode: "fresh" },
    { id: "review-plan", model: "haiku",  contextMode: "fresh" },   //
small critic for plans
    { id: "build",       model: "sonnet", contextMode: "fresh" },
    { id: "review-code", model: "opus",   contextMode: "fresh" }    //
large critic for code
```

```
  ]
};
```

The review codon is an LLM-as-judge in the sense of Chapter 14, and everything that chapter says about calibration applies: a reviewer that has never been measured against labelled examples is a reviewer whose kappa is unknown, however fresh its context. The model choices in the example follow a finding from the same work that is counter-intuitive enough to state carefully: **smaller models make better critics of plans, and larger models make better critics of code.** A small model reviewing a plan surfaces the problems a large model would quietly compensate for; a critic that fixes problems in passing hides them from the record. A large model reviewing code is better because it must build an understanding of the codebase from a fresh context and locate a precise defect, which the small model cannot do. The example above applies that asymmetry; an earlier draft of this chapter had a small model reviewing code, which contradicted the finding it was illustrating.

Automated review surfaces

Move repeatable checks out of the agent's review workload. A script can run tests between build and review codons, extract the diff and collect coverage, linting and type-check results. Prithvi of Anthropic (2025) reports that this pattern, called automated review surfaces, reduced the cost of a comic-generation pipeline by about ten times while improving quality; the figure is Anthropic's own report of its own pipeline and has not been independently reproduced here.

Inspect the loop for other steps tooling can perform without model judgement.

Tooling handles repeatable checks between codons, while agents review questions that need judgement. Measure the effect of additional iterations on the task set before deciding how many the loop should run.

The Meta-Cortex: Agents Talking to Each Other

The **meta-cortex** is a shared communication layer through which agents publish findings, requests, decisions and confirmations, and subscribe to the types they care about. Subscription is matching on the `type` field: an agent registers the types it wants, and the layer delivers matching messages into a per-agent queue that the agent drains at the start of its next turn, so delivery is at turn boundaries rather than mid-reasoning. A message is small and typed:

```
{"from": "agent-b", "type": "decision", "at": "2026-10-08T09:41:12Z",
 "content": "Use JWT with 1-hour expiry; sessions rejected for the
API",
 "refs": ["decision-log#D-042"]}
```

An agent that finds a memory leak publishes a `finding`; an agent that needs a schema publishes a `request`; an agent that chooses JWT over sessions publishes a `decision` like the one above, and any agent subscribed to decisions receives it on its next turn. Every message is

logged with its sender and time, so the layer doubles as the audit trail of the orchestration.

The communication layer lets agents coordinate hand-offs through shared messages and a record. Its resilience depends on the implementation: an in-memory bus inside one process has a single point of failure, while a durable queue adds storage, recovery and operational work. The author's implementation uses the in-memory bus.

Where Multi-Agent Isolation Fails

Three costs are worth stating against the pattern's benefits.

Context walls are expensive. Every fresh codon rebuilds its understanding from nothing. A loop of three iterations over four codons is twelve fresh contexts, each loading the constitutional tier, the brief and the relevant files. For a small task the review loop can cost more than the work.

Merging parallel results is where errors hide. Three sub-agents working on independent parts produce three results that were each consistent with their own context and are not consistent with each other. The merge step needs the same validation as any other output (Chapter 14), and it is the step most often written as a concatenation. The resolution pattern the author uses is to treat the merge as a fourth codon: a fresh-context agent receives only the three results and the shared decision log, is asked to list every conflict between

them before producing anything, and hands the conflict list to a person if it is non-empty. A merge that reports zero conflicts is validated like any other output; a merge that resolves conflicts silently is the failure mode.

The shared core drifts if anyone can write to it. The decision log and the knowledge graph are only stable if writes to them are curated. An orchestration that lets every sub-agent append to the graph has rebuilt the shared-context pollution problem one layer down.

Durable Handoffs Between Sessions

A bounded brief works while the recipient is active. Across sessions, the sender also needs to record who owns the next action, when it is due and how the recipient acknowledges it. The stakeholder is the coordinator, concerned that a pending commitment will be lost or acted on after its assumptions expire. An illustrative requirement is: **the handoff service shall expose each commitment's owner, freshness and acknowledgement state before resuming work**.

The author's Continuity Loop note describes a `memory/handoffs/PENDING.yaml` buffer loaded at session start only when it is less than 24 hours old, alongside permanent daily ledgers. Study E6 (Appendix I) summarises the author-reported pattern. Expiry bounds automatic reuse; it does not guarantee automatic resumption weeks later. After a longer absence, the receiving agent

needs an explicit ledger review and renewed confirmation of relevance and authority.

At session end the agent writes the handoff file; at session start a new instance loads it automatically when it is younger than the expiry window, then archives it to a permanent ledger. A later recipient can review the written state after a longer absence, but must reconfirm expired assumptions and authority before resuming work.

```
# memory/handoffs/PENDING.yaml, illustrative structure from the
account
created_utc: "2026-10-08T17:30:00Z"
expires_utc: "2026-10-09T17:30:00Z"   # 24-hour window: zombie
prevention
context:
  executive_summary: "..."
  next_actions:
    - "Push the reviewed change after approval"
    - "Re-index memory search (embedding provider changed)"
  open_loops:
    - { item: "Sibling-agent naming", state: "awaiting-human" }
```

```
# Illustrative handoff contract
handoff_id: maintenance-017
owner: maintenance-reviewer
requirement_id: R-CHANGE-01
status: awaiting_acknowledgement
created_utc: "2026-10-08T12:00:00Z"
expires_utc: "2026-10-09T12:00:00Z"
evidence_refs: [case-report-017]
next_action: "Review the candidate against the current policy"
on_expiry: review_ledger_and_reconfirm
```

Independent agents can use a designated shared surface while keeping their workspaces private. The contract must define permitted writers, acknowledgement, duplicate handling and the action taken after expiry. A file alone supplies none of those guarantees. Write the handoff so a fresh recipient can identify the requirement, evidence and next action without reconstructing the sender's conversation.

A sibling-agent deployment in the same workspace specified the contract explicitly:

- **Isolated workspaces.** Each agent has its own workspace and memory; neither reads the other's private state.
- **One shared surface.** A designated directory holds handoff files and nothing else — no ambient coupling through shared configuration or logs.
- **Structured schema.** Status, artefacts produced, next actions and expiry. A handoff is a promise with a deadline; unbounded shared state has no owner.
- **Write for a stranger.** The receiving agent may be a fresh instance with zero conversation history. If a handoff assumes the context of the chat that produced it, it has failed.

Knowledge graphs can represent commitments as well as facts. The useful distinction is operational: the handoff contract gives a commitment an owner and lifecycle, while the graph can supply its supporting entities and evidence. Chapter 17 links parent and child decision traces; a handoff reference preserves that connection

without granting new execution authority. Figure 18.2 includes freshness and acknowledgement before memory promotion.

A durable inbox gives pending work a lifecycle that a transcript alone does not supply. Record the owner, current evidence and deadline for a decision that must survive a session, and require acknowledgement or renewed review when the handoff expires.

Summary

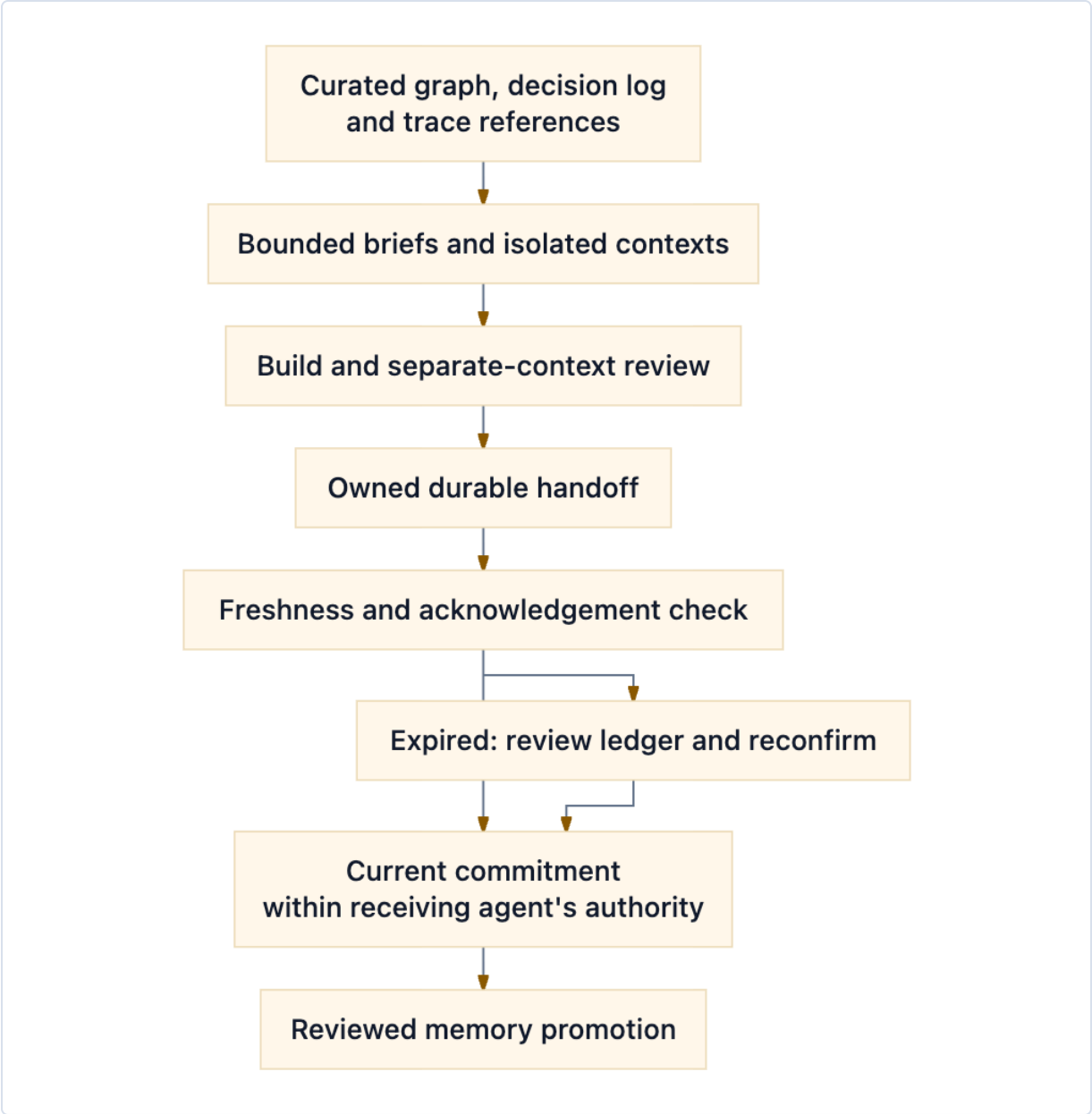


Figure 18.2: Chapter 18 recap: Bounded agent contexts connect through reviewed results and durable handoffs with ownership, freshness and acknowledgement.

Multi-agent systems require explicit context management:

1. Sub-agents receive a minimal brief with a token ceiling, not the orchestrator's context

2. Memory has four lifetimes, with deliberate promotion to long-term rather than wholesale copying
3. The hybrid topology shares a knowledge graph, the task and a decision log, and isolates everything else
4. The knowledge graph is the shared backbone, written to only through curation
5. Build-review loops separate reviewer context from builder context, with small critics for plans and large critics for code in the cited pattern
6. Automated review surfaces take agents out of reviews that need no judgement
7. A meta-cortex gives coordination and an audit trail, not resilience, unless it is built as a durable queue

These patterns let a system scale past one agent while keeping the coherence that Chapters 4 and 8 built.

Cross-session commitments need an owner, freshness checks, acknowledgement and an explicit route after expiry.

The result passed to execution is an owned, current handoff with bounded scope and recoverable evidence. Expiry calls for review or reconfirmation; it does not grant another agent permission to improvise. Chapter 19 provides runtime checks for the rules and mutations that must remain visible through that handoff.

Source for the added operational case: Experiment E6 (Appendix I), the Continuity Loop handoff pattern and the sibling-agent contract,

8 October 2026; the observations have not been independently reproduced for this edition.

Key Sources

- Anthropic, Claude Code documentation on sub-agents
- Hrishi (Anthropic), “Agentic Programming Patterns: Build-Review Codon Loops” (2025): the context wall and the critic-size asymmetry
- Prithvi (Anthropic), on the generator-evaluator pattern and automated review surfaces (2025): the tenfold cost figure is Anthropic’s report of its own pipeline
- Terraphim knowledge graph implementation (github.com/terraphim/terraphim-ai)
- The author’s orchestration code for the hybrid topology, decision log and in-memory meta-cortex
- Chapters 8, 9, 10, 11, 14 and 20 of this book

Chapter 19: Hooks and Automation for Context

Chapter 18 showed how multi-agent systems coordinate. **Hooks** are functions the runtime calls at fixed points in an agent's execution: around model calls and tool use, or at session start. They can enforce checkable rules from CLAUDE.md (Chapter 13) by inspecting inputs, refusing an operation or requiring a check before it proceeds. Enforcement depends on the runtime honouring the hook's result.

Hooks can also change what the model sees. A silent prompt rewrite changes the model's context without a reviewable account of the edit. Record those mutations and define what the runtime does when a hook cannot complete its check.

Where Hooks Fire

Figure 19.1 shows where each hook fires relative to the tool call.

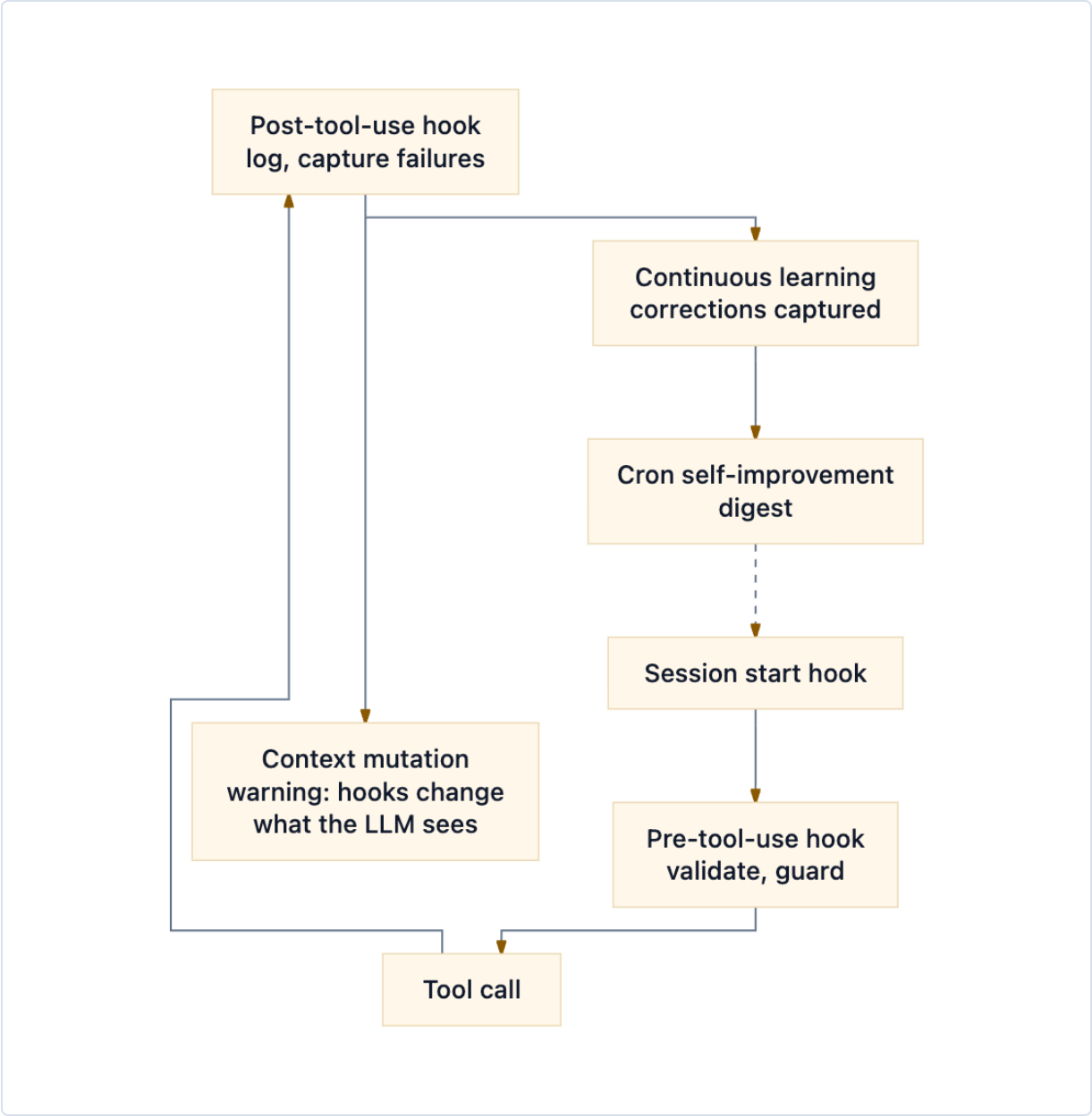


Figure 19.1: Hook lifecycle around a tool call, and the learning loop that feeds captured corrections into a scheduled digest.

Five hook points cover nearly every runtime:

Hook point	Runs when	Typical use
Session start	A session begins	Load constitutional context, check the environment

Hook point	Runs when	Typical use
Before model call	Each time a prompt is about to be sent	Check the token budget (Chapter 10) and compress if needed; retrieve and attach relevant context; validate the input
After model call	Each time a response returns	Validate the output; log the exchange; update session memory
Before tool use	Each time the agent is about to run a tool	Validate the arguments; check permissions; refuse protected operations
After tool use	Each time a tool returns	Validate or correct the result; redact sensitive data; log; capture failures

The before-model hook is where the budget of Chapter 10 is enforced in practice: it measures the assembled context, runs the compression pipeline of Chapter 11 if the budget is exceeded, and only then lets the call proceed. The after-tool hook is where the learning loop of Chapter 23 begins: a failed command is captured with its context so that a later process can decide whether it was a one-off or a pattern.

A runtime exposes these as a pipeline: an ordered list of before-hooks that each receive and return the context, and an ordered list of after-hooks that each receive and return the response. Order matters. A validation hook that runs after an augmentation hook validates the augmented prompt; the other way round, it validates what the user wrote and misses what the hook added.

Tool-Use Hooks: Where Enforcement Lives

A before-tool hook can refuse an operation before it runs. The author uses this pattern for file edits:

```
async function preEditFileHook(call) {
  const { path, oldString } = call.args;
  if (!await exists(path)) throw new Error(`File not found: ${path}`);
  if ([".env", "secrets.yaml", "credentials.json"].some(p =>
path.endsWith(p)))
    throw new Error(`Protected file: ${path}`);
  if (!(await readFile(path)).includes(oldString))
    throw new Error("oldString does not match file content");
  return call;
}
```

These checks reject missing files, edits to listed protected files and replacements whose original text does not match. They enforce those restrictions even if the agent's proposed edit conflicts with its instructions.

In Claude Code these hooks are configured in the settings file as matchers on tool names with a command to run; the command's exit status decides whether the tool call proceeds. The author's configuration pairs a before-tool hook that guards Git operations and rewrites known-bad commands with an after-tool hook that captures failed commands for the learning store, the two hooks in Figure 19.1. The exact configuration syntax changes between releases and is in the documentation listed under Key Sources.

The Context Mutation Problem

A hook that changes the prompt before the model sees it has performed a **context mutation**: the model reasons about something the user did not write and the model does not know was changed. The pattern is useful (a hook that replaces `npm` with `bun` in every command enforces a toolchain policy without a rule the agent might forget) and it is dangerous for three reasons.

A hook that rewrites file paths can make the model refer to a file it never actually read. An augmentation hook that attaches retrieved context can push the assembled prompt over budget silently, so that the model loses the end of its own conversation to compaction without anyone having decided that. And a correction hook that fixes outputs before they are returned hides a systematic problem behind a patch: the model keeps making the error, the hook keeps fixing it, and nobody sees the error rate.

The discipline that keeps mutation safe is that every mutation is recorded on the context it changed:

```
function preHook(context) {
  const mutations = [];
  if (context.prompt.includes("npm")) {
    context.prompt = context.prompt.replace("npm", "bun");
    mutations.push({ type: "text_replace", from: "npm", to: "bun" });
  }
  context.metadata.mutations = mutations;   // the audit trail travels
with the context
  return context;
}
```

Without the `mutations` array, the record omits the change from `npm` to `bun`. Inspect hook records when a command or prompt differs from the agent's proposal.

Learning From Interactions: Two Cadences

Hooks are where continuous learning starts, and the design question is when the learning runs.

	Real-time learning hook	**Scheduled digest**
Runs	After every model call or tool failure	On a timer, outside interactive sessions
Captures	One interaction: what failed, what was corrected	A period: patterns across many captures, plus external sources
Latency cost	Added to every interaction	None during sessions
Quality of what is learned	Noisy: one event is not a pattern	Curated: the digest reads many events and filters
Auditability	A stream of entries	A dated artefact per run

The author's practice uses both. The after-tool hook records failures and corrections in a learning store during the interaction. A scheduled process, the daily self-improvement digest in the Terraphim tooling, reads the store, deduplicates against what it has already processed, scans a tiered set of external sources, and produces a short dated artefact: three to five items with their relevance to the current configuration, one suggested experiment,

and concrete setup changes to consider. Chapter 23 describes the review of these proposals before any rule changes.

Extracting a reusable pattern during every interaction would add model-call cost and latency and risk treating isolated incidents as patterns. Captured observations give the digest evidence to compare across interactions.

A Taxonomy for Hooks

A system with more than a handful of hooks needs them organised, or the order and purpose of each is lost. Five categories cover the ones the author runs:

Category	Purpose	Examples
Validation	Check inputs and outputs	Argument validation, output schema checks
Security	Protect the system	Permission checks, protected files, redaction
Logging	Record what happened	Audit log, metrics, failure capture
Transformation	Change data in flight	Augmentation, policy rewrites, correction
Learning	Improve the system over time	Failure capture, pattern extraction

Each hook carries an identifier, a category, the point it fires at, a priority that fixes its order relative to others at the same point, and an enabled flag. The priority rule the author uses: validation first,

security next, transformation after both (so that what is transformed has been validated and permitted), logging after transformation (so that the log shows what was actually sent), learning last. Transformation hooks are the ones that need a `mutations` record; the taxonomy makes it obvious which hooks those are.

Where Hooks Go Wrong

Three failure modes recur.

Silent mutation, described above: the fix is the audit record, and a periodic read of it.

Hooks that fail open. A hook that throws on an unexpected condition and is caught by a runtime that proceeds anyway has enforced nothing. The author learned this from a pre-tool hook whose rewrite of a command caused the runtime to auto-approve the rewritten command instead of asking; the fix was to have the hook return an explicit “ask” decision rather than a modified command. Chapter 14’s rule applies: an enforcement step that cannot decide must block.

Latency creep. Each real-time hook adds time to every interaction, and a system that accretes hooks for a year is slow for reasons nobody can name. The remedy is the two-cadence split: anything that does not have to happen before the next turn goes to the digest.

An approval applies to an identified artefact and governing policy revision. A hook that merely prints a verdict supplies advisory evidence; enforcement requires the action path to check that evidence and withhold the action when it is missing or stale. Chapter 16 specifies this admission boundary and Chapter 17 records its outcome.

Summary



Figure 19.2: Chapter 19 recap: Hooks enforce rules at runtime, record context changes and feed a curated learning loop.

Hooks are the automation and enforcement layer of context engineering:

1. Five hook points around the model call and the tool call cover nearly every runtime

2. Before-tool hooks are where rules become enforceable, because they can refuse
3. Every context mutation is recorded on the context it changed
4. Real-time hooks capture; a scheduled digest analyses; the split keeps interactive sessions fast and learning curated
5. A five-category taxonomy with explicit priorities keeps hook order deliberate
6. Hooks fail by mutating silently, by failing open, and by accreting latency; each has a specific remedy

Hooks enforce checkable rules from Chapter 13 and supply observations to the learning loop in Chapter 23.

For the maintenance action, a before-tool check can enforce the current scope and quota only if the runtime supplies reliable state and honours refusal. Retain the check result and any context mutation for the trace. Chapter 20 examines composition in the author's OpenClaw system, where some loading rules remain instructions rather than enforced controls.

Key Sources

- Anthropic, Claude Code documentation: hooks (settings-file matchers, PreToolUse and PostToolUse events; read October 2026)
- Model Context Protocol specification (modelcontextprotocol.io, 2025 revision)

- The author’s hook configuration: Git safety guard and knowledge-graph rewrite (before tool), failure capture into the learning store (after tool), and the Terraphim self-improvement digest skill that reads it
- Chapters 10, 11, 13, 14 and 23 of this book

Chapter 20: The OpenClaw Context System

OpenClaw is the personal AI assistant system built and operated by the author. This chapter shows how it combines the budgets, memory, skills and hooks described in Parts III and IV, and identifies where its implementation relies on instructions that the runtime does not enforce.

Separately loaded tiers let the author control when identity, memory and skills enter the context, and inspect the cost and exposure of each tier. The loading rule needs an enforcement mechanism. OpenClaw relies on instructions for most of these rules; the sections below distinguish those conventions from controls implemented in its loader.

The synthetic maintenance case has reached a proposed operational design: relevant context, assessment, permission and a trace. This chapter uses the author's separate, real system to examine what composition can leave unenforced. Its scheduled-work examples help distinguish successful invocation from completion and delivery, the distinction behind R-USE-01. They do not demonstrate that the synthetic repair architecture has been deployed or that its shared-resource limits are enforced.

Architecture Overview

OpenClaw's workspace directory groups context by loading rule. Each top-level entry below corresponds to a context tier or an operational component.

Figure 20.1 groups the workspace entries by their loading rules.

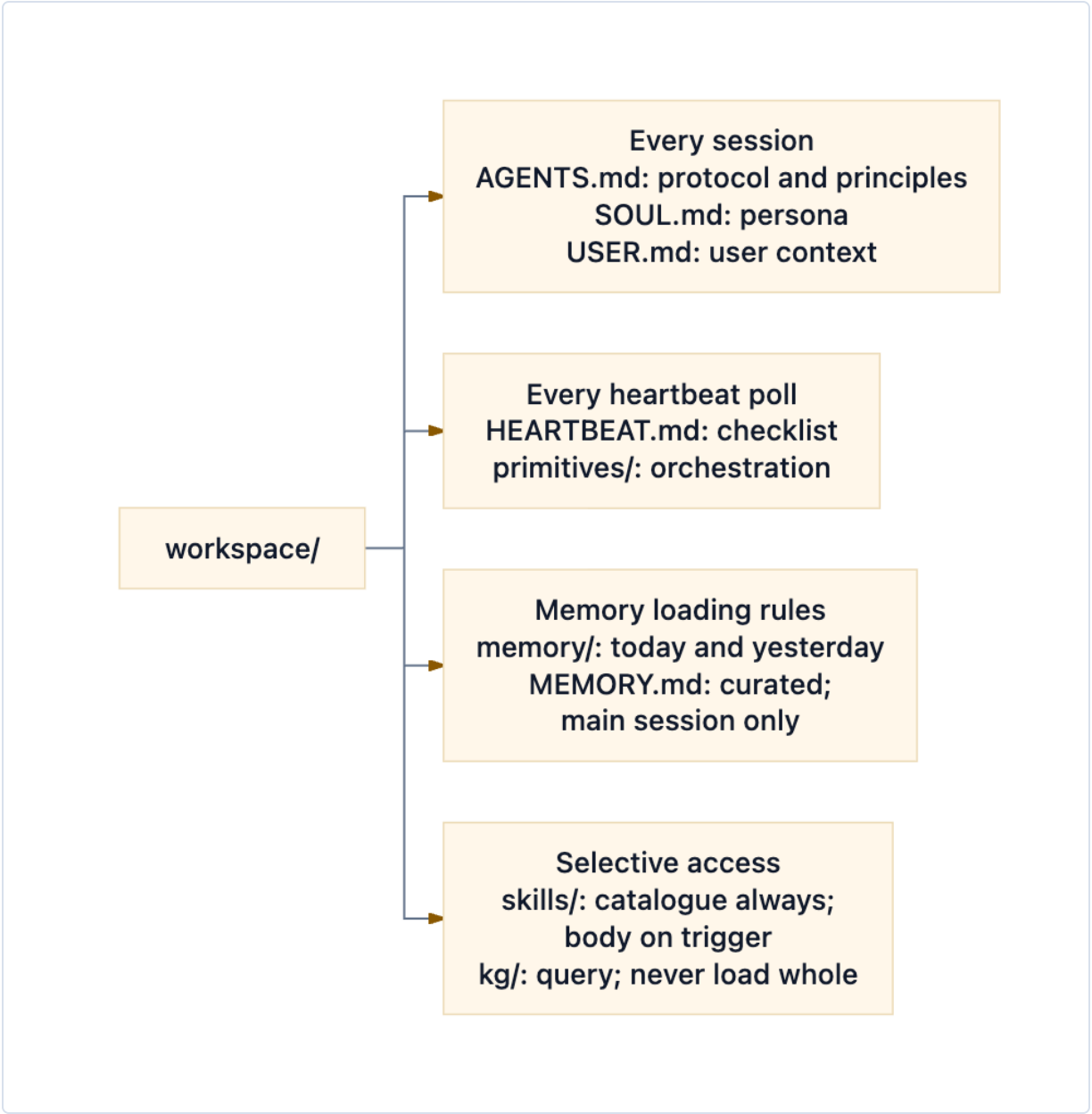


Figure 20.1: OpenClaw groups session files, heartbeat inputs, memory and selectively accessed capabilities by loading rule. The diagram describes workspace conventions, rather than runtime enforcement guarantees.

The three files read at the start of every session are the “never compressed” layer of Chapter 10. Together they run to about 2,700 words, which is the one place where OpenClaw spends generously: the persona and the session protocol are the content whose loss between sessions is most visible, so they are the content guaranteed to be present. HEARTBEAT.md is loaded on every autonomous poll, and the file carries its own instruction at the top, “keep this file small: heartbeat context is loaded every poll”, which is the token budget of Chapter 10 enforced by a comment rather than by a limit. Everything else is loaded on demand.

Figure 20.2 shows the four subsystems and how the heartbeat loop ties them together.

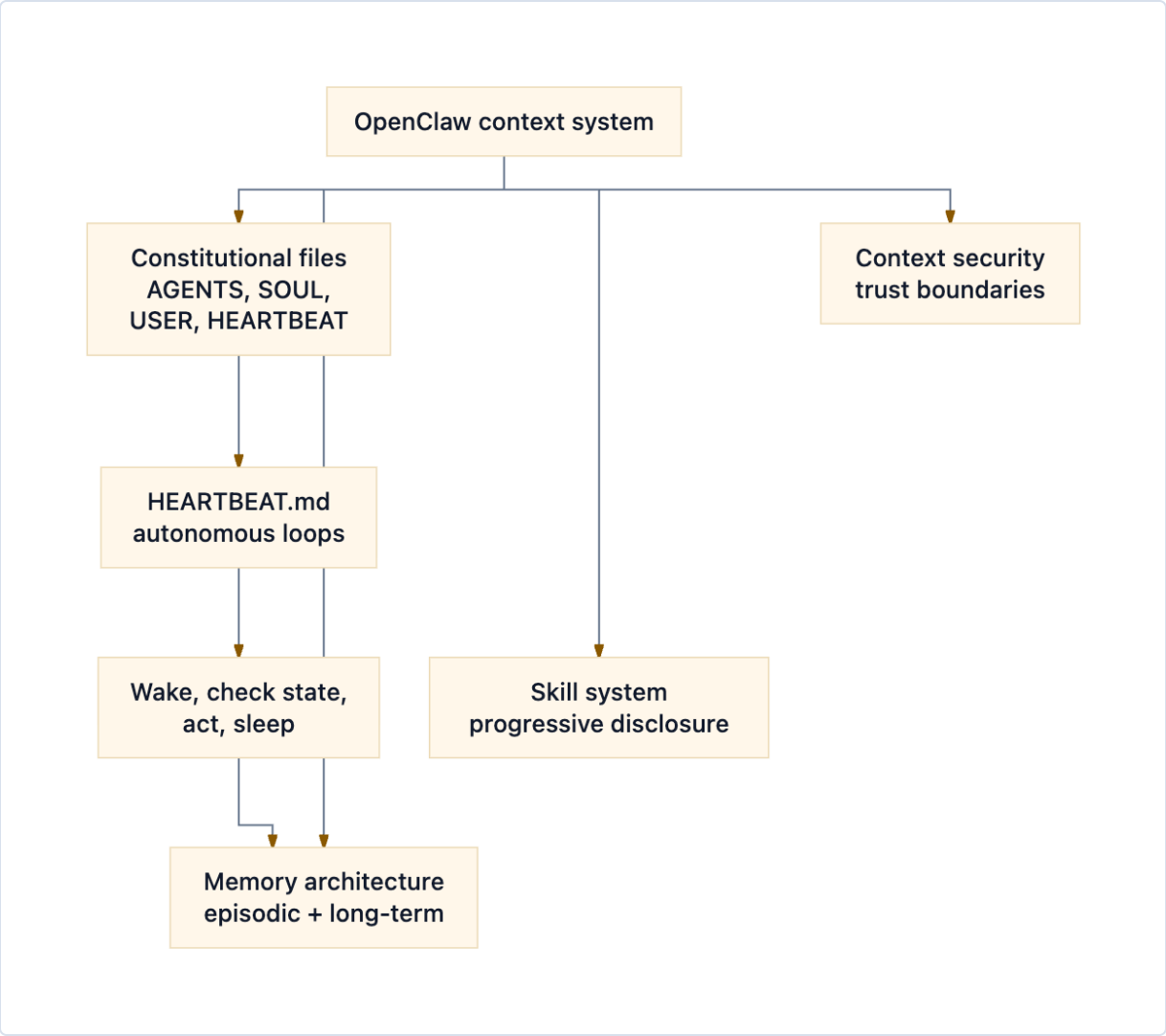


Figure 20.2: OpenClaw's context system: constitutional files, memory, skills and security boundaries, with the heartbeat loop driving autonomous operation.

Memory Architecture

OpenClaw keeps four kinds of memory, distinguished by lifetime rather than by content.

```
const memory = {
  constitutional: ["AGENTS.md", "SOUL.md", "USER.md"],  // read every
session
```

```
  daily: ["memory/<today>.md", "memory/<yesterday>.md"], // raw log,
two days loaded
  longTerm: "MEMORY.md",                                  // curated,
main session only
  shortTerm: {}                                           //
ephemeral, cleared each session
};
```

Daily notes and the curated long-term file serve different purposes. Daily notes record events in order during the session; loading only two days at start bounds their cost. MEMORY.md retains facts selected through review. This applies Chapter 11's compression priority when writing memory: the raw log can remain long because it is not loaded whole, while curation limits the content retained in the long-term file.

Memory Loading Rules

The session protocol lives in AGENTS.md and is read by the model at the start of every session:

```
## Every Session
1. Read SOUL.md          -- this is who you are
2. Read USER.md          -- this is who you're helping
3. Read memory/<today>.md and memory/<yesterday>.md
4. If in MAIN SESSION (direct chat with your human): also read
MEMORY.md
```

Rule 4 is the chapter's clearest example of a safety property, and of the limit of enforcing it by instruction. MEMORY.md is the index of personal context. It is meant to be read only in a direct session with the author and never in a session that a group chat or a second

person can see. In the current system that rule is a sentence in AGENTS.md, which means it is context: the model follows it because it has read it, and anything that pushes AGENTS.md out of the window, overrides it, or argues around it removes the protection. The stronger design, which OpenClaw does not yet implement, builds the session context from a rule table in the loader and gives the loader no code path that reads a main-session-only file in a shared session. The difference between the two is the difference between a rule that is context and a rule that is architecture, and it is the single most important upgrade on the system's list.

Skill System

Skills follow the catalogue-then-body pattern of Chapter 12. The catalogue is a few hundred tokens listing every skill with its triggers and tier; the body of a skill is loaded only when a trigger matches.

```
skills:
  - name: github
    triggers: ["gh", "github", "pr", "issue"]
    tier: core
  - name: gog
    triggers: ["gmail", "calendar", "email"]
    tier: core
  - name: weather
    triggers: ["weather", "forecast", "temperature"]
    tier: optional
```

Skill Activation

```
async function activateSkills(query) {
  const triggered = skills.filter(s => s.triggers.some(t =>
query.includes(t)));
  const tierOrder = { core: 0, standard: 1, optional: 2 };
  triggered.sort((a, b) => (tierOrder[a.tier] ?? 2) -
(tierOrder[b.tier] ?? 2));
  return triggered.slice(0, 5);   // at most five active skills
}
```

Two numbers in this code are choices rather than facts, and both are contestable. The cap of five active skills comes from the budget arithmetic of Chapter 12: five full skill bodies at a few hundred to a thousand tokens each is the most that fits beside the constitutional tier and a useful retrieval allowance inside a 12,000-token working budget. A system with a larger budget could raise it; a system that raised it without raising the budget would push its constitution towards the compaction threshold, the fill level at which the runtime starts summarising older context, which OpenClaw sets at 90 per cent of the window. The tier ordering means that when more than five skills trigger, the optional ones are dropped first, which encodes the judgement that a wrong answer about the weather costs less than a wrong answer about a pull request.

The weakness of trigger matching by substring is that it is the deterministic search of Chapter 6 with none of its vocabulary discipline: “pr” triggers the GitHub skill inside the word “prepare”. OpenClaw tolerates this because the cost of a wrongly activated skill is a few hundred tokens, not a wrong action. A system where activation had side effects would need the thesaurus-backed matching of Chapter 7 instead.

HEARTBEAT.md: Autonomous Loops

The runtime reads HEARTBEAT.md every hour to start unattended work. The file lists items to check or act on, with completed items ticked and dated so that the next poll can see what has already happened.

```
# HEARTBEAT.md
# Keep this file SMALL: heartbeat context is loaded every poll.

- [x] 2026-09-12: service de-scope complete, verified, evidence
recorded
- [ ] If a guardian check FAILS three polls in a row, page: "3rd
consecutive, needs decision"
- [ ] Expired suppressions: ask once, re-mention weekly at most, do
not spam
```

```
async function heartbeat() {
  const tasks = await scanTasks();
  const actionable = tasks.filter(t => !t.blocked);
  if (actionable.length > 0) {
    await execute(prioritize(actionable)[0]);
  } else if (tasks.some(t => t.blocked)) {
    await notifyHuman(getBlockerQuestion(tasks));
  }
}
```

The hourly interval is the second contestable number. It is the period at which an unattended loop's mistakes are still bounded: a poll that misreads an item and starts the wrong work wastes at most one interval before its ticked-and-dated log is read by the next poll or by the author. Shorter intervals multiply the cost of a misread; longer ones make the system feel absent. The rules in the checklist about

paging (“three consecutive failures, then page”) and about nagging (“re-mention weekly at most”) are the context budget of Chapter 10 applied to the human: a person reading notifications has a budget too, and the loop is written to respect it.

What the heartbeat does not do matters as much as what it does. It does not modify its own checklist’s rules, its skills, or the constitutional files. Changes to those go through the hooks of Chapter 19 and the judge gate of Chapters 14 and 23, where they are reviewed before they land. An autonomous loop that could edit its own constitution would be the composition failure named at the start of the chapter.

Context Security

OpenClaw distinguishes a main session, a direct conversation with the author, from shared sessions that other people can see, and the difference is which files the session protocol tells the model to read.

```
const sharedContext   = { workspace: true, skills: true, memory:
false, personal: false };
const personalContext = { workspace: true, skills: true, memory: true,
personal: true  };
```

The intended separation applies Chapter 18’s context wall to people: shared sessions should receive only material approved for sharing. The current implementation depends on the model following AGENTS.md. File classification is also maintained by hand, so a personal note filed in the wrong directory can enter a shared session.

The proposed controls are the loader-enforced rule table described above and a filing convention checked by Chapter 19's pre-commit hooks; neither is implemented.

Designed Versus Enforced

The gap the chapter keeps returning to is easiest to see in one table.

Property	Designed by	Enforced today by	Upgrade
Constitution always present	Session protocol, step 1 and 2	The model reading AGENTS.md	Loader prepends the three files unconditionally
Personal memory out of shared sessions	Session protocol, step 4	The model reading a sentence	Loader rule table with no read path for main-session-only files
Heartbeat cannot edit its own rules	Convention	Nothing mechanical	Before-tool hook refusing writes to the constitutional files (Chapter 19)
Skills capped at five	Loader code	The loader	Already enforced
Daily notes limited to two days	Session protocol, step 3	The model	Loader loads exactly two files

What the Composition Shows

Loading rules connect the mechanisms from Parts III and IV. They specify when each context tier enters a session and define the intended protections: retaining the constitution, excluding personal

memory from shared sessions and preventing heartbeat edits to its own rules. Most remain instructions for the model rather than restrictions applied by the loader. The table above identifies which controls need implementation before the system can rely on those protections.

Applying This to Your System

1. List every rule your agent is supposed to follow and mark each as enforced by instruction (the model reads it) or by construction (code makes the alternative impossible). The first column is your risk register.
2. Move the rules that protect people or secrets into the second column first: a loader rule table for what enters a session, a before-tool hook for what the agent may write.
3. Size the constitutional tier deliberately and keep it out of the compression path; measure it with the model's tokeniser rather than guessing.
4. Give any unattended loop an interval, a ticked-and-dated log, and no write access to its own rules.

Scheduled Jobs: Delivery and Its Failure Modes

The operator needs the morning briefing to contain current information and arrive on time. A successful scheduler exit is only one observation about that service. An illustrative requirement is: **the**

delivery service shall verify each required phase, validate the produced content and expose fallback use before reporting successful delivery.

The author's cron reliability experiment (E1, Appendix I) reports 2,055 runs: 1,586 in a legacy registry era and 469 in a later era. Reported success rates are 75.9% and 51.4%; timeouts account for 178 of 228 errors in the stated error subset. These observational cohorts differ in time and job composition. They do not isolate the effect of a redesign, and repeated runs from the same job are not independent systems.

The account's failure taxonomy is useful beyond the aggregate rate. Shared resource exhaustion can make several jobs fail rapidly at the same phase. A job can report an error after delivering useful output, or report success while producing a placeholder. A fallback can deliver the output while hiding a broken primary credential. In a multi-stage pipeline, extraction may fail while consolidation remains healthy, or the reverse. Record phase identity and inspect both status and the expected deliverable to distinguish these conditions.

```
# Illustrative delivery checks
job_contract:
  phases: [extract, consolidate, deliver]
  content_check: required_sections_and_freshness
  fallback_visibility: required
  recovery_evidence: successful_phase_and_delivery_checks
```

Splitting long jobs into shorter phases is a candidate intervention when timeouts dominate. The reported move from eight jobs to

thirteen does not itself prove a reliability gain. Compare equivalent workloads, completion times and content validity before concluding that the extra phases helped. Similarly, two clean cycles can support a local recovery rule but cannot establish durable recovery for every job.

Figure 20.3 connects scheduled phases with status, content and delivery evidence. Chapter 17's trace design explains how the observations can share an identity; Chapter 23 assigns ownership and review cadence to the practice maintaining them.

Preserve the Diagnostic Evidence

The same experiment (E1, Appendix I) reports 93.2% successful runs for a stateless news aggregator and 13.7% for a multi-stage briefing job. It also reports conditional success probabilities of 0.85 after success and 0.10 after failure, with an odds ratio of 52. These are attributed observations, not an independently verified comparison. The transition cohort, counting method and dependence between jobs need inspection before interpreting the odds ratio or transferring it to another fleet. The useful question is whether job identity, shared resources and recent state explain more than an aggregate success rate.

Keep the concrete failure signatures alongside that question:

Reported incident	Evidence to retain	Requirement it helps assess

Reported incident	Evidence to retain	Requirement it helps assess
Disk exhaustion made nine jobs fail rapidly; reported median 1.6 seconds over six days	Shared precondition, error, phase and run timestamps	The delivery service shall expose common resource failures
An error status accompanied an apparently delivered output	Status, content validation and delivery receipt	Success reporting shall reconcile execution and delivery observations
A reported successful run produced a 744-byte placeholder	Required fields and useful content, alongside size	The content service shall reject incomplete deliverables
A secondary voice provider reportedly concealed four days of primary-provider failure	Provider selected, fallback reason and actual environment check	The delivery service shall expose fallback use
Extraction and consolidation failed at different times	Separate phase observations and their input/output identities	The operator shall be able to locate the failed phase

The values describe the experiment's reported incidents; none is a universal alarm threshold. A longer file can still be empty of useful information, and a brief valid output can be small. Validate the content contract rather than replacing scheduler status with file size as the sole success metric. Retain the reported eight-to-thirteen job split as an intervention to evaluate, alongside retries, resource repair and changes to workload. Compare delivery quality and operational cost before treating decomposition as the cause of recovery.

Summary

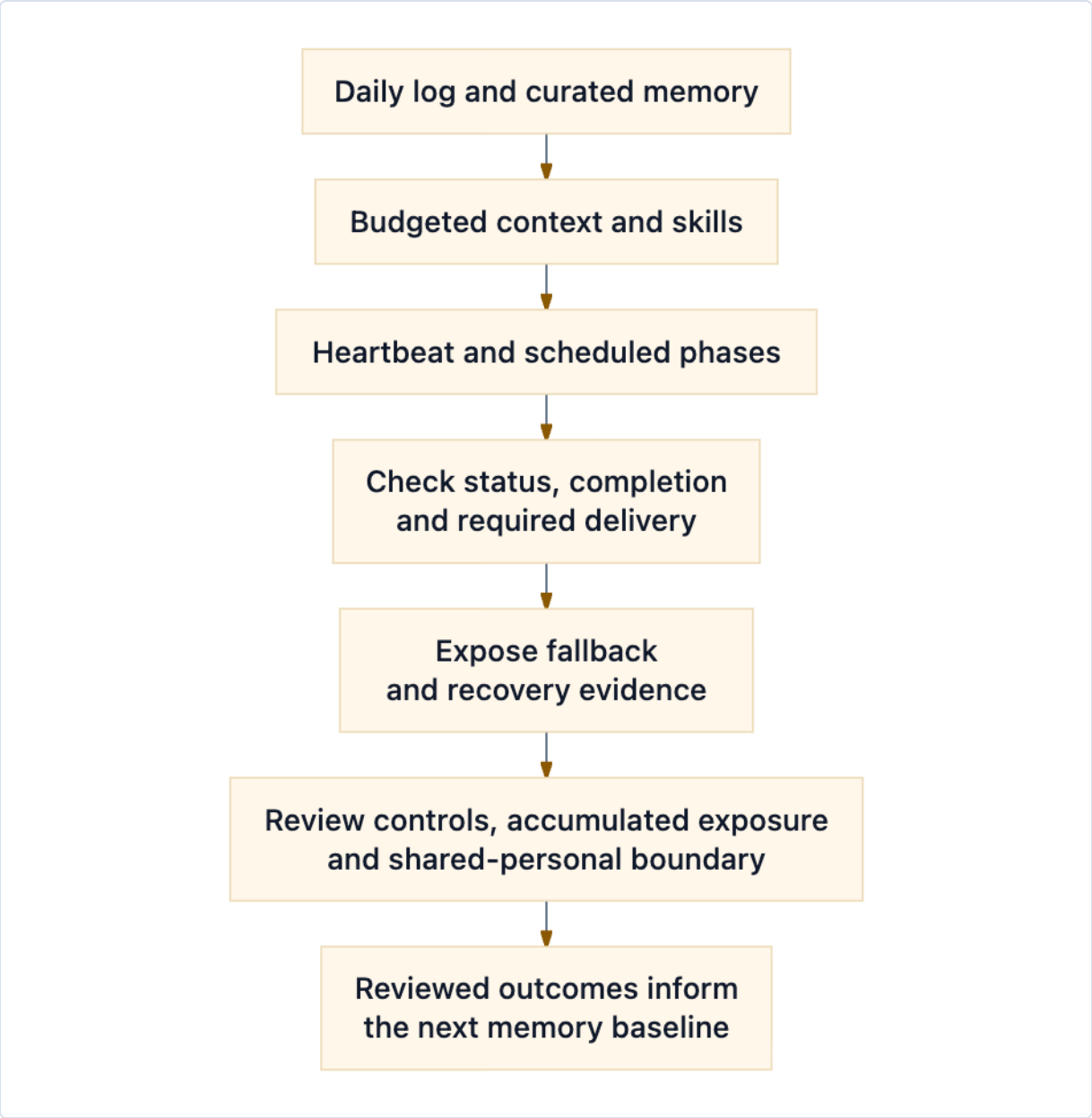


Figure 20.3: Chapter 20 recap: Budgeted context and scheduled phases require content, delivery and fallback evidence before results return to memory.

OpenClaw demonstrates:

- A layered memory architecture with a raw daily log and a curated long-term file, compressed at the point of writing
- Skill modularity with a budget-derived cap and a tier order that encodes what may be dropped
- An hourly autonomous heartbeat with a ticked-and-dated checklist; the interval and checklist do not themselves enforce resource or authority limits
- A shared-versus-personal boundary that is designed by construction and currently enforced by instruction, which is the system's known weakness

Scheduled delivery requires phase status, valid content and visible fallback; scheduler success alone is insufficient.

The evidence handed to governance must identify which controls the runtime enforces and which still rely on instruction. A heartbeat interval alone cannot bound accumulated resource use or loss across repeated runs. Chapter 21 assigns ownership and change authority for those remaining exposures, including consequences that cross team boundaries.

Source for the added operational case: Experiment E1 (Appendix I), the cron reliability study of 2,055 scheduled runs, February to September 2026; the numerical results have not been independently reproduced for this edition.

Key Sources: OpenClaw workspace files (AGENTS.md session protocol, HEARTBEAT.md, runtime configuration with a one-hour

heartbeat and a 0.9 compaction threshold; author’s system, read October 2026); Chapters 10, 11, 12, 18 and 19 of this book for the mechanisms composed here.

Chapter 21: Enterprise Context Governance

When several teams depend on the same vocabulary or domain model, they need an agreed process for changing it. Enterprise context governance defines who may change shared context, which checks the change must pass and what evidence and approval records it needs.

Chapter 4 established the reconciled domain model; Chapter 14 described checks against it. This chapter applies those controls across repositories and teams. Chapter 23 gives a worked example of maintaining them within one repository.

In the synthetic repair case, the quota-sharing team may be affected without owning the assistant or approving its initial request. Impact review must investigate that exposure, identify the relevant service and risk owners, and preserve uncertainty where effects are not yet observed. The requirement may need a revised allocation or a different repair design. Approval by the requesting owner alone cannot establish that the shared consequence is acceptable.

Agent evolution must align with the organisation’s risk model and approved risk appetite. The accountable risk owner shall translate concerns about harm and loss into requirements allocated to policy, execution and monitoring services. Autonomous changes are permitted only in organisational activities where those services

enforce an approved operating boundary: permitted actions, affected data and systems, resource and duration limits, protected behaviour, detection and intervention thresholds, and a route to stop or recover. Assess exposure across a whole run and across users sharing a model, prompt or tool configuration; a per-call limit alone does not contain repeated or correlated failures. The agent shall not widen its own permissions or relax these controls. Where consequences cannot be credibly bounded within the approved appetite, restrict it to proposing changes for accountable review or withhold the capability. Unbounded autonomy creates open-ended loss exposure and may leave liability uncapped; technical containment does not itself establish a legal liability cap. Risk tolerance, accountable ownership and continuing review are also explicit concerns of the NIST AI RMF and the UK government's Orange Book.

Why Enterprise Context Fails Differently

Chapter 1 argued that AI projects fail by context drift rather than by crashes. At enterprise scale the drift has a specific mechanism: reinterpretation at boundaries. Each team receives a specification, a vocabulary or a model from another team and interprets it in the light of its own concerns. The procurement team's "customer" is a billing entity; the support team's "customer" is a person with a problem; the data team's "customer" is a row with a lifetime value. Each interpretation is locally correct, and an AI system fed all three without a reconciled model will produce confident answers that are wrong for everyone.

Chapter 3 described the limits of documentation without reconciliation: definitions can drift when nobody is required to check them against the shared model. An enterprise gate makes that check part of the change process. A change to shared context takes effect after it passes a checkpoint whose criteria were agreed in advance and whose verdict is recorded.

The Gated Lifecycle

Gates work when they are few, named, and attached to stages that everyone already recognises. The following structure has four stages and two gates per stage, the number the organisations the author has worked with could consistently maintain.

```
lifecycle:
  stages:
    - name: discovery
      gates: [domain_model_baseline, vocabulary_approved]
    - name: design
      gates: [architecture_approved, knowledge_graph_baseline]
    - name: development
      gates: [code_quality_passed, validation_tests_passed]
    - name: deployment
      gates: [integration_passed, security_review_passed]
```

Two of these gates are specific to context engineering and are routinely missing from organisations that have the other six. The **vocabulary gate** at discovery asks whether the terms the project will use are already in the shared thesaurus (Chapter 8), and if not, whether the new terms have been reconciled with the existing ones

rather than added beside them. The **knowledge graph baseline** at design asks whether the project's entities, relationships and invariants have been expressed in the graph that the AI systems will retrieve from, and versioned, so that later changes can be diffed. A project that passes architecture review without passing either of these will build an AI system on a private vocabulary, and the drift begins before the first line of code.

Figure 21.1 shows the states a governed artefact moves through. The transitions are the point: an artefact leaves *Published* only through a change request that re-enters *Review*, and it reaches *Deprecated* only when something else supersedes it. There is no transition from *Published* to *Draft*, because an artefact that is being edited in place while others depend on it is the mechanism of drift the whole structure exists to prevent.

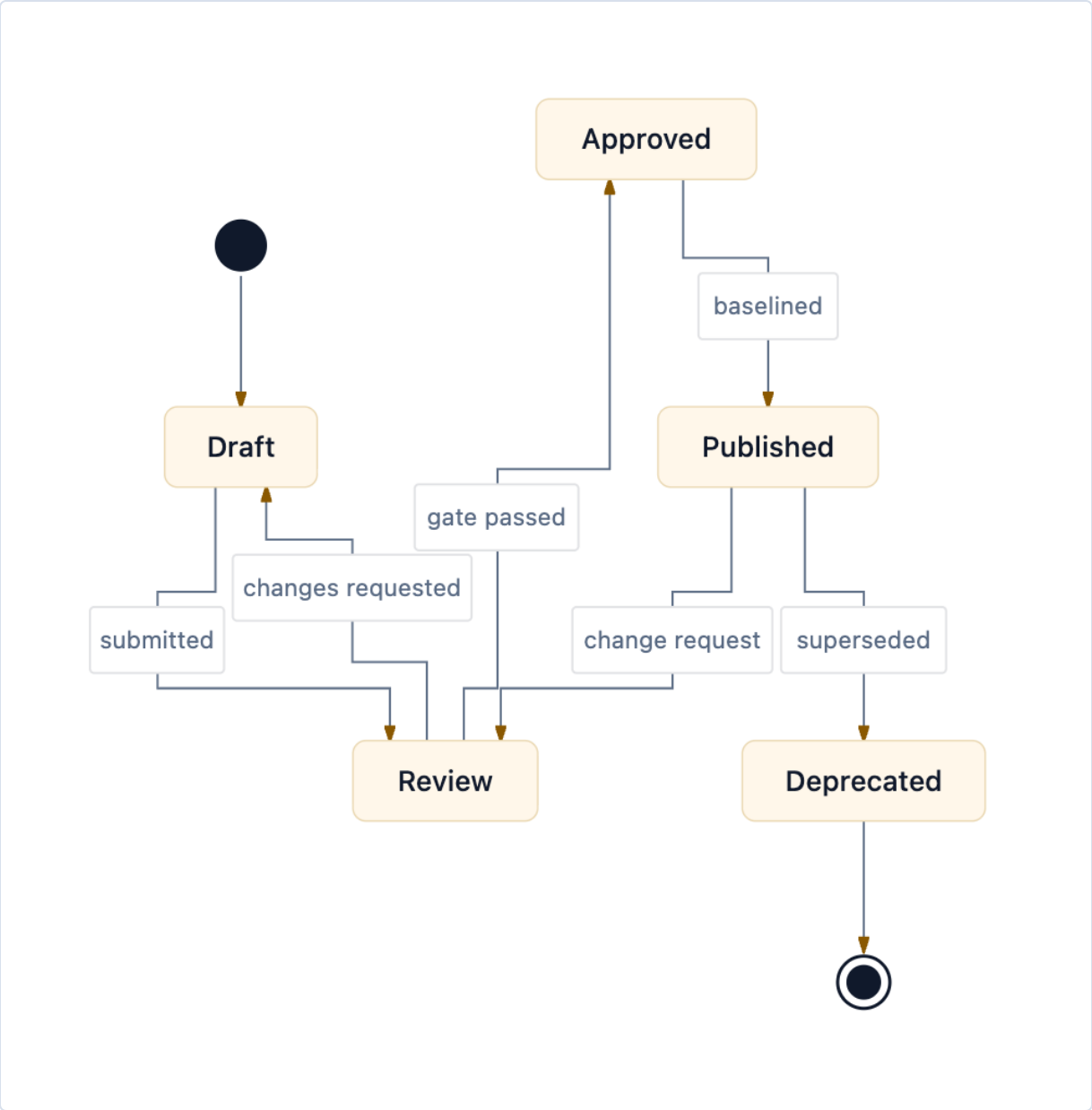


Figure 21.1: States of a governed context artefact. Every change to a published artefact goes back through review; nothing is edited in place.

What a Gate Checks

A gate is a list of named requirements, each with a validator, and a verdict that is the conjunction of their results. The validators are the

deterministic and judge-based checks of Chapter 14; the gate is the organisational decision to require them.

```
async function validateGate(stage, gate) {
  const requirements = getGateRequirements(stage, gate);
  const results = {};
  for (const req of requirements) {
    results[req.name] = await req.validate();
  }
  const passed = Object.values(results).every(r => r.passed);
  return { passed, results, recommendation: passed ? "APPROVED" :
"NEEDS_WORK" };
}
```

The function combines validation results. Teams still need to agree and record what `getGateRequirements` returns at each stage, and decide who handles a failed check.

Change Management

Published context changes. Vocabularies grow, domain models are corrected, regulations shift what a term is allowed to mean. Governance does not prevent change; it ensures that change is proposed, its impact is known before it lands, and the approval is attached to the record.

```
class ContextChangeManager {
  async requestChange(change) {
    const impact = await analyzeImpact(change);      // what depends
on this?
    const approvals = await getApprovals(impact);    // who must
agree, given the impact?
```

```
    if (approvals.allPassed) {
      await applyChange(change);
      await logChange(change, approvals);
    }
    return { approved: approvals.allPassed, impact, approvals };
  }
}
```

Impact analysis identifies the owners who need to review a change. Chapter 14 showed how a dependency graph over artefacts answers "what does this change affect"; at enterprise scale the same graph answers "whose approval does this change need". A change to a term used by one project needs that project's owner. A change to a term in the shared thesaurus needs every owner whose artefacts reference it. Deriving approvals from these dependencies makes the review scope explicit.

Two rules keep change management from decaying into a queue nobody reads:

- **The proposer supplies the evidence.** A change request to shared context carries the reconciliation: which existing terms it touches, why the existing definition is wrong or incomplete, and what will break if the change is not made. A request without evidence is returned, not reviewed.
- **Automated approval is bounded in writing.** Some changes can be approved by a validation gate alone: a synonym added to a term whose definition does not change, a document re-indexed under an existing vocabulary. Which changes those are is a governance decision, recorded as a dated line in a decision log,

and the carve-outs (anything touching security, anything that weakens a gate, anything the validators cannot check) always go to a person. Chapter 23 describes a single-repository version of this rule and what happened when the automation it bounded turned out to be weaker than believed.

Context Health Across Projects

Measure the following metrics per project, then review them across projects to identify shared problems.

```
const healthMetrics = {
  // Structural health
  vocabularyCoverage:      "share of a project's domain terms present
in the shared graph",
  artefactReconciliation:  "share of artefacts aligned with the
current domain model baseline",

  // Operational health
  contextDrift:            "rate of unreconciled change to published
artefacts per week",
  retrievalAccuracy:       "share of KG queries whose results a
reviewer confirms relevant",

  // Governance health
  gatePassRate:            "share of gates passed first time",
  changeApprovalLatency:   "days from change request to decision",
  gateCalibrationAge:      "days since each automated validator was
last calibrated"
};
```

Three of these carry warnings. A **gate pass rate** at which almost nothing is ever refused is not good news; it means the gates are not

checking anything the teams were not already doing, and the governance lead should look for the gate that has quietly become a formality. **Change approval latency** that climbs is the leading indicator of teams routing around governance, and the usual cause is approvals assigned by role rather than derived from impact. **Gate calibration age** is the enterprise form of the lesson in Chapter 23: an automated validator whose last calibration was before the model behind it changed is a gate on faith, and a governance function that does not track this number is approving on a judge it has not measured.

Avoid relying on a single composite health score. Averaging coverage, drift, pass rate and latency can conceal which project needs attention and why. Report the metrics separately, with the three warnings above attached. If teams also use a composite score, review its components before deciding what to investigate.

Roles at Enterprise Scale

Chapter 23 names four roles for a single practice. Enterprise governance adds one and changes the shape of another.

The added role is the **context owner** for each shared artefact: the person whose approval a change to that artefact needs, named in the artefact's metadata, and replaced explicitly when they move on. Shared context with no named owner is shared context that nobody will defend at a gate, and it drifts first.

The changed role is the **governance lead**, who at single-practice scale decides what automation may decide, and at enterprise scale also decides what *other teams* may decide about shared context. The useful test for whether the role is working is the reversal count from Chapter 23 applied across teams: how many changes to shared context, approved locally, were later reversed because another team's dependency was missed. Zero reversals across many changes means impact analysis is working. Several means the dependency graph is incomplete, and the fix is to complete the graph, not to add approvers.

The governance owner shall define who can access decision evidence, how long it is retained and which changes invalidate approval. A trace reference is useful only while an authorised reviewer can resolve it or identify its retention status. Chapter 17 explains the record; Chapter 16 explains how promotion authority is bounded.

Govern the Harness as an Operational Service

The Australian Signals Directorate's September 2026 agentic-AI harness guidance recommends least-privilege access, oversight, monitoring and governance of the software connecting models to organisational tools and data. This operational harness is broader than Chapter 15's evaluation harness. State which responsibility is meant before assigning its owner.

For a security owner’s concern about excessive authority, derive requirements for permitted tools, sensitive-action review, evidence access and escalation. Allocate them to the relevant policy, execution and evidence services. The solution shall enforce the agreed restrictions at the action boundary; a diagram or favourable judge result does not implement them. Review those controls when tool permissions, model configurations or fallback routes change. The guidance informs the control design and does not certify this book’s example or the author’s implementation.

Summary

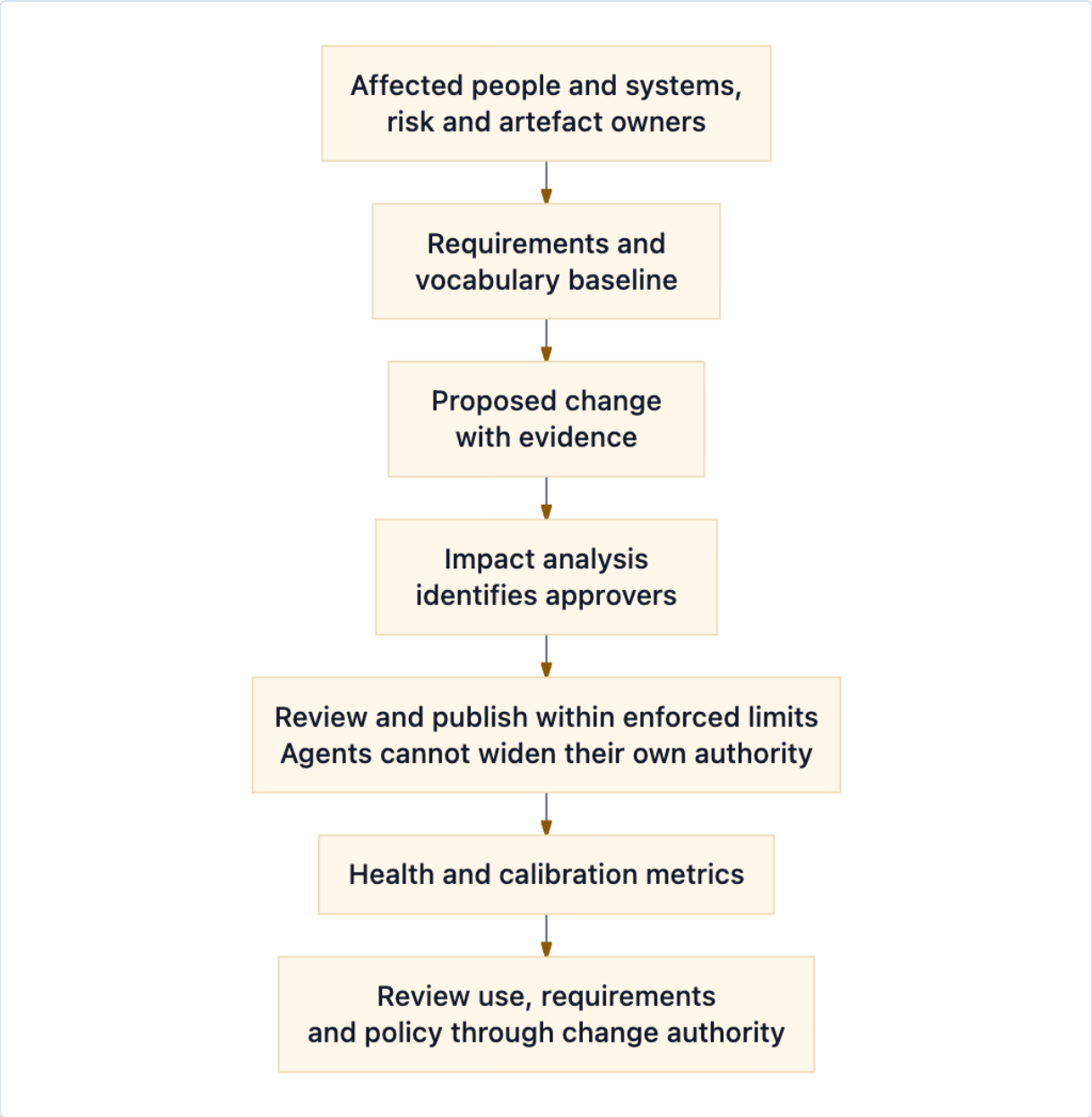


Figure 21.2: Chapter 21 recap: Enterprise governance joins named owners, baselines, impact-based approval and separately reported health measures.

Enterprise context governance needs:

- A gated lifecycle with few, named gates, including the two that are specific to context: vocabulary and knowledge graph baseline

- Artefacts that leave the published state only through review, never by editing in place
- Change management where impact analysis derives the approvers and the proposer supplies the evidence
- Automated approval bounded in writing, with carve-outs that always reach a person
- Agent evolution confined to enforced operating boundaries within the organisation's approved risk appetite
- Health metrics reported separately, with calibration age tracked for every automated validator
- A named owner for every shared artefact and a governance lead measured by cross-team reversals

The result is a current baseline, enforced operating limits and an explicit route for changing them. New evidence can reopen intended use or requirements through the applicable authority; the agent's own permissions remain bounded meanwhile. Chapter 22 returns to retrieval as a governed component choice, combining the lanes introduced in Chapters 6–9 and measuring the selector as well as each lane.

Key Sources: The reconciled domain model and lifecycle gates of the Zestic AI delivery framework (Chapters 3 and 4); the validation gates and impact analysis of Chapter 14; the single-practice governance constitution and calibration record of Chapter 23; enterprise software governance literature on stage-gate processes (Cooper, "Winning at

New Products", 1986 onward) and configuration management baselining (IEEE 828).

Additional source: Australian Signals Directorate, *Agentic AI harnesses*, 11 September 2026: least privilege, oversight, monitoring and governance.

Risk governance sources: NIST AI RMF 1.0, GOVERN 1.3, 1.5 and 2.3; HM Treasury, *The Orange Book: Management of Risk — Principles and Concepts*. The containment argument also draws on the author's unpublished note, "Liability is the third line of cost of goods sold", 10 August 2026; its unresolved legal and insurance questions are not treated as established findings here.

Chapter 22: Hybrid Pipeline Patterns

Chapter 6 established that deterministic search and vector embeddings serve different purposes, and Chapter 7 showed how Terraphim's graph scoring gives deterministic precision for vocabulary it knows. A system receiving several kinds of query needs to combine these methods. This chapter recommends a router with a cheap deterministic first lane, reserving fan-out across all lanes for cases where the additional recall justifies its cost.

A router also introduces classification errors. A misrouted query may reach a method unable to answer it, with no second retrieval attempt. The final section examines this failure mode.

The return to retrieval follows the operational and governance chapters deliberately. We can now ask which route supplies the required reference under the current workload, budget, evidence and permission constraints. In the maintenance case, matching a known job name and retrieving an unfamiliar incident report may need different lanes. Neither route may omit a protected condition or treat retrieved text as action authority.

Where Hybrid Pipelines Sit

Retrieval designs can be placed on a progression, and knowing where a design sits says what it lacks. TrustGraph (2025) describes the progression as eight steps: models answering from training data

alone; RAG, stuffing prompts with chunks retrieved by embedding similarity; GraphRAG, with richer knowledge representations; OntologyRAG, with structured ontologies for precision; retrieval analytics specialised for temporal, anomaly and clustering problems; self-describing stores that carry metadata about their own structure; dynamic strategies derived by the model for unseen information types; and autonomous learning, where systems re-ingest their own outputs. In the author's reading of the systems he has seen, most production work sits at steps two to four, and the patterns in this chapter address three and four with reaches towards five.

The point of the progression is that each step does not replace the last. A mature system uses plain RAG for some queries, graph-grounded retrieval for others and ontology-driven precision for a third class, and the hybrid pipeline is the layer that decides which.

One finding from the same source shapes a design choice below: TrustGraph reports that passing graph results to a model in a structured form such as Cypher or RDF improves responses despite the extra tokens, because the syntax carries what is a node, a property and a relationship. That is their report of their own experiments and has not been reproduced here; the author's experience is consistent with it, and the pipelines below do not flatten graph results to prose before the model sees them.

The Architecture

Figure 22.1 shows the architecture the routing patterns below are variations of.

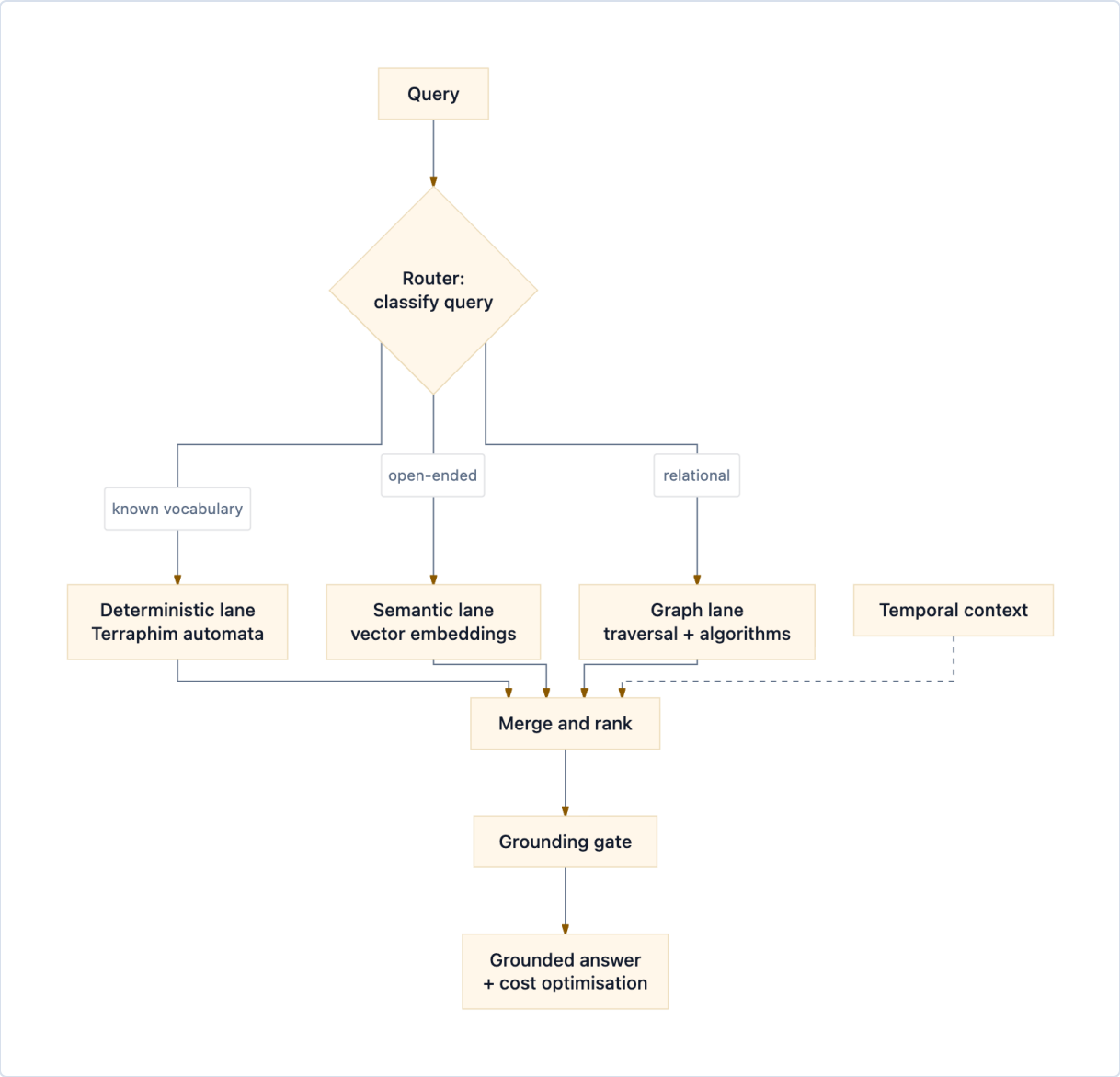


Figure 22.1: Query routing across deterministic, semantic and graph lanes, merged and passed through a grounding gate.

A hybrid pipeline has a router, lanes, a merge step and a gate. The router classifies the query; the lanes are the retrieval methods; the

merge combines whatever lanes ran; the grounding gate of Chapter 9 checks the result before it reaches the model or the user.

The router chooses which retrieval methods run. Its signal is **vocabulary coverage**, defined below as the share of query characters covered by recognised vocabulary terms.

```
function classifyQuery(query, kg) {
  const coverage = matchVocabulary(query, kg).coverage;   // 0 to 1,
defined below
  if (coverage > 0.8) return "deterministic";             // the graph
knows this query
  if (coverage > 0.4) return "hybrid";                     // known
terms plus unknown ones
  return "semantic";                                       // the
graph knows none of it
}
// The grounding gate is not a route. Every candidate answer passes it
after
// retrieval; accuracy-critical queries (requiresFactCheck)
additionally fail
// closed if the gate cannot resolve a claim.
```

Coverage is the share of the query's characters inside matched surface forms of the active role's thesaurus after normalisation (lower-casing, abbreviation expansion where the thesaurus lists the abbreviation, stop words excluded from the denominator), so a multi-word term counts in full and an unmatched generic word counts against it. Chapter 6 gives the routing code.

The thresholds are the author's settings for a vocabulary of a few hundred terms; a clinical vocabulary of a million surface forms would set them differently. The principle is stable: when the graph knows

most of the query, deterministic retrieval is faster and more precise; when it knows little, only semantic retrieval can help; in between, both run.

Three Routing Patterns

Cascade: try cheap first

Figure 22.2 shows the cascade with the indicative cost of each lane.

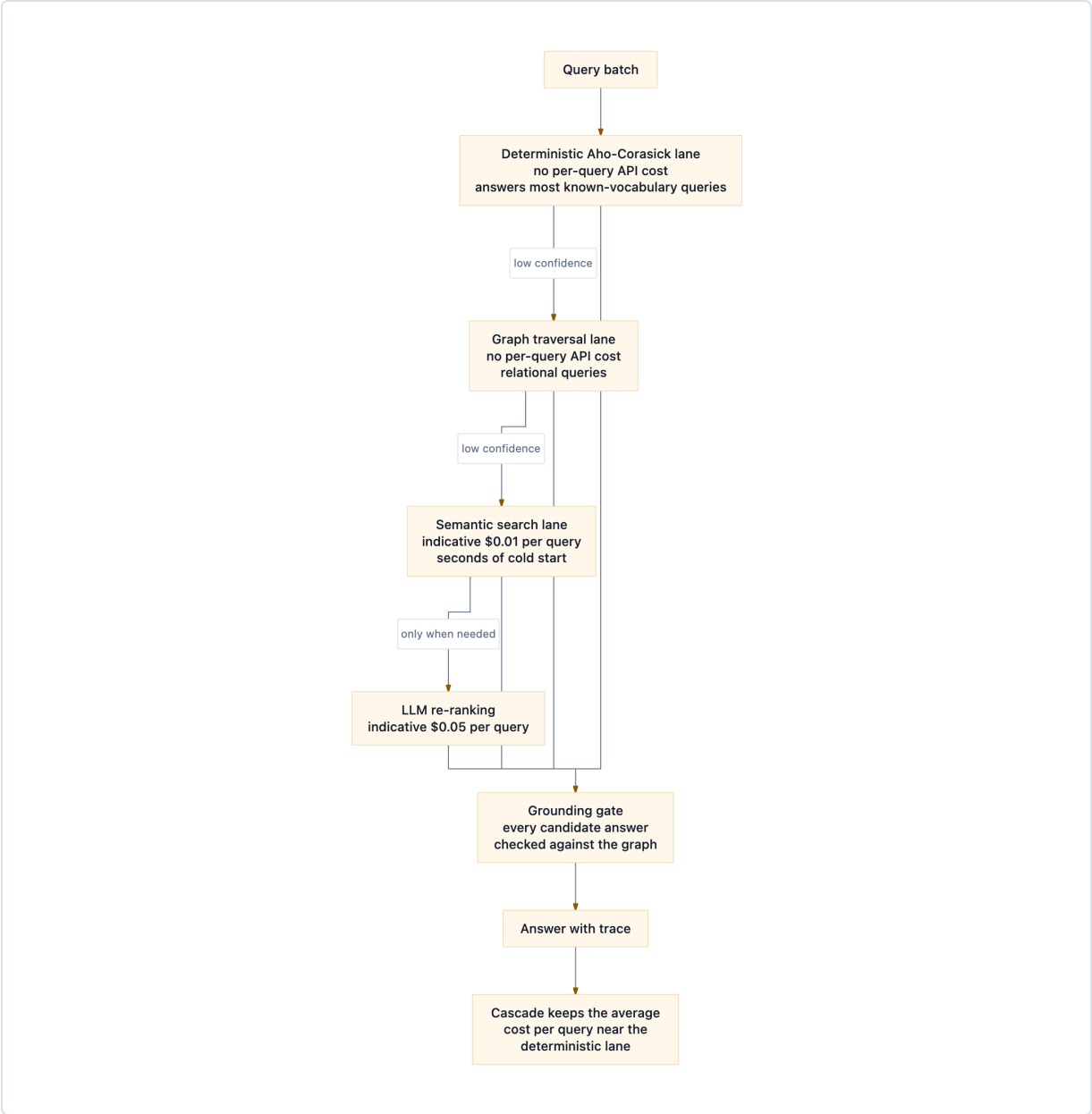


Figure 22.2: The cost cascade: each lane is tried only when the cheaper one lacks confidence, keeping the average cost per query near zero.

The cascade runs the cheapest lane first and escalates only when its confidence is low: deterministic search, then graph traversal, then semantic search if the deterministic result is weak, then LLM re-ranking if the semantic result is weak. Grounding is not a lane in the cascade; every candidate answer passes the grounding gate after

whichever lane produced it. The cascade optimises cost. In the author's deployments, where the vocabulary was built for the domain, most queries resolve in the deterministic lane and the average cost per query stays near zero; that share is an observation from a handful of systems, not a measured population. The cascade's weakness is latency on the queries that escalate, which pay for every lane they pass through.

Fan-out: run everything, merge

Fan-out runs the deterministic, semantic and graph-traversal lanes in parallel and merges the results, boosting documents that more than one lane found. It optimises recall and costs the most, because every query pays for every lane. The merge weights (the author starts at 0.4 deterministic, 0.35 semantic, 0.25 graph) are a tuning problem with no principled answer: they encode a belief about which lane is more trustworthy for this corpus, and they need re-tuning when the corpus or the vocabulary changes. Fan-out is the right pattern when a miss is expensive and a query is rare; it is the wrong default.

Semantic enrichment: deterministic seed, then fill gaps

The third pattern is the one the ProCoder case study below uses. It runs in five phases: match the query's known terms with the automaton; expand them through the graph to a depth of two; extract the query's concepts and check which are not covered by the expansion; run semantic search only for the uncovered concepts; retrieve documents for the whole expanded set and rank by graph connectivity. The principle is **deterministic methods establish a**

precise foundation and probabilistic methods fill the gaps, which is this book's hybrid thesis in one sentence. The semantic lane runs only on what the graph could not reach, so its cost scales with the vocabulary's blind spots rather than with query volume.

The four kinds of context a pipeline assembles, and where each comes from:

Context	Source	Purpose	Cost
Direct	Deterministic search	Exact matches for known vocabulary	None per query
Structural	Graph traversal	Relationships, paths, connectivity	None per query
Semantic	Embeddings	Novel terms and paraphrase	Per query
Gap	Semantic fallback	Concepts the graph analysis found missing	Per query, only when gaps exist

Choosing a Storage Backend

The retrieval architecture can work over RDF triples, property-graph nodes and edges, or joined relational tables. TrustGraph (2025) reports deployments with over a billion nodes and edges in Apache Cassandra, alongside Neo4j deployments using the same retrieval layer. Its account suggests that the pipeline can be separated from the backend. Evaluate ontology quality, synonym coverage,

relationship completeness and update behaviour alongside the storage requirements.

The Untapped Toolbox

Decades of graph-algorithm research are available to retrieval systems; in the author's reading of the open-source GraphRAG implementations, community detection is used and little else is. Four uses are cheap and deterministic: choosing traversal depth per query type rather than fixing it; clustering documents by graph structure rather than by text similarity; treating densely connected subgraphs as high-confidence knowledge domains; and flagging entities with unusual connectivity as errors or discoveries. A pipeline that seeds PageRank from the query's matched nodes and then runs community detection over the high-ranked neighbourhood returns documents from the community most relevant to the query, with every step explainable and none of it probabilistic. The author's systems use the first two; the other two are on the list.

Temporal Context

Information changes, and freshness is not accuracy. Old data corroborated by several sources may be more trustworthy than new data from one; a fact established by consistent observation over time is on firmer ground than a fresh claim. A hybrid pipeline weights the two signals by query type: for a factual query, corroboration dominates recency (the author weights them about 70 to 30); for a

question about current state, recency dominates (30 to 70); for anything else, evenly. The weights are judgements, and the point is that there are two signals, not one.

Cost

The cascade works because the lanes differ in cost by orders of magnitude. The figures below are illustrative only: they assume a query of under a hundred tokens, a hosted embedding model and a hosted re-ranking model at 2026 list prices for a mid-tier provider, and exclude indexing and storage. Replace them with your own before deciding anything:

Method	Per-query cost	Cold start	Precision	Recall
Deterministic (Aho-Corasick)	~0	none	High for known terms	Low for novel terms
Graph traversal	~0	none	High where connected	Medium
Semantic search	~$0.01	seconds	Medium	High
KG grounding	~$0.001	none	High	Medium
LLM re-ranking	~$0.05	none	High	High

The deterministic figures are the author's measurements described in Chapter 7; the semantic and re-ranking figures are indicative 2026 ranges taken from the public pricing pages of hosted embedding

and re-ranking services (Key Sources), rounded to an order of magnitude. A simple estimator that classifies a batch of queries with the router and sums the lane costs gives the expected spend before any of it is incurred, which is how the author decides whether a corpus is worth building a vocabulary for.

Case Study: Terraphim and ProCoder

ProCoder AI Pack selects the files to include in a coding agent's context with hand-written glob patterns: `src/auth/*.rs`, `src/models/user.rs`. That works for a task whose relevant files the developer already knows, and fails when they do not, when related concepts span directories, or when new files appear and the globs are not updated.

The hybrid replacement follows the semantic-enrichment pattern: extract the prompt's key terms; expand them through the Terraphim thesaurus ("authentication" becomes auth, login, credential, session); search the indexed codebase for the expanded set; check the matches for concepts from the prompt that none of them covers; run semantic search for those gaps only; return the union. Both deterministic steps run in under a millisecond on the author's hardware; the whole selection, including graph traversal, in about two.

Aspect	Manual globs	Hybrid pipeline
File selection	User-defined patterns	Derived from the prompt through the graph

Aspect	Manual globs	Hybrid pipeline
Synonyms	None	Thesaurus edges
Related concepts	Added by hand	Graph traversal
Missed concepts	Undetected	Flagged by gap analysis
Maintenance	Per task	Follows the graph

The hybrid approach is not always better. For a small task the developer understands, globs are faster to write than a vocabulary is to build. For prompts that span several concerns, the pipeline retrieved more complete file sets in the author's trials; no controlled comparison is claimed.

Where Routing Fails

The router is a classifier, and classifiers misclassify. A query with high vocabulary coverage that is nevertheless about something the graph does not encode (a question about *why* two known entities relate, when the graph records only *that* they do) is routed deterministically and returns a precise answer to the wrong question. A query with low coverage because the user used an abbreviation the thesaurus lacks is routed semantically, at cost, when one synonym edge would have made it free. Fan-out does not have this problem and pays for it on every query; the cascade has it and pays only on the misroutes.

The mitigations are the ones this book keeps returning to. Log the route taken with every query and read the log; the misroutes cluster.

Feed the abbreviations the semantic lane keeps rescuing back into the thesaurus (Chapter 8). And keep the grounding gate after every lane, so that a confident wrong answer from a misroute is caught for the same reason any other confident wrong answer is.

Measure routing separately from lane quality. A useful lane cannot repair a request sent to the wrong capability. Chapter 12 supplies negative and compound routing cases; Chapter 16 separates typed selection from permission, and Chapter 17 links the chosen lane to its observed outcome.

Summary

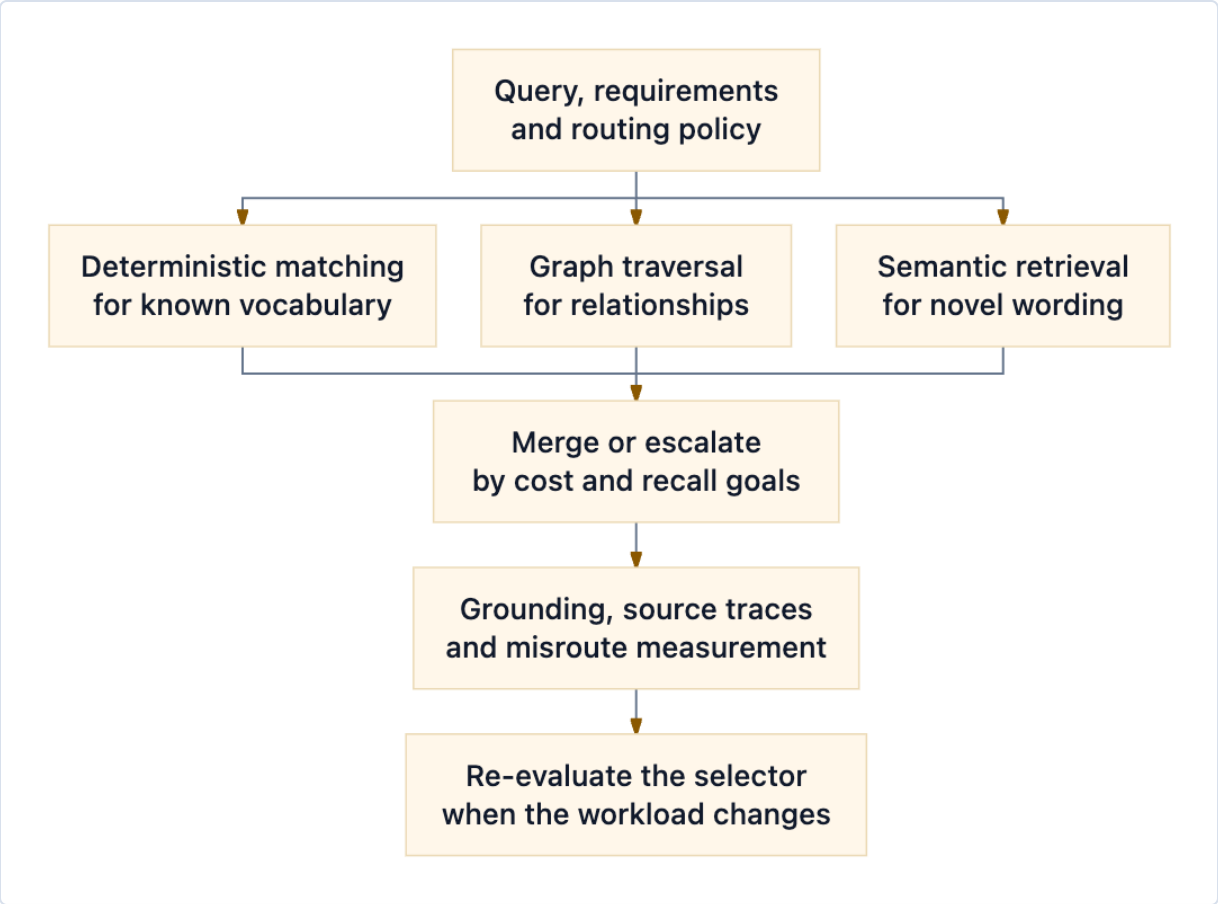


Figure 22.3: Chapter 22 recap: Hybrid routing combines complementary retrieval methods and checks accuracy-critical output.

Hybrid pipelines combine retrieval methods by query:

- Deterministic for known vocabulary: fast, precise, explainable, free per query
- Graph traversal for relationships: no per-query cost
- Semantic for novel queries: paraphrase and the long tail, at a cost
- Grounding for anything accuracy-critical

Three routing patterns serve three priorities: the cascade for cost, fan-out for recall, semantic enrichment for completeness. Measure

misroutes separately from retrieval quality within each lane.

The handoff is an assembled pipeline with measured selection failures, source traces and explicit fallback limits. Chapter 23 keeps those measures current and connects failures to owned corrective work. A change in query distribution can require a new selector evaluation even when the underlying corpus and graph have not changed.

Key Sources

- TrustGraph, documentation and engineering blog at trustgraph.ai (2025): the eight-step progression, the structured-format finding and the backend-equivalence claim are the vendor’s reports of its own deployments and experiments, not reproduced here
- Terraphim AI implementation and the author’s measurements described in Chapter 7
- ProCoder AI Pack documentation (context_globs and working_globs configuration), for the glob-based baseline the case study replaces
- Microsoft GraphRAG documentation, for the GraphRAG stage of the progression
- Hosted embedding and re-ranking service pricing: provider public pricing pages, checked October 2026; indicative per-query ranges, not quotes
- Chapters 6, 7, 8 and 9 of this book

Chapter 23: Building a Context Engineering Practice

The previous chapters describe mechanisms: knowledge graphs, token budgets, skills, hooks, validation gates, judges. Maintaining them requires named owners, measurements and a process for acting on the findings. This chapter develops that practice through a single worked example: a knowledge repository that used automated judge assessments for most of a year, with recorded verdicts and a calibration whose later review exposed important limitations.

The example is one person's repository. It exposes failures that an enterprise practice also needs to check for: inactive gates, uncalibrated judges and misleading metrics.

Roles and Responsibilities

A context engineering practice needs four roles. In a large organisation they are four people or four teams; in a small one they are hats worn by one or two people, and the discipline is to notice which hat is on.

```
roles:
  - name: Context Engineer
    responsibility: Building and maintaining knowledge graphs
    skills: [ontology, graph databases, semantic modelling]

  - name: Prompt Engineer
```

```
    responsibility: Designing and optimising prompts and skills
    skills: [LLM mechanics, testing, iteration]

  - name: Validation Engineer
    responsibility: Building and calibrating validation gates
    skills: [testing, monitoring, drift detection, statistics]

  - name: Governance Lead
    responsibility: Deciding what the gates are allowed to decide
    skills: [process design, auditing, compliance]
```

The context engineer owns the vocabulary: the knowledge graph, the thesaurus, the synonym edges of Chapter 8. When a new term appears in three documents and the graph does not know it, that is their queue. The prompt engineer owns the skills and rules files of Chapters 12 and 13, and is the person who notices that a skill fires on the wrong trigger. The validation engineer owns the gates of Chapter 14, which in practice means owning the calibration set, because a gate without a calibration set is a guess with a log file. The governance lead owns the question the other three cannot answer for themselves: which decisions may the automation take alone, and which must wait for a person.

The separation that matters most is the last one. In the worked example the governance decision was recorded as a dated line in a decision log: automated rule changes apply on a judge GO, except for security rules, changes that weaken a gate, and rules that cannot be verified, which always wait for a human. That line is the practice's constitution. Everything below operates inside it.

The Gate

Figure 23.1 shows the loop the practice runs. Work enters through hooks, is judged before it lands, feeds a learning store when it fails, and the learning store proposes changes to the rules that are themselves judged before they apply. Calibration sits outside the loop and periodically checks whether the judge inside it still deserves its position.

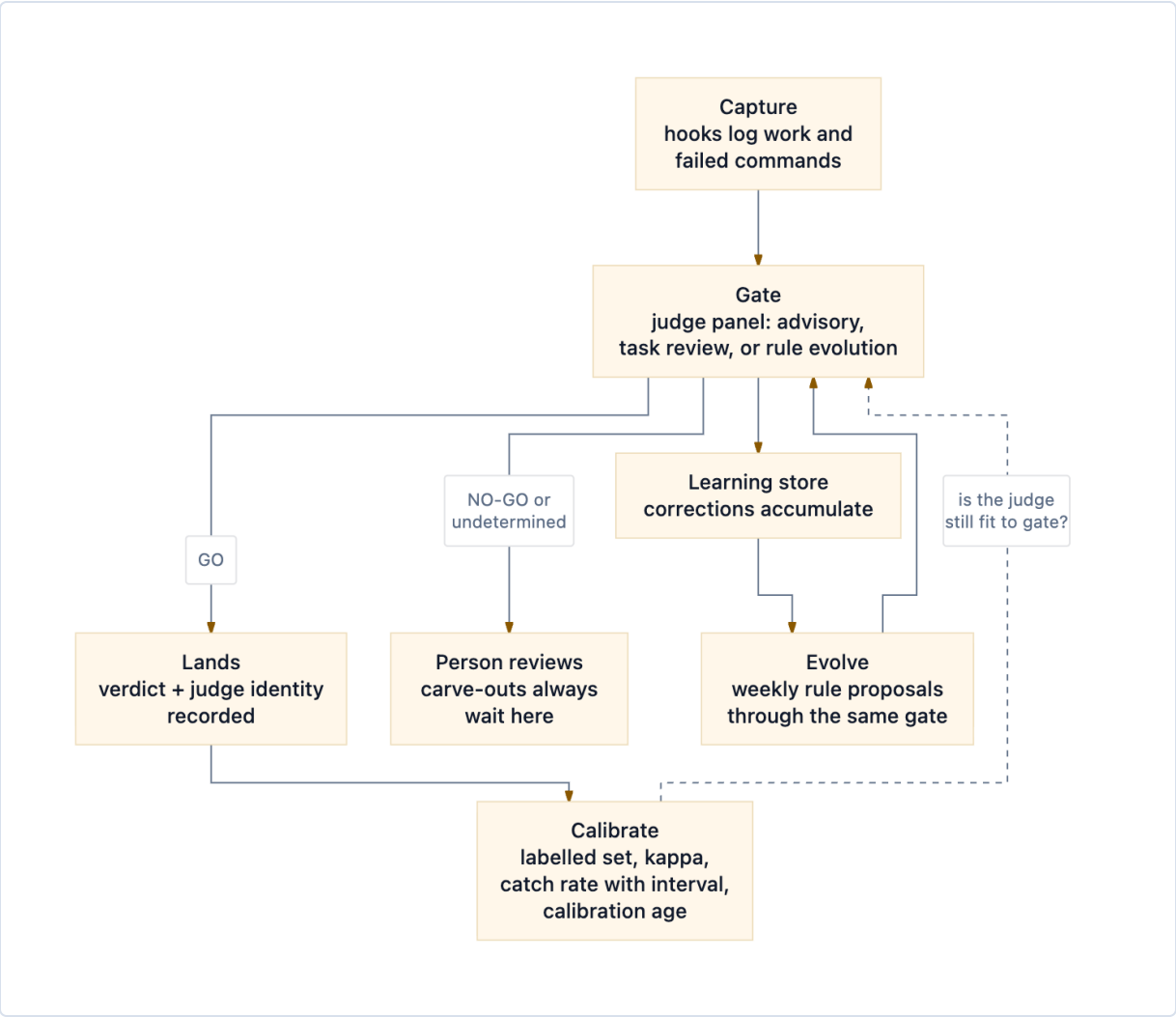


Figure 23.1: The practice loop: capture through hooks, gate through a judge panel, evolve rules through the same gate, and calibrate the judge against a labelled set on a schedule.

The judge runs in three configurations, chosen by what is at stake:

- **Pre-push, advisory.** A single fast model scores every knowledge file before it is pushed. An undetermined verdict is logged and the push proceeds. This tier exists to catch raw web clips and empty notes, not to make decisions.
- **Task review, gating.** A fast model and a larger reasoning model run in sequence; a split goes to a third model as tiebreaker, and anything still undetermined escalates to a person. This gates the research and design documents of the disciplined development phases.
- **Rule evolution, panel.** Both models run and must concur for a GO. A split goes to the tiebreaker; any undetermined verdict escalates. This gates changes to the practice's own rules, which is the one place where a wrong GO compounds.

The panel requires agreement rather than an average score, so a lenient judge cannot offset a strict one's refusal. An unavailable judge triggers escalation. Calibration exposed a further limitation: one panel member contributed no measured failure detection.

The Calibration Story

In March 2026 the practice ran its first calibration. Fifty knowledge files were labelled GO or NO-GO by a frontier model used as the reference, producing 42 GO and 8 NO-GO. Four candidate judges

then scored the same fifty files on the same rubric, and the results were compared with the reference.

The headline at the time was that the larger reasoning model was “the best judge tested and exceeded targets”: 90 per cent verdict agreement, five of the eight failures caught, two false alarms. The fast model managed 84 per cent agreement. Both numbers went into the practice’s records and the tier configuration was set accordingly: fast model for pre-push, fast plus reasoning model for everything that gates.

Six months later, a reading of the evaluation literature prompted a recomputation, and the recomputation changed the record in three ways.

First, the fast model’s 84 per cent agreement was exactly the agreement a judge gets by saying GO to everything on a set where 84 per cent of files are GO. Its Cohen’s kappa was zero. It had caught none of the eight failures. The practice had been running an advisory pre-push assessment, and half of a two-judge panel, on a model with no measured ability to detect a bad file. The panel’s catch rate was the reasoning model’s alone; the fast model added latency and nothing else.

Second, the reasoning model’s five of eight, quoted for six months as evidence of excellence, carries a Wilson 95 per cent interval from 0.31 to 0.86. On eight negatives the practice could not tell a judge that catches a third of failures from one that catches six in seven. Its kappa of 0.61 clears the usual bar for gating a release and does not

reach the bar for unsupervised use. The rule-evolution gate was, by the practice's own constitution, unsupervised use.

Third, by the time of the recomputation the reasoning model behind the deep tier had been replaced by a newer version, and the tiebreaker had never been calibrated at all. The kappa in the records belonged to a model no longer in the configuration. During the review of this book's own chapters, a further step was taken: the model identifiers in the shared configuration were found to point at a decommissioned model and an expired credential, and the dispatcher had been returning an undetermined verdict in under two seconds for anyone who called it. No gate had failed open, because undetermined escalates, but no gate had been judging either.

The reasoning model met the practice's chosen threshold for gating, but the configuration outlived its calibration. Assign recalibration to the validation engineer, schedule it and reserve a hold-out set that is never used to tune the prompt. Record kappa and failure detection alongside agreement, and tie each verdict to the model version that produced it.

A Shared Vocabulary for Failure

The calibration also produced a failure taxonomy. The eight NO-GO files were each described in a paragraph of plain language, and those descriptions were grouped into seven categories: raw capture, no actionable insight, wrong audience, product bookmark,

unsubstantiated claims, structural incoherence and linguistic imprecision.

Two of the plain-language descriptions became the practice's shorthand. A "bookmark with a relevance label" is a repository feature list with two sentences of why-it-matters appended and no evaluation of whether it does. A "product page captured as knowledge" is vendor documentation filed as if someone had assessed it. Once named, both became easy to refuse at the point of writing, which is cheaper than refusing them at the gate.

The taxonomy also revealed the gatekeeper dimension. Seven of the eight failures scored two or lower on usefulness to the intended reader, and all eight carried the "no actionable insight" category. In this sample, the three-dimension rubric largely tracked usefulness. Chapter 14 describes the replacement: one binary question per category, each answered with a quoted sentence, and a verdict derived from the answers. The categories came from observed failures and still need testing on new material.

Skill Development

Training a team for this practice is less about teaching the mechanisms, which are documented, than about teaching the habits that the mechanisms do not enforce:

```
curriculum:
  foundation:
```

```
    - Context engineering fundamentals and the four dimensions of
context
    - Knowledge graph basics and the thesaurus pattern
    - Token budgets and what constitutional context means
  intermediate:
    - Building and validating a knowledge graph
    - Designing a validation gate and its calibration set
    - Reading a verdict: what kappa, catch rate and an interval each
tell you
  advanced:
    - Multi-agent context and the context wall
    - Governance: deciding what automation may decide
    - Recalibrating a judge after a model change
```

Include verdict interpretation in training. The fast-model example shows why engineers need to ask for class balance, kappa and the failure catch rate when presented with an agreement percentage.

Measuring Success

Track judge performance, throughput and practice health separately:

```
const practiceMetrics = {
  // Gate quality (recomputed at every calibration, with intervals)
  judgeKappa:         "Cohen's kappa against the reference set, per
tier",
  catchRate:          "NO-GO cases caught, with a Wilson 95%
interval",
  flipRate:           "verdict changes on re-running the same input",
  calibrationAge:     "days since the gating model was last
calibrated",

  // Throughput and cost
  escalationRate:     "share of verdicts that reached a human",
  costPerCorrect:     "judge spend divided by verdicts later confirmed
right",
```

```
  // Practice health
  vocabularyLag:      "terms seen in three documents but absent from
the graph",
  rulePromotions:     "rule changes applied, and how many a human
later reversed"
};
```

Four of these deserve comment. Calibration age is the metric that would have caught the shelf-life problem: a gating model with a calibration age of never is a gate on faith. Flip rate measures repeatability on the assessed cases. Estimate uncertainty for the actual comparison; a flip rate does not establish a universal noise floor. Cost per correct outcome, rather than accuracy, is how to rank judge tiers against each other when both are adequate. And rule promotions later reversed by a human is the one number that tells the governance lead whether the constitution is set correctly: reversals warrant investigation of the failed criteria, while an absence of reversals requires checks that review actually occurred.

Define the reference and denominator when reporting unsupported-claim prevalence. That prevalence differs from catch rate, which measures the share of actual failures detected; report uncertainty for each.

The Closed Loop

A controlled practice closes this loop through an explicit promotion service. Corrections captured by Chapter 19’s hooks can inform proposed rule changes, but the inspected pre-push hook is advisory

and its standalone runner prints a result rather than guaranteeing durable storage. A model's GO is assessment evidence. The action path must separately enforce approval scope, governing policy, protected cases and exact artefact identity, then persist the decision and outcome. Record configured model identities honestly when immutable provider versions are unavailable. Chapters 16 and 17 specify these policy and recording responsibilities; the local tooling's intended workflow does not establish that every responsibility is implemented.

For some months the loop applied rule changes using a panel whose effective judge had a kappa of 0.61 and whose other member had a kappa of zero. The changes were small and logged; none has yet needed reversing. Those limits were not understood when the changes were approved. The scheduled remedy is a larger calibration set with at least thirty negatives, a second reference judge from another model family and a comparison of binary-question and current rubrics on the same files. Until that calibration runs, the records mark the rule-evolution gate as uncalibrated.

Controlled improvement needs evidence of benefit and protected-case preservation. Compare context-aware and context-blind assessments on matched cases when investigating the contribution of context. Freeze the grader and references, retain failed candidates, and estimate uncertainty for the actual comparison. A verdict flip rate describes repeatability; it is not a universal noise floor or a substitute for failure-detection sensitivity. Chapter 15 provides the harness and Chapter 17 records experiment lineage.

Summary

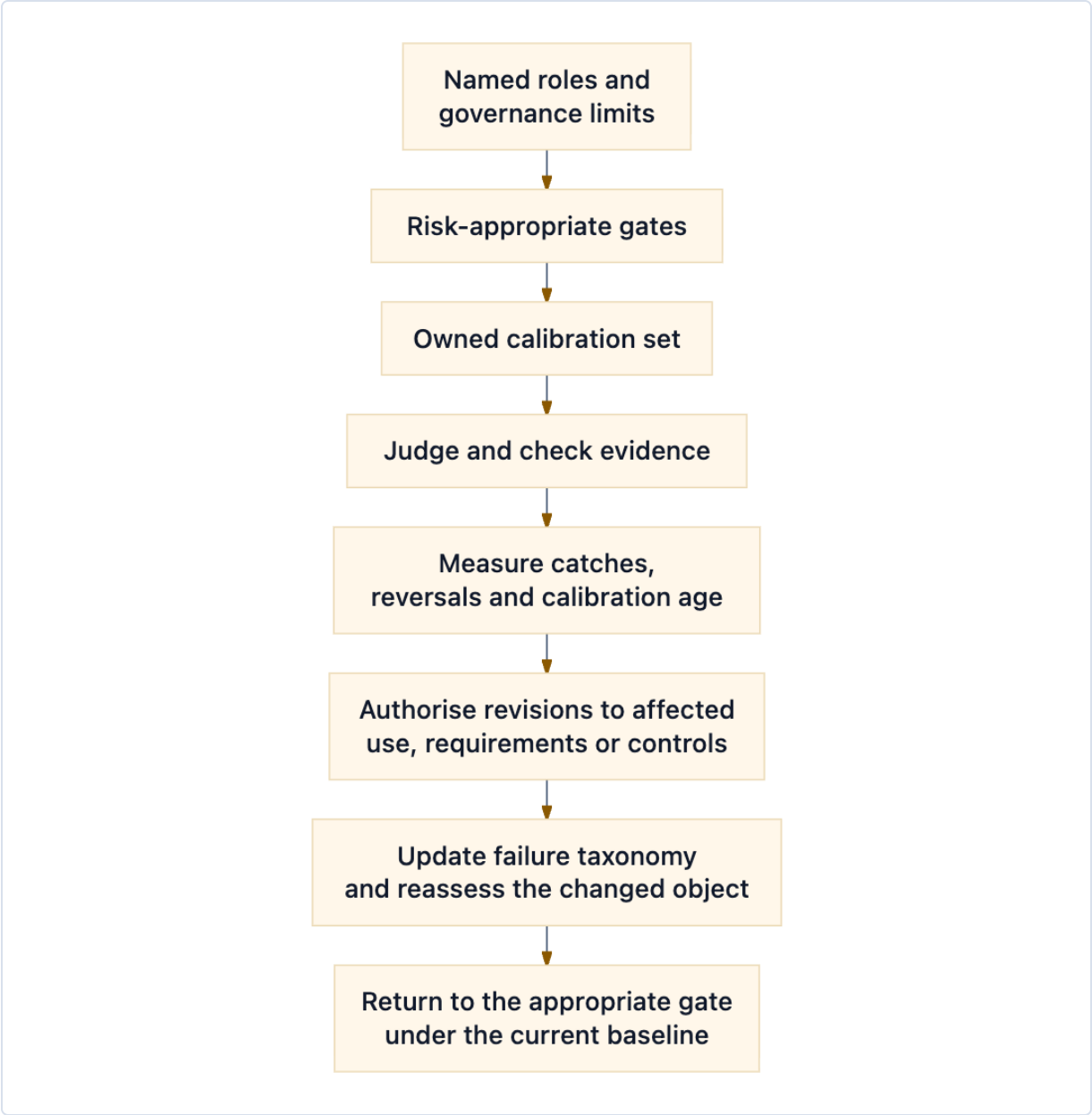


Figure 23.2: Chapter 23 recap: A context engineering practice needs accountable roles, calibrated gates and feedback from real failures.

Building a practice requires:

- Four roles, with the governance role deciding what the other three may automate
- A gate whose configuration matches what is at stake, from advisory to panel
- A calibration set owned by someone, recalibrated on a schedule, with negatives counted and intervals reported
- A failure taxonomy derived from the practice's own failures, feeding binary questions into the judge
- Separately reported metrics: kappa, catch rate with intervals, flip rate, calibration age, reversals
- A closed loop where every automated change carries the verdict and the judge identity that approved it

Return to the synthetic repair case: compare the observed job outcome with the need, current requirements and protected work, including effects on the quota-sharing team. Identify whether the failure concerns a source, selector, grader, policy, use assumption or requirement. The relevant owners authorise revisions and obtain new evidence; completing a review or closing a ticket does not itself demonstrate improvement.

This maintained evidence supports Chapter 24's component choices and Chapter 25's research questions. Keep the current basis inspectable while testing a proposed change. Stop, retain the baseline or escalate when the evidence cannot justify another autonomous step within the approved limits.

Key Sources: The author’s knowledge-repository judge (profiles, verdict log, calibration comparisons of 19 March 2026, error taxonomy of 24 March 2026, recomputation of 4 October 2026); Cohen (1960) on kappa; Wilson (1927) on the score interval; Makarevych, “Evals: How to Know Whether an AI System Works” (2026); Chapters 14 and 19 of this book.

Chapter 24: The Knowledge Graph Market Landscape

The previous chapters presented knowledge graphs as engineering infrastructure. This chapter surveys the available solutions, the changes the author has observed in their use, and criteria for evaluating them in your own context engineering practice.

The maintenance case now supplies concrete selection demands: source-linked retrieval, current rule and quota references, bounded loading and evidence that another reviewer can inspect. A category name or vendor demonstration cannot establish those properties for this workload. Evaluate candidate components against the allocated service requirements, operating constraints and observed failure cases established earlier. Revisit the architecture if no component can fulfil the requirement within the approved limits.

Two things to state before the survey. First, a disclosure: Terraphim, discussed under the fourth category, is developed by the author's company, Zestic AI; the evaluation framework at the end is written so that it can be applied against Terraphim as readily as against anything else, and the reader should check that it is. Second, scope: the chapter covers the four categories the author has evaluated or built on; memory layers for agents and graph-RAG libraries such as LightRAG, Graphiti and Cognee are adjacent and are not assessed here.

Categories of Knowledge Graph Solutions

Figure 24.1 shows the four categories this chapter covers.

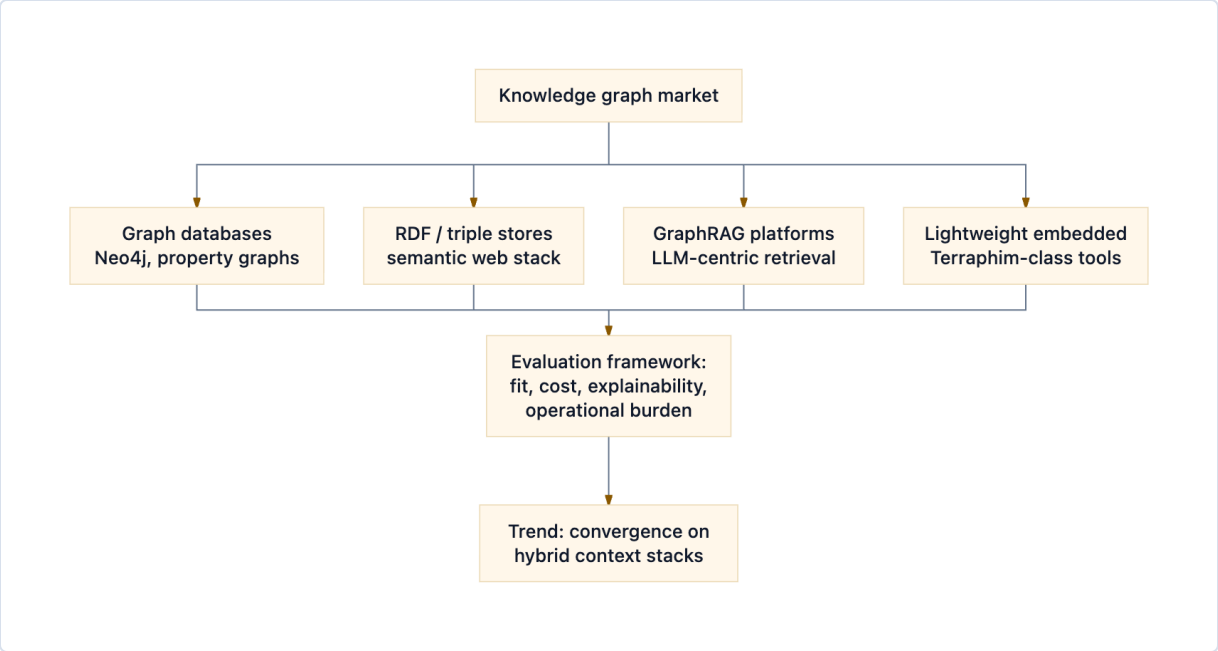


Figure 24.1: Four market categories and the evaluation framework applied to all of them.

Four categories cover the products the author has worked with. Each solves a different part of the problem, and the summary at the end says which part:

1. Graph Databases (General Purpose)

These are the established platforms for storing and querying graph-structured data.

Platform	Model	Query Language	Deployment
Neo4j	Property graph	Cypher	Self-hosted, AuraDB (cloud)

Platform	Model	Query Language	Deployment
Amazon Neptune	Property graph + RDF	Gremlin, SPARQL, openCypher	AWS managed
Azure Cosmos DB (Gremlin API)	Property graph	Gremlin	Azure managed
TigerGraph	Property graph	GSQL	Self-hosted, cloud
Stardog	RDF/OWL	SPARQL	Enterprise

Neo4j has led the DB-Engines ranking of graph database management systems for over a decade (db-engines.com, graph DBMS category, checked October 2026), with a mature ecosystem and strong community. Its Cypher query language was a principal input to the ISO/IEC 39075 GQL standard published in 2024. For teams already on AWS, Neptune offers a managed alternative with both property graph and RDF support.

These platforms provide graph storage and querying. The context engineering pipeline adds retrieval, reconciliation and validation on top of that storage layer.

2. LLM-Powered Graph Construction (GraphRAG)

A newer category uses LLMs to automatically extract entities and relationships from unstructured text, building knowledge graphs without manual curation.

Microsoft GraphRAG is the most prominent example. Its workflow:

1. **Indexing**: LLMs extract entities and relationships from a document corpus
2. **Community detection**: Graph algorithms identify clusters of related entities
3. **Summarisation**: LLMs generate summaries at multiple levels of the hierarchy
4. **Query**: Questions are answered using graph-structured context rather than raw document chunks

```
# Simplified GraphRAG extraction pipeline
def extract_graph(documents):
    entities = []
    relationships = []
    for doc in documents:
        # LLM extracts structured knowledge from text
        extraction = llm_extract(doc,
schema=entity_relationship_schema)
        entities.extend(extraction.entities)
        relationships.extend(extraction.relationships)

    # Build graph and detect communities
    graph = build_graph(entities, relationships)
    communities = detect_communities(graph)
    return graph, communities
```

Strengths: Handles large unstructured corpora; automated extraction reduces manual curation effort; community detection provides multi-scale summarisation.

Limitations: Extraction errors can propagate into the graph; extraction quality depends on the LLM and has no deterministic

guarantee; costs scale with corpus size. Extracted relationships require checking against their sources.

GraphRAG is best suited for exploratory analysis of large document collections where manual curation is impractical. It is less suitable for safety-critical applications where extraction errors carry real consequences.

3. Context Graph Platforms

An emerging category optimises graph-structured data specifically for AI consumption, rather than general-purpose graph queries.

TrustGraph represents this approach. Built on Apache Cassandra as its default triplestore, TrustGraph focuses on the pipeline from raw data to AI-ready context:

- Stores triples (subject-predicate-object) optimised for AI retrieval
- Supports RDF natively, with translation to property graph formats (Neo4j) available
- TrustGraph reports one production deployment running over a billion nodes and edges in Cassandra, and reports that its retrieval agents behave the same on its Cassandra and Neo4j backends (TrustGraph documentation and blog, trustgraph.ai, 2025; the vendor's own account, not independently measured)

The context graph approach differs from general-purpose graph databases in its design priorities: rather than optimising for ad-hoc queries, it optimises for feeding structured context into LLM

prompts. This includes preserving structured formats (Cypher, RDF) in the context, because the structure itself carries semantic information that improves LLM responses.

4. Deterministic Knowledge Retrieval

This category applies knowledge graphs to **retrieval and validation**, matching queries against a curated vocabulary with deterministic precision.

Terraphim operates in this space. As described in Chapters 7 and 8, its approach:

- Compiles curated vocabularies into Aho-Corasick automata
- Matches query terms against the vocabulary in under a millisecond in the author's measurements (Chapter 7)
- Scores results by graph connectivity, the ranking method Chapter 7 calls the graph scorer: documents rank by the weight and depth of the graph paths that connect their matched terms to the query
- Returns the matched vocabulary term and graph path for each result, so every answer can be traced

The tradeoff is explicit: Terraphim requires upfront vocabulary curation but delivers deterministic, explainable retrieval with no per-query API cost; the local computation is sub-millisecond per query in the author's measurements. This makes it complementary to, rather than competitive with, the other categories.

Market Trends

The author's observations suggest four reasons teams are reconsidering their retrieval designs.

Trend 1: RAG Fatigue

Retrieval-Augmented Generation (RAG) fetches document chunks by embedding similarity and places them in the prompt before the model answers. The trend below is an observation, not a survey: it comes from the handful of enterprise programmes the author advised in 2024 and 2025 and from vendor material, which has an interest in the conclusion. In those programmes, teams that adopted basic RAG hit the same limitations as they scaled:

- **Precision problems**: "Semantically similar" does not mean "relevant." Vector similarity retrieves documents that mention related topics but may not answer the actual question.
- **No explainability**: When a RAG system returns poor results, there is no audit trail explaining *why* those documents were retrieved.
- **No validation**: The LLM can ignore retrieved documents entirely, or hallucinate despite having correct context.
- **Cost at scale**: Embedding computation and vector database queries add per-query costs that compound at enterprise scale.

These limitations have led the teams described above to consider knowledge graphs, ontologies and hybrid pipelines for more precise

retrieval and inspectable evidence paths. Chapter 22 describes the progression from RAG to GraphRAG (retrieval over an extracted graph) to OntologyRAG (retrieval constrained by a curated ontology for precision).

Trend 2: Regulation Driving Structured Knowledge

Regulatory frameworks are creating demand for explainable AI systems:

- **EU AI Act** (Regulation (EU) 2024/1689): Article 13 requires high-risk AI systems to be transparent enough for deployers to interpret and use their output, and Article 12 requires logging. A graph-grounded retrieval path gives an operator an audit trail that a similarity score alone does not.
- **FDA guidance** (*Clinical Decision Support Software*, final guidance, September 2022): developers of clinical decision software are expected to document how a recommendation was reached; graph-grounded systems can show the vocabulary match and path for each output.
- **Financial regulation** (for example the Bank of England PRA supervisory statement SS1/23, *Model risk management principles for banks*, 2023): firms using models in trading and compliance are asked by their regulators to explain outputs; deterministic retrieval with a logged match trace is one way to meet that request.

Regulation is adding “can you explain it” to “does it work” as a purchasing criterion. A graph answer helps because it carries its own trace, the matched term and the path that connected it, which an operator can show to a reviewer; the trace is supporting evidence for transparency and auditability, not a complete compliance control, and the obligations in the instruments above go well beyond it.

Trend 3: Hybrid Architectures

The hybrid architectures considered in this book combine multiple retrieval methods:

- **Deterministic search** for known vocabulary: fast, precise, free
- **Graph traversal** for relationship queries: follows connections, no API cost
- **Semantic search** for novel queries: flexible, handles paraphrasing
- **LLM-based extraction** for building graphs from unstructured data

Each of the four approaches above has limits. For a regulated or high-volume deployment, assess whether combining methods and routing by query type addresses those limits at an acceptable cost. Chapter 22 describes these hybrid pipeline patterns in detail.

Trend 4: The Semantic Web, Revisited

The trend the author least expected is the revival of semantic web concepts. RDF (the W3C’s triple format for stating facts as subject, predicate, object), OWL (its ontology language for defining classes

and constraints) and SKOS (its vocabulary for thesauri and controlled terms) date from the 2000s semantic web movement, which promised machine-readable interoperability and stalled on the cost of writing everything by hand. Language models can read and write those formats, which reduces the authoring and translation cost that stalled them; governance, validation and ontology alignment remain as expensive as they were.

Two newer protocols pursue the same interoperability goal from the agent side. MCP (Model Context Protocol) standardises how a model reaches tools and data sources; A2A (Agent-to-Agent) standardises how agents exchange tasks and results. Whether they succeed where the semantic web struggled remains an open question.

Evaluation Framework

When selecting knowledge graph technology for context engineering, consider these dimensions:

Dimension	Questions to Ask
Curation model	Manual vocabulary? Automated extraction? Hybrid?
Query determinism	Same query always returns same results?
Explainability	Can you trace why each result was returned?
Update latency	How quickly does the graph reflect new knowledge?
Scale	Thousands of entities? Millions? Billions?
Integration	Does it plug into your existing LLM pipeline?
Cost model	Per-query costs? Fixed infrastructure?

Dimension	Questions to Ask
Regulatory fit	Does it satisfy your industry's explainability requirements?

To turn the eight answers into a decision, score each dimension 0 to 5 against anchors and weight the scores. Anchors for the four heaviest dimensions: explainability 0 means no trace at all, 3 means a trace exists but needs an engineer to read it, 5 means every result carries a trace a non-engineer reviewer can follow; query determinism 0 means results vary run to run, 3 means stable for a pinned configuration, 5 means byte-identical by construction; regulatory fit 0 means nothing to show an auditor, 3 means logs that could be assembled into evidence, 5 means evidence produced as a by-product of every query; curation model 0 means the vendor's extraction is the only source of truth, 3 means extraction with review, 5 means a curated vocabulary with owners. Then weight them. The author's default weights, which a regulated deployment should raise and an internal tool may lower: explainability and regulatory fit 0.2 each; query determinism and curation model 0.15 each; integration, cost model, update latency and scale 0.075 each. Score each product 0 to 5 per dimension, multiply, sum, and treat any product scoring below 2 on a dimension weighted 0.2 as excluded regardless of total. A hospital diagnostic assistant and a startup chatbot will rank the same four categories in different orders under this procedure, which is the point of making the weights explicit.

A worked example, scored by the author for a regulated clinical assistant with the default weights. It is an illustrative judgement, not

a benchmark.

Dimension	Weight	Neo4j as a bare graph database	Rationale	Contribution
Explainability	0.20	2	Stores the graph; produces no reviewer-readable trace by itself	0.40
Regulatory fit	0.20	2	Evidence must be assembled from application logs	0.40
Query determinism	0.15	2	Depends entirely on what is built on top	0.30
Curation model	0.15	2	Neutral; the vocabulary is whatever is loaded	0.30
Integration	0.075	4	Mature drivers and ecosystem	0.30
Cost model	0.075	4	Fixed infrastructure, no per-query fee	0.30

Dimension	Weight	Neo4j as a bare graph database	Rationale	Contribution
Update latency	0.075	3	Immediate writes, but reindexing is the application's job	0.225
Scale	0.075	5	Billions of relationships in deployments the vendor documents (Key Sources 9)	0.375
Total				**2.60**

The two 2s on dimensions weighted 0.2 meet the stated minimum, so this example does not trigger exclusion. They still identify explainability and regulatory fit as areas where the application needs additional evidence and controls. Assess the database together with the proposed retrieval and grounding layers before treating the combined solution as adequate. Record your own candidate scores and their supporting evidence before comparing them with a vendor's assessment.

Summary

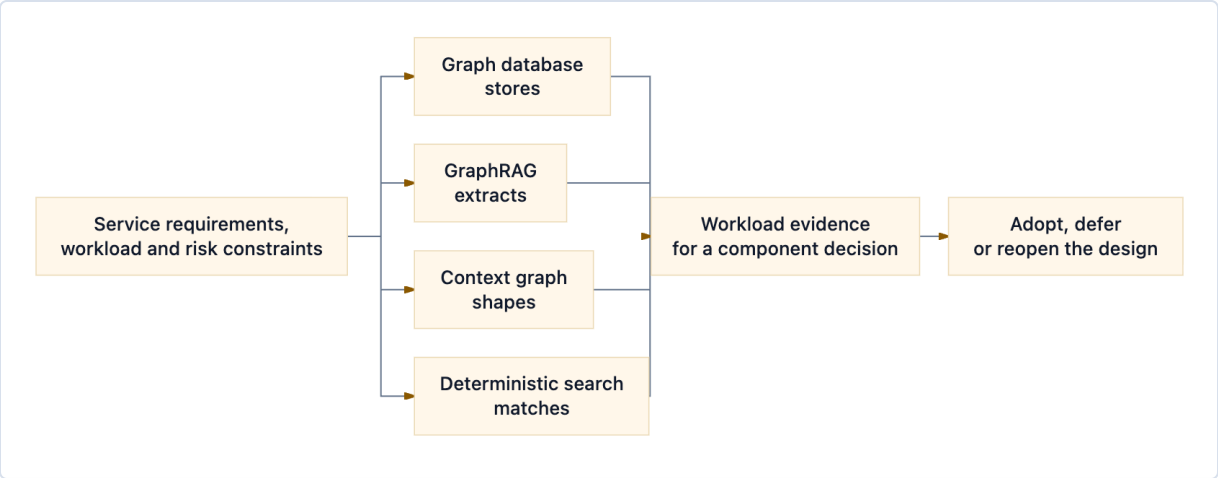


Figure 24.2: Chapter 24 recap: Graph databases, GraphRAG, context graph platforms and deterministic retrieval serve different parts of a combined stack.

Four forces move the market: RAG fatigue pushing enterprises toward structure, regulation requiring explainability, hybrid architectures combining retrieval methods, and the return of semantic-web standards as formats models can read. The four categories answer different parts of the resulting question. A graph database stores the graph; GraphRAG builds one from text when nobody can curate it; a context graph platform shapes the graph for a model to consume; deterministic retrieval checks and routes against a curated vocabulary. A production stack usually needs the first and one of the others, and Chapter 22's hybrid pipeline is how they are combined.

The output of selection is a justified component decision with workload evidence, costs and unmet conditions. It can also be a decision to defer or revise the design. Chapter 25 turns the unresolved conditions into testable questions; a market survey cannot supply missing evidence about service behaviour.

Key Sources

1. Microsoft Research, “From Local to Global: A Graph RAG Approach to Query-Focused Summarization” (Edge et al., arXiv 2404.16130, 2024), and the GraphRAG project documentation
2. DB-Engines ranking, graph DBMS category, db-engines.com (checked October 2026)
3. ISO/IEC 39075:2024, *Database languages: GQL*
4. TrustGraph documentation and engineering blog, trustgraph.ai (2025): deployment scale and backend equivalence as reported by the vendor
5. Regulation (EU) 2024/1689 (the EU AI Act), Articles 12 and 13
6. US Food and Drug Administration, *Clinical Decision Support Software*, final guidance, September 2022
7. Bank of England Prudential Regulation Authority, SS1/23, *Model risk management principles for banks*, May 2023
8. Terraphim AI implementation and the author’s measurements in Chapter 7
9. Neo4j documentation and published customer case studies on graph scale, neo4j.com (checked October 2026), as reported by the vendor

Chapter 25: Open Problems and Research Directions

This final chapter identifies open problems in the methods developed across the previous 24 chapters. The questions concern verification capacity, coherence across contexts and evidence for controlled improvement.

The Verification Bottleneck

Figure 25.1 maps the open problems this chapter covers and how they connect.

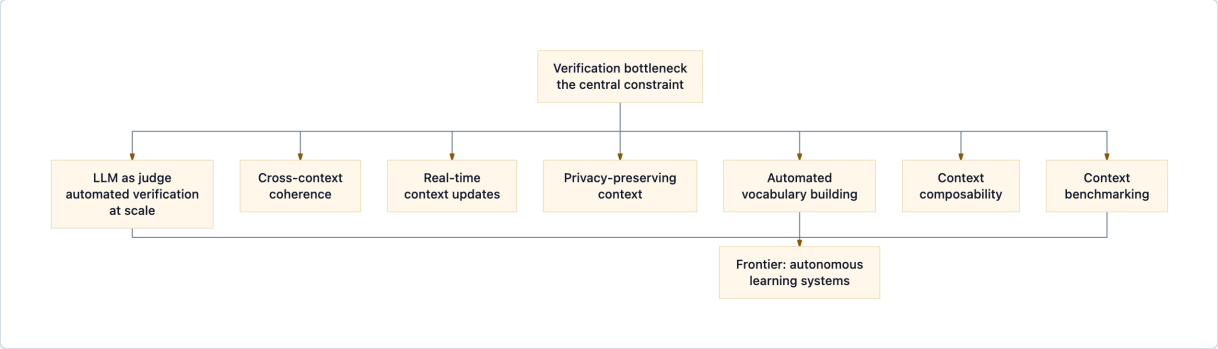


Figure 25.1: The open problems of this chapter, all downstream of the verification bottleneck.

This chapter starts with **verification**: how to check growing volumes of generated code and content with limited review capacity.

The Missing Junior Loop

Junior engineers develop judgement through work such as writing code, running tests and fixing bugs, with senior engineers reviewing their output. AI agents increasingly perform those tasks, raising a question about how teams will preserve the review experience through which juniors learn.

But the review cycle that trained seniors has no equivalent. When a junior engineer writes code, a senior can review it and provide feedback. When an AI generates 1,000 lines of code in an hour, no human can review it at that pace. The “junior loop” that created expertise is breaking.

This creates a structural problem: who trains the next generation of senior engineers if the junior work is automated? And who verifies the AI’s output if the senior pipeline thins?

The Codifier’s Curse

When you codify expertise into AI systems, you lose the tacit knowledge that was never explicitly documented. The system appears to work until it encounters edge cases where that tacit knowledge mattered. This is what this book calls the Codifier’s Curse: the act of automating expertise can destroy the conditions that created it.

The knowledge graphs, vocabularies and validation rules described in this book represent codified expertise. If their builders leave without training replacements, the system may continue working while the team loses the ability to explain or safely change it.

The Economics of Verification

The core economic insight, from Catalini, Hui and Wu (2026), is that once execution is abundant the binding constraint on growth is human verification bandwidth: the capacity to validate, audit and take responsibility for what was produced. The author's inference from that, and from the systems described in this book, is that verification cost grows faster than generation cost as agents are deployed; the inference is not a measurement. Consider the progression:

1. AI generates code faster than anyone reads it; a thousand lines in an hour is an illustrative order of magnitude, not a measurement
2. Careful human review of that much unfamiliar code takes days
3. Pressure to ship leads to superficial verification
4. Subtle bugs accumulate as "hidden technical debt"

This is the structure Chapter 1 called the AI sandwich: AI does the measurable generation, humans do the unmeasurable verification, AI does the final polish. The middle layer does not scale.

At some point it becomes economically rational to deploy unverified AI outputs. The result is hidden systemic risk: failures that are statistically invisible until they are catastrophic.

Measurability Bias

"Measurability-biased" progress describes a preference for automating work whose results are easy to check. That can direct

investment towards measurable tasks while leaving harder-to-measure work, including domain alignment and maintainability, under-resourced.

For context engineering, this means: the parts of context management that are easy to measure (token counts, retrieval accuracy, response latency) get optimised, while the parts that are hard to measure (conceptual coherence, domain alignment, long-term maintainability) get ignored.

Goal-Driven Execution as Partial Solution

One approach to scaling verification: build it into the generation loop. Rather than generating first and verifying later, define verifiable success criteria before generation begins:

- "Add validation" becomes "Write tests for invalid inputs, then make them pass"
- "Fix the bug" becomes "Write a test that reproduces it, then make it pass"
- "Refactor X" becomes "Ensure tests pass before and after"

These criteria make the task and its checks explicit, reducing the material a reviewer must assess without a stated reference. Defining adequate success criteria still requires expertise.

Structural Verification: Making the Spec Small Enough to Prove

A different response to the verification bottleneck is to shrink what needs verifying rather than scale who verifies it. Datadog's systems research team is building **Temper** (public repository github.com/nerdsane/temper; the description below follows its README as of October 2026, and the conclusions drawn from it are the author's), an application substrate composed from six deliberately unoriginal primitives:

- **OData**, a standard for describing typed entities and their relationships, so that the data model is self-describing
- **I/O automata**, explicit state machines that say which transitions are legal, for behaviour
- **The actor model**, isolated units that communicate by messages, for execution
- **Cedar**, a policy language, for deciding who may do what
- **Wasm** (WebAssembly), for sandboxing untrusted code
- **Rust**, as the implementation language

Primitive	Role	What it constrains
OData	Typed, self-describing entities and relationships	Data
I/O automata	Explicit state machines	Behaviour
Actor model	Isolated units communicating by message	Execution
Cedar	Policy language	Access

Primitive	Role	What it constrains
Wasm	Sandboxed execution of untrusted code	Isolation
Rust	Implementation language	Memory safety of the runtime

Two terms from formal methods appear below. An *SMT solver* takes a logical statement about a system and either finds values that satisfy it or proves none exist; a *model checker* explores every reachable state of a state machine and reports any that violate a stated property. Both work only when the thing being checked is small and precisely described, which is the point of the six primitives.

The architectural proposal combines established primitives so that an application can be described as entities, state machines and policies. That description makes the relationships between the data model, workflow and authorisation explicit. Its small, precise specification is the proposed target for SMT solvers, model checkers and theorem provers, reducing the amount of application logic subject to each proof. The runtime itself is tested under deterministic simulation as well as unit tests.

In this proposal, the runtime enforces two checks on an agent's proposed state change: the automaton checks whether the transition is valid, and the Cedar policy checks whether the actor is authorised to make it. This gives Chapter 14's verification gate a bounded object to assess: a typed transition and its applicable policy. People still need to review the model and policy that define the permitted behaviour.

The trade-off is scope: Temper's approach applies to application logic expressible as typed entities and finite-state behaviour. It leaves judgement calls that resist formalisation, including Chapter 21's governance concerns, outside that proof boundary. State transitions over a typed domain model include record updates, workflow transitions and permission changes; for these, the proposal assigns checks to the compiler and solver before execution. As of this writing Temper is early-stage systems research with a public repository and no released product. No independent evaluation of it has been performed for this book.

LLM as a Judge: What Is Still Open

Chapter 14 set out how to use a language model as a judge and what its calibration showed: Cohen's kappa (agreement corrected for chance) over raw agreement, negatives over examples, binary questions over ordinal scores, and fail closed. Those are solved problems in the sense that the practice is known even where it is not followed. Three problems are not solved; a fourth subsection records a finding about self-correction that bears on all of them.

Self-Preference in the Reference

Every calibration needs a reference judgement, an *oracle* in the language of Chapter 23, and the cheapest reference is a frontier model. If the material being judged was largely written with the help of the same model family, the reference inherits the family's preferences and every kappa measured against it is measured

against a tilted ruler. The calibration in Chapter 14 has exactly this exposure: the oracle and much of the corpus share a lineage, and nobody has yet measured how much that matters. The experiment is cheap, a second oracle from a different family or a human label on the same files and the oracle-to-oracle kappa, and almost nobody runs it.

Calibrated Confidence

A judge that reports a confidence of 0.9 should be right about nine times in ten. Of the judge systems and vendor judge products whose public materials were reviewed for Chapters 14 and 23, none reports whether that is so; the claim is about those materials, not the field. The measurement is expected calibration error over a reliability diagram (Naeini, Cooper and Hauskrecht, 2015), standard in classification for a decade, and the research route to improving it is reinforcement learning with a calibration term in the reward. Until a judge's confidence is itself calibrated, confidence thresholds in a gate ("escalate below 0.7") are policy dressed as measurement.

Compare Instead of Score, Inside the Loop

Pairwise comparison had the highest agreement with human preference labels in MT-Bench's comparison of judging modes (Zheng et al., 2023). Whether that finding holds for agent-trace judging is untested. A pipeline could generate several candidates for a bounded task and compare them before accepting one, using Chapter 14's position swaps to check for order effects. The benefit and cost need measuring on the pipeline's own tasks.

Metacognitive Self-Correction

FINAL-Bench (Hugging Face, 2025), a benchmark of multi-step agent tasks that separates whether a model notices its own errors from whether it acts on them, reports a counterintuitive decomposition: in its task set, **94.8 per cent of the performance gain between weaker and stronger agents came from error recovery (acting on detected errors), not from metacognitive accuracy (correctly identifying uncertainty levels)**. The figure is the benchmark authors' own decomposition on their own tasks and has not been reproduced for this book.

The reported decomposition suggests testing whether acting on detected errors improves the verification workflow. Compare correction attempts with leaving the candidate unchanged, including cases where the detected error is a false alarm. Their relative cost and harm depend on the task; the benchmark figure alone cannot establish when correction is preferable.

Cross-Context Coherence

Maintaining semantic alignment across sessions, agents and time is the first unsolved problem. When multiple AI agents work on the same project, each operates within its own context window. Agent A may establish a convention in session 1 that Agent B contradicts in session 2. Over time, the accumulated context across sessions becomes incoherent.

Current approaches (shared CLAUDE.md files, handover documents, memory systems) are ad-hoc. No theory in common use among agent toolchains says how to compose contexts from multiple sources while maintaining coherence; adjacent formal work on knowledge-base merging and belief revision exists and has not crossed over. Key questions:

- How do you detect when two agent sessions have established contradictory conventions?
- What is the "merge conflict" equivalent for context, and how do you resolve it?
- Can context coherence be measured, or is it inherently subjective?

A tractable experiment: record the conventions each session establishes as entries in the decision log of Chapter 18, then run a deterministic check for pairs of entries that assign different values to the same key. That detects the detectable cases; the contradictions phrased differently in each session are the research problem.

Real-Time Context Updates

The second problem is hot-reloading a knowledge graph without breaking coherence. When a knowledge graph is updated while agents are actively using it, several problems arise:

- Existing context may reference entities or relationships that have been modified

- Cached query results become stale
- In-flight operations may see an inconsistent graph state

This is analogous to database consistency under concurrent writes, but with an additional complication: the “clients” (LLMs) have long-lived context windows that cannot be atomically updated. The nearest working practice is versioning: the graph carries a version, every retrieved fragment carries the version it came from, and a session that mixes versions is flagged rather than silently continued. What is open is what to do with the flagged session other than restart it.

Privacy-Preserving Context

The third problem is using personal or sensitive context without exposing it. User preferences, organisational knowledge and proprietary data may improve an answer’s relevance, while their exposure creates privacy and security risks. The question is which context is necessary and how its use can be controlled.

Partial approaches exist: redacting personal identifiers before retrieval, differential privacy over the corpus an index is built from, and running the model inside a trusted execution environment so the context never leaves it. None answers the open questions:

- Can context be injected into prompts in a way that prevents the LLM from reproducing it verbatim?

- How do you audit what context an LLM “learned” from a conversation vs what it already knew?
- Which of the techniques above survive contact with a model that can be prompted to quote its context?

Automated Vocabulary Building

Manually building domain vocabularies does not scale. The vocabularies described in Chapter 8 require domain experts to define terms, synonyms, and relationships. For large or rapidly evolving domains, this bottleneck limits adoption.

Research directions:

- **LLM-assisted extraction**: Use LLMs to extract candidate vocabulary from domain documents, with human validation of the results
- **Incremental learning**: Start with a seed vocabulary and expand it as the system encounters new terms in use
- **Cross-domain transfer**: Adapt vocabularies from related domains rather than building from scratch

The challenge is quality. The author’s observation, from two automatically extracted vocabularies that were later curated by hand, is that the errors in an extracted vocabulary are trusted as if they were curated, so a mostly-right vocabulary can do more harm than none; two cases are an observation, not a rate, and the proportion at which the harm begins has not been measured.

Context Composability

Context composability is the operational face of cross-context coherence: combining contexts from multiple sources into a coherent whole. A practical AI system needs to compose: - User-specific context (preferences, history) - Project-specific context (domain model, conventions) - Organisation-specific context (policies, standards) - Tool-specific context (API documentation, configuration)

Currently, this composition is ad-hoc: contexts are concatenated, hoping the LLM will sort out conflicts. A formal composition algebra would define:

- **Precedence rules**: When contexts conflict, which wins?
- **Scope boundaries**: Which context applies to which queries?
- **Inheritance**: Can child contexts override parent contexts?

Context Benchmarking

We need **standard benchmarks** for context quality. Current evaluation is subjective ("the AI seems to give better answers with this context"). Rigorous benchmarks would measure:

Metric	Definition	Current State
Retrieval accuracy	Percentage of relevant documents retrieved	Well-measured for RAG

Metric	Definition	Current State
Grounding rate	Percentage of claims verified against KG	Emerging
Drift detection speed	Time to detect meaningful context drift	No standard
Context efficiency	Quality improvement per token of context	No standard
Composability	Quality after merging multiple contexts	No standard

Without benchmarks, two teams that each report "grounding improved quality" cannot be compared, and the author has not been able to compare his own systems across years for the same reason.

The Frontier: Towards Autonomous Learning

One research direction is systems that improve their own context over time: reingesting outputs with provenance metadata and adjusting retrieval strategies as evidence accumulates.

This frontier involves:

- **Self-describing information stores**: Systems carrying metadata about their own structure, enabling adaptive retrieval without human configuration
- **Dynamic retrieval strategies**: LLMs deriving retrieval strategies for unseen information types, rather than relying on pre-configured pipelines

- **Temporal reasoning**: Weighing repeated observations over decades against recent unconfirmed data, distinguishing freshness from accuracy

In the open-source GraphRAG implementations the author reviewed, community detection was common while other graph algorithms were used less. Chapter 22 suggests experiments with traversal depth, clustering, density and outlier detection. Their effect on retrieval quality needs comparison against the existing pipeline.

What You Can Do Now

Five steps a team can take this quarter:

1. Add a pairwise comparison to one agent pipeline: generate two candidates for a bounded task and let a calibrated judge choose, in both orders
2. Start recording the grounding rate, the share of claims the gate could check, so that benchmarking can begin from your own numbers
3. Record the resolved model version and the judge identity on every verdict, so that recalibration has something to tie to
4. Audit which of your context sources have an owner, a version and a check (Chapter 2's table), and treat the blanks as the composability problem arriving early
5. Pick one vocabulary and measure how many of its terms were added by extraction versus curation; that ratio is your exposure

to the automated-vocabulary problem

Confidence Under Distribution Shift

The out-of-distribution follow-up to the author’s exp-002 routing experiment (E4, Appendix I) reports a fresh 100-case set across eight unfamiliar domains. The gated router achieves 70% accuracy against the baseline’s 22%, while its reported mean Brier is slightly worse: 0.2475 against 0.2388. Compared with its in-distribution Brier of 0.0158, the gated score worsens approximately 15.7-fold. These are author-reported figures whose underlying artefacts were unavailable for editorial reproduction.

Brier combines calibration and resolution; a reversed Brier ranking alone does not establish calibration inversion. The open problem is to identify which confidence signals remain useful under changed inputs. Publish the scored event, class distribution and reliability analysis before interpreting the ratio as a calibration loss. Test gate paraphrases, non-native phrasing and compound requests separately. Chapter 16 explains why unfamiliar inputs may warrant abstention even when a response has a valid type.

Inspect Which Routing Capability Failed

The same follow-up (E4, Appendix I) also reports 33% compound exact-set correctness and erroneous decisions carrying confidence from 0.85 to 0.95. The compound subset denominator and confidence-event definition are not supplied in the inspected diff, so

these figures cannot support a complete uncertainty estimate or calibration diagnosis. They identify useful follow-up slices: operation coverage, paraphrase sensitivity, negative rejection and confident error. Evaluate each against the Chapter 12 routing requirement rather than treating overall accuracy as a sufficient acceptance criterion.

Selecting Among Improving Branches

A system can maintain several candidate branches with different mutation strategies and select among them per task. This introduces a second evaluation problem: the selector may fail even when an appropriate branch exists. Selection needs its own held-out cases, abstention policy and versioned candidate catalogue. The branch-router advantage reported in the author's experiments (Appendix I) remains a research lead here; the numerical claim is omitted pending its primary source and evaluation protocol.

The maintainer's concern is admitting an apparently improved branch that damages other tasks. Requirements should allocate protected-case assessment to the evaluation service and exact-revision approval to the promotion service. Retain rejected branches and their results in the lineage record. Chapter 15 supplies the harness; Chapter 17 supplies the links needed to inspect branch selection later.

For a population of improvement branches, investigate diversity versus redundancy, evaluation budget, changing candidate sets and

lineage at merge time. A selector calibrated on yesterday's branches may face a different answer space after mutation. Freeze the branch catalogue for each comparison and assess the selector independently. The reported advantage remains an unverified research lead; it should not be used to justify automatic promotion.

Maintaining an Unattended Practice

The author's deployment records (Appendix I) report a 63-day interval between human commits alongside repetitive governance reports and unresolved work. A commit gap does not establish a complete absence of human attention, and the observations do not prove that absence caused the reporting failure. They do motivate a practical question: how does the service owner detect when scheduled reporting continues but commitments remain overdue?

Review evidence-backed issue closure, lesson application opportunities and unresolved decisions alongside report production. These measures also need protection against gaming: closing an issue without satisfying its requirement should not improve the practice-health result. Define the owner, escalation deadline and permitted action when supervision is unavailable. The concern is neglected obligations; the allocated coordination service needs to expose overdue commitments and preserve their evidence. Chapter 23 discusses ownership, while Chapter 18 discusses durable handoffs. Figure 25.2 connects these questions to evaluation and governance.

For unattended operation, distinguish a safe escalation deadline from emergency authority. A break-glass policy needs a named authoriser, permitted actions, expiry, evidence requirements and subsequent review. Repeated overdue reports should trigger that policy's specified response, not let the agent invent new permissions. Test whether health metrics can be improved by closing the wrong issues or recording lessons without an opportunity to apply them.

Summary

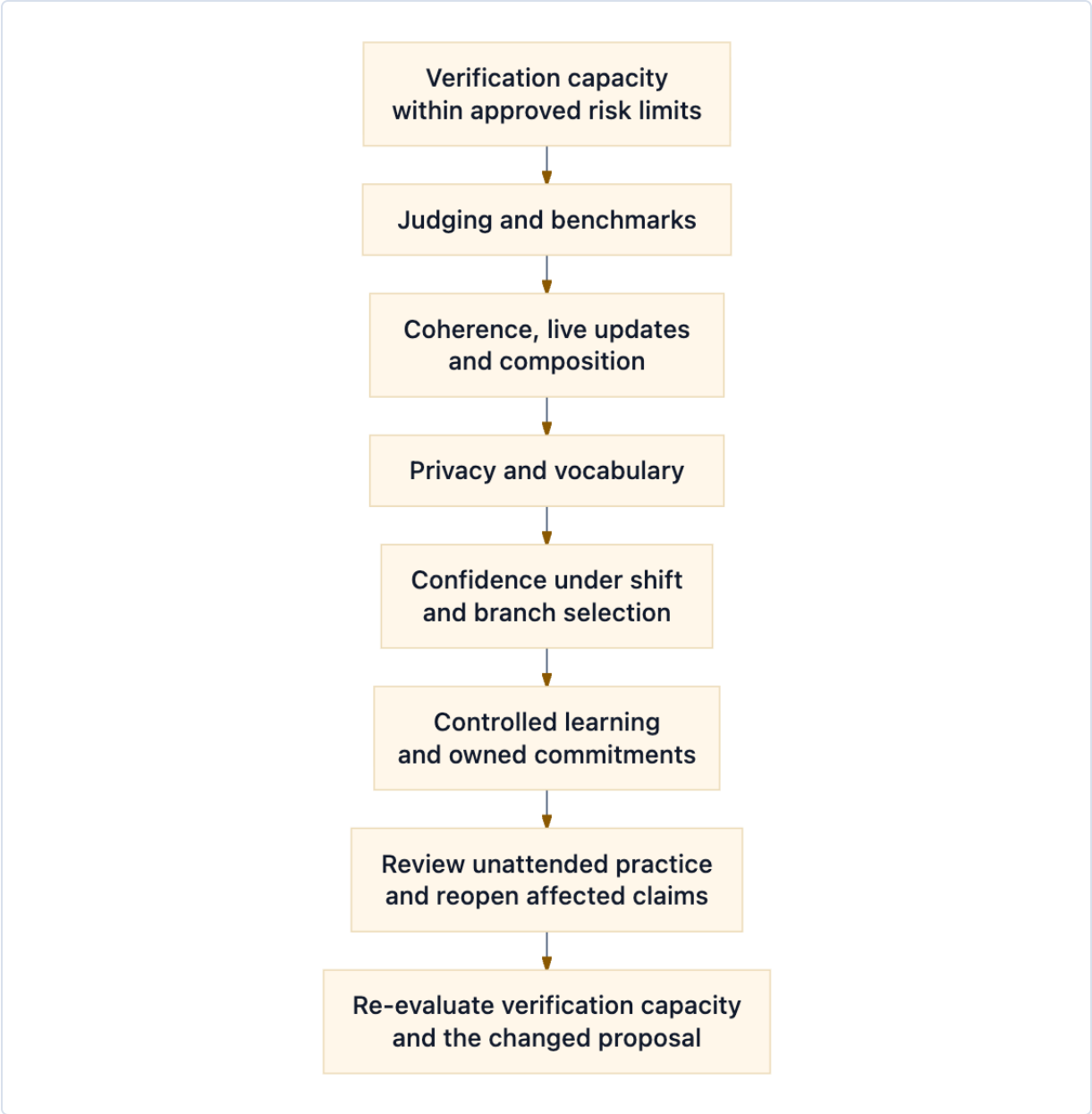


Figure 25.2: Chapter 25 recap: Verification capacity connects confidence under shift, branch selection and unattended commitments to the wider research agenda.

The verification bottleneck is the umbrella under which the rest of this chapter sits; the eleven problems beneath it are:

1. **Judging at scale**: who checks the AI's work, with what calibration, and how a judge's own confidence is measured

2. **Cross-context coherence**: Multiple agents, sessions, and time – how to keep context consistent?
3. **Real-time updates**: Changing knowledge graphs without breaking in-flight operations
4. **Privacy in context**: Using personal data without exposing it
5. **Automated vocabulary**: Building domain vocabularies without requiring full-time experts
6. **Context composability**: Merging multiple context sources with formal rules
7. **Benchmarking**: Standard metrics for comparing context engineering approaches
8. **Autonomous learning**: Systems that improve their own retrieval over time
9. **Confidence under shift**: Testing whether confidence remains useful on unfamiliar inputs
10. **Branch selection**: Evaluating the selector as well as the candidate improvements
11. **Unattended practice**: Detecting overdue commitments when reporting continues

Each research question needs a stated workload, comparison and failure criterion before a proposed improvement can become an established practice.

The synthetic maintenance case began with a request to restore one failed job. It exposed questions about meaning, reference quality,

context selection, protected behaviour, authority, delivery and shared-resource consequences. Each chapter supplied a result needed to inspect one of those questions. The story does not claim an end-to-end deployment: the original fixtures remain illustrative, and the real-system and research results retain their own scopes.

The final test is whether the service fulfils agreed requirements under current conditions, with acceptable consequences, and what evidence justifies changing it. Use observations to revisit the affected claims, including intended use and requirements when necessary. Preserve the current baseline until an authorised revision takes effect. Autonomous improvement remains bounded by approved risk limits and separately enforced authority; where the justification no longer holds, stop or return the proposal for accountable review.

Source for the added operational cases: Experiments E3 and E4 (Appendix I), the exp-002 skill-router evaluation and its out-of-distribution follow-up, with the author's research digest of 8 October 2026; the numerical results have not been independently reproduced for this edition.

Key Sources: Catalini, Hui and Wu, "Some Simple Economics of AGI" (arXiv 2602.20946, 2026) on verification bandwidth, Karpathy LLM coding guidelines, FINAL-Bench metacognitive research, TrustGraph context graph progression, LLM-as-Judge best practices literature, Naeini, Cooper and Hauskrecht (2015) on expected calibration error, "Beyond Binary Rewards: Training LMs to Reason About Their Uncertainty" (RLCR, arXiv 2507.16806) and "Rewarding Doubt" (arXiv 2503.02623) on calibration-aware training, FINAL-Bench (Hugging

Face, 2025), Temper (Datadog systems research, github.com/nerdsane/temper) on structural verification via typed entities + I/O automata + Cedar policy

Appendix A: Context Optimiser Skill (SKILL.md)

This appendix provides a reference implementation for a **context optimiser skill** – a self-contained module that manages token budgets, skill activation, and compression within an AI agent system.

The skill follows the architecture described in Chapter 12 and the token budget principles from Chapter 10.

SKILL.md Format

Every skill in a context-engineered system is defined by a `SKILL.md` file. This is both human-readable documentation and machine-parseable configuration.

```
# Context Optimizer

## Identity
- Name: context-optimizer
- Tier: core
- Version: 1.0.0

## Triggers
- context
- tokens
- budget
- compress
- optimize
```

```
## Description
Manages token budget allocation, skill activation/deactivation,
and context compression. Runs automatically when context usage
exceeds warning thresholds.

## Dependencies
- None (this skill is self-contained)

## Configuration
- max_active_skills: 5
- token_warning_threshold: 12000
- token_critical_threshold: 15000
- auto_deactivate_after: 10  # turns

## Tools Required
- File read (for loading skill content)
- Token counter (for budget tracking)
```

Reference Implementation

Skill Loader

The skill loader reads `SKILL.md` files and parses them into a structured format:

```
function parseSkillMd(content) {
  const sections = {};
  let currentSection = null;

  for (const line of content.split("\n")) {
    if (line.startsWith("## ")) {
      currentSection = line.slice(3).trim().toLowerCase();
      sections[currentSection] = [];
    } else if (currentSection && line.trim()) {
      sections[currentSection].push(line.trim());
    }
```

```
  }

  return {
    name: extractField(sections, "identity", "Name"),
    tier: extractField(sections, "identity", "Tier"),
    version: extractField(sections, "identity", "Version"),
    triggers: extractList(sections, "triggers"),
    description: sections["description"]?.join(" ") ?? "",
    dependencies: extractList(sections, "dependencies"),
    config: extractKeyValues(sections, "configuration")
  };
}

function extractField(sections, section, field) {
  const line = sections[section]?.find(l => l.startsWith(`-
${field}:`));
  return line?.split(":").slice(1).join(":").trim() ?? null;
}

function extractList(sections, section) {
  return (sections[section] ?? [])
    .filter(l => l.startsWith("- "))
    .map(l => l.slice(2).trim());
}

function extractKeyValues(sections, section) {
  const config = {};
  for (const line of sections[section] ?? []) {
    if (line.startsWith("- ") && line.includes(":")) {
      const [key, ...rest] = line.slice(2).split(":");
      const value = rest.join(":").trim();
      config[key.trim()] = isNaN(value) ? value : Number(value);
    }
  }
  return config;
}
```

Context Optimiser Core

The optimiser manages the full lifecycle: activation, budget tracking, compression, and deactivation.

```
class ContextOptimizer {
  constructor(skillCatalog, config = {}) {
    this.catalog = skillCatalog;
    this.activeSkills = new Map();
    this.turnCounter = 0;

    this.maxActive = config.max_active_skills ?? 5;
    this.warnAt = config.token_warning_threshold ?? 12000;
    this.criticalAt = config.token_critical_threshold ?? 15000;
    this.deactivateAfter = config.auto_deactivate_after ?? 10;
  }

  // Main entry point: called each turn
  async optimizeContext(userQuery, currentContext) {
    this.turnCounter++;

    // 1. Determine which skills should be active
    const candidates = this.findCandidates(userQuery);

    // 2. Activate new skills (up to limit)
    for (const candidate of candidates) {
      if (!this.activeSkills.has(candidate.name)) {
        await this.activate(candidate);
      } else {
        this.activeSkills.get(candidate.name).lastUsedTurn =
this.turnCounter;
      }
    }

    // 3. Deactivate stale skills
    this.deactivateStale();

    // 4. Enforce limit
    this.enforceLimits();
```

```
    // 5. Check budget
    const budget = this.calculateBudget(currentContext);

    if (budget.total > this.criticalAt) {
      this.compress(currentContext);
    }

    return {
      activeSkills: [...this.activeSkills.values()],
      budget,
      compressed: budget.total > this.warnAt
    };
  }

  findCandidates(query) {
    const q = query.toLowerCase();
    return this.catalog
      .filter(skill => skill.triggers.some(t => q.includes(t)))
      .sort((a, b) => {
        const tierOrder = { core: 0, standard: 1, optional: 2 };
        return (tierOrder[a.tier] ?? 2) - (tierOrder[b.tier] ?? 2);
      });
  }

  async activate(skill) {
    const content = await loadSkillContent(skill.name);
    this.activeSkills.set(skill.name, {
      ...skill,
      content,
      tokens: estimateTokens(content),
      activatedTurn: this.turnCounter,
      lastUsedTurn: this.turnCounter
    });
  }

  deactivateStale() {
    for (const [name, skill] of this.activeSkills) {
      if (this.turnCounter - skill.lastUsedTurn >
this.deactivateAfter) {
        this.activeSkills.delete(name);
```

```
      }
    }
  }

  enforceLimits() {
    if (this.activeSkills.size <= this.maxActive) return;

    const sorted = [...this.activeSkills.entries()]
      .sort(([, a], [, b]) => {
        const tierOrder = { core: 0, standard: 1, optional: 2 };
        const tierDiff = (tierOrder[a.tier] ?? 2) - (tierOrder[b.tier]
?? 2);
        if (tierDiff !== 0) return tierDiff;
        return b.lastUsedTurn - a.lastUsedTurn;
      });

    // Keep top N, remove the rest
    const toRemove = sorted.slice(this.maxActive);
    for (const [name] of toRemove) {
      this.activeSkills.delete(name);
    }
  }

  calculateBudget(context) {
    const constitutional = context.constitutionalTokens ?? 2000;
    const catalog = estimateTokens(JSON.stringify(this.catalog));
    const skills = [...this.activeSkills.values()]
      .reduce((sum, s) => sum + s.tokens, 0);
    const memory = context.memoryTokens ?? 0;
    const working = context.workingTokens ?? 0;

    return {
      constitutional,
      catalog,
      skills,
      memory,
      working,
      total: constitutional + catalog + skills + memory + working
    };
  }
```

```
  compress(context) {
    // Priority: drop optional skills first, then summarise memory
    const optional = [...this.activeSkills.entries()]
      .filter(([, s]) => s.tier === "optional");

    for (const [name] of optional) {
      this.activeSkills.delete(name);
    }
  }
}
```

Usage Example

```
// Load catalog at startup (~200-300 tokens)
const catalog = await loadSkillCatalog("skills/catalog.yaml");

// Create optimizer
const optimizer = new ContextOptimizer(catalog);

// Each turn
const result = await optimizer.optimizeContext(
  userQuery,
  { constitutionalTokens: 2000, memoryTokens: 1500, workingTokens: 800
}
);

console.log(`Active: ${result.activeSkills.map(s => s.name).join(",
")}`);
console.log(`Budget: ${result.budget.total} tokens`);
```

File Structure

A complete context optimiser skill lives in a single directory:

Figure A.1 shows the skill package layout.

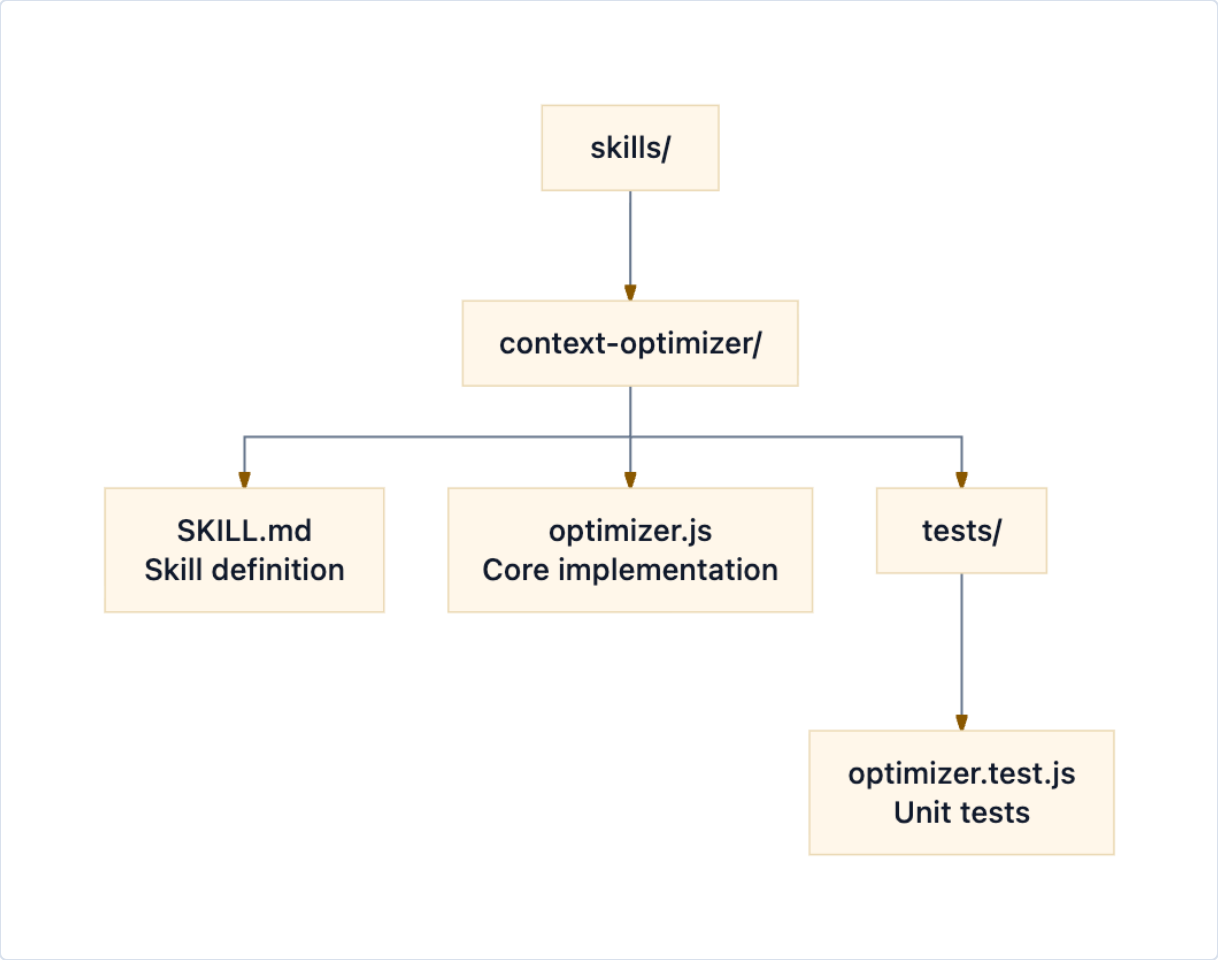


Figure A.1: The context-optimizer skill directory contains its definition, implementation and a tests directory holding optimizer.test.js.

Key Sources

- Chapter 10: Token Budget Architecture (budget allocation)
- Chapter 11: Compression Strategies (compression priority order)
- Chapter 12: Skill Architecture for AI Agents (catalogue, activation, deactivation)
- OpenClaw context management (production patterns)

Appendix B: Token Tracker Implementation

This appendix provides a reference implementation for a **token tracking system** that monitors context composition, enforces budgets, and triggers compression when thresholds are exceeded.

The design follows the token budget architecture described in Chapter 10.

Token Estimation

Before tracking tokens, you need to estimate them. Exact tokenisation requires model-specific tokenisers, but a practical approximation works for budget management:

```
// Approximate token estimation
// English text averages ~4 characters per token
// Code averages ~3.5 characters per token
// YAML/JSON averages ~3 characters per token
function estimateTokens(text, type = "text") {
  const charsPerToken = {
    text: 4,
    code: 3.5,
    yaml: 3,
    json: 3
  };

  const ratio = charsPerToken[type] ?? 4;
  return Math.ceil(text.length / ratio);
}
```

```
// For precise counting, use a tokeniser library
// import { encode } from "tiktoken";
// function countTokens(text) { return encode(text).length; }
```

Token Tracker

The tracker maintains a running count of tokens by category and raises alerts when thresholds are crossed.

```
class TokenTracker {
  constructor(config = {}) {
    this.budget = {
      constitutional: 0,
      skillCatalog: 0,
      activeSkills: 0,
      memory: 0,
      working: 0,
      buffer: 0
    };

    this.limits = {
      warn: config.warnAt ?? 12000,
      critical: config.criticalAt ?? 15000,
      hard: config.hardLimit ?? 20000
    };

    this.history = [];
    this.alerts = [];
  }

  // Update a category
  set(category, tokens) {
    if (!(category in this.budget)) {
      throw new Error(`Unknown budget category: ${category}`);
    }
```

```
    this.budget[category] = tokens;
    this.check();
  }

  // Increment a category
  add(category, tokens) {
    this.set(category, this.budget[category] + tokens);
  }

  // Get total usage
  get total() {
    return Object.values(this.budget).reduce((a, b) => a + b, 0);
  }

  // Get remaining budget
  get remaining() {
    return this.limits.warn - this.total;
  }

  // Get usage as percentage of warning threshold
  get usagePercent() {
    return Math.round((this.total / this.limits.warn) * 100);
  }

  // Check thresholds and raise alerts
  check() {
    const total = this.total;

    if (total > this.limits.hard) {
      this.alert("HARD_LIMIT", `Token count ${total} exceeds hard
limit ${this.limits.hard}`);
      return "hard_limit";
    }

    if (total > this.limits.critical) {
      this.alert("CRITICAL", `Token count ${total} exceeds critical
threshold ${this.limits.critical}`);
      return "critical";
    }
```

```
    if (total > this.limits.warn) {
      this.alert("WARNING", `Token count ${total} exceeds warning
threshold ${this.limits.warn}`);
      return "warning";
    }

    return "ok";
  }

  alert(level, message) {
    const entry = {
      level,
      message,
      total: this.total,
      breakdown: { ...this.budget },
      timestamp: Date.now()
    };

    this.alerts.push(entry);
  }

  // Record a snapshot for history
  snapshot() {
    this.history.push({
      total: this.total,
      breakdown: { ...this.budget },
      timestamp: Date.now()
    });
  }

  // Get a formatted breakdown
  report() {
    const lines = [];
    lines.push(`Token Budget Report (${this.usagePercent}% of warning
threshold)`);
    lines.push("—".repeat(50));

    for (const [category, tokens] of Object.entries(this.budget)) {
      const bar = "".repeat(Math.round(tokens / 200));
      const pct = Math.round((tokens / this.total) * 100);
```

```
      lines.push(`  ${category.padEnd(16)}
${String(tokens).padStart(6)} (${pct}%) ${bar}`);
    }

    lines.push("─".repeat(50));
    lines.push(`  ${"TOTAL".padEnd(16)}
${String(this.total).padStart(6)}`);
    lines.push(`  ${"Remaining".padEnd(16)}
${String(this.remaining).padStart(6)}`);

    return lines.join("\n");
  }
}
```

Budget Allocation Presets

Pre-defined allocations for common scenarios:

```
const BUDGET_PRESETS = {
  // Standard: balanced allocation for general use
  standard: {
    constitutional: 2000,
    skillCatalog: 100,
    activeSkills: 5000,
    memory: 2000,
    working: 2000,
    buffer: 900
  },

  // Skill-heavy: more room for active skills
  skillHeavy: {
    constitutional: 2000,
    skillCatalog: 200,
    activeSkills: 7000,
    memory: 1500,
    working: 1500,
    buffer: 800
```

```
  },

  // Memory-heavy: more room for conversation history
  memoryHeavy: {
    constitutional: 2000,
    skillCatalog: 100,
    activeSkills: 3000,
    memory: 4000,
    working: 2000,
    buffer: 900
  },

  // Minimal: tight budget for smaller models
  minimal: {
    constitutional: 1500,
    skillCatalog: 100,
    activeSkills: 3000,
    memory: 1000,
    working: 1000,
    buffer: 400
  }
};

function createTrackerWithPreset(preset) {
  const alloc = BUDGET_PRESETS[preset];
  if (!alloc) throw new Error(`Unknown preset: ${preset}`);

  const tracker = new TokenTracker({
    warnAt: Object.values(alloc).reduce((a, b) => a + b, 0),
    criticalAt: Math.round(Object.values(alloc).reduce((a, b) => a +
b, 0) * 1.25),
    hardLimit: Math.round(Object.values(alloc).reduce((a, b) => a + b,
0) * 1.5)
  });

  return tracker;
}
```

Compression Trigger

When the tracker raises an alert, the compression pipeline activates:

```
function onBudgetAlert(tracker, context) {
  const status = tracker.check();

  if (status === "warning") {
    // Soft compression: summarise old memory, drop optional skills
    context.memory = summariseOldTurns(context.memory, keepLast: 5);
    tracker.set("memory", estimateTokens(context.memory));
  }

  if (status === "critical") {
    // Hard compression: drop all non-core skills, aggressive
summarisation
    context.activeSkills = context.activeSkills.filter(s => s.tier ===
"core");
    context.memory = summariseOldTurns(context.memory, keepLast: 2);

    tracker.set("activeSkills", context.activeSkills.reduce(
      (sum, s) => sum + estimateTokens(s.content), 0
    ));
    tracker.set("memory", estimateTokens(context.memory));
  }

  if (status === "hard_limit") {
    // Emergency: constitutional only
    context.activeSkills = [];
    context.memory = extractKeyFacts(context.memory);
    context.working = truncateToEssentials(context.working);

    tracker.set("activeSkills", 0);
    tracker.set("memory", estimateTokens(context.memory));
    tracker.set("working", estimateTokens(context.working));
  }
}
```

Usage Example

```
const tracker = createTrackerWithPreset("standard");

// Set constitutional (never changes)
tracker.set("constitutional", 2000);
tracker.set("skillCatalog", 250);

// As skills activate
tracker.set("activeSkills", 3500);

// As conversation grows
tracker.set("memory", 1800);
tracker.set("working", 1200);

// Check status
console.log(tracker.report());
// Token Budget Report (73% of warning threshold)
// ──────────────────────────────────────────────────
//   constitutional    2000 (23%) ██████████
//   skillCatalog       250 ( 3%) ▌
//   activeSkills      3500 (40%) ████████████████
//   memory            1800 (21%) ████████
//   working           1200 (14%) ██████
//   buffer               0 ( 0%)
// ──────────────────────────────────────────────────
//   TOTAL             8750
//   Remaining         3250

// Take a snapshot for history
tracker.snapshot();
```

Key Sources

- Chapter 10: Token Budget Architecture (12,000 token rule, allocation breakdown)
- Chapter 11: Compression Strategies (priority order: dedupe, summarise, truncate, drop)
- Chapter 12: Skill Architecture (skill activation/deactivation lifecycle)

Appendix C: Skill Catalogue Schema

This appendix defines the complete schema for skill catalogues used in context-engineered AI agent systems. The schema covers the lightweight catalogue format (loaded at startup) and the full SKILL.md format (loaded on activation).

Catalogue Entry Schema (YAML)

The catalogue is the minimal representation loaded at startup, consuming approximately 200-300 tokens for 20 skills.

```yaml
# skill-catalog.schema.yaml
type: object
required: [skills]
properties:
  skills:
    type: array
    items:
      type: object
      required: [name, triggers, tier]
      properties:
        name:
          type: string
          description: "Unique skill identifier (kebab-case)"
          pattern: "^[a-z][a-z0-9-]*$"
          examples: ["web-search", "terraphim-memory", "file-editor"]

        triggers:
          type: array
          items:
            type: string
          minItems: 2
```

```
          maxItems: 10
          description: "Keywords that activate this skill"

        tier:
          type: string
          enum: [core, standard, optional]
          description: >
            Priority tier for activation and eviction.
            core: always preferred, last to evict.
            standard: normal priority.
            optional: first to evict under pressure.

        estimated_tokens:
          type: integer
          minimum: 100
          maximum: 5000
          description: "Expected token cost when fully loaded"

        dependencies:
          type: array
          items:
            type: string
          description: "Other skills required (loaded together)"

        auto_deactivate:
          type: integer
          minimum: 1
          default: 10
          description: "Turns of non-use before automatic eviction"

        compression_priority:
          type: integer
          minimum: 1
          description: "Drop order when budget is exceeded (1 = first
to drop)"
```

Example Catalogue

A production catalogue for an AI coding assistant:

```yaml
skills:
  # Core skills (always preferred)
  - name: file-editor
    triggers: ["edit", "modify", "change", "update", "write", "fix"]
    tier: core
    estimated_tokens: 600

  - name: code-runner
    triggers: ["run", "execute", "command", "bash", "shell",
"terminal"]
    tier: core
    estimated_tokens: 400

  - name: terraphim-memory
    triggers: ["memory", "recall", "remember", "kg", "knowledge",
"search"]
    tier: core
    estimated_tokens: 1200

  # Standard skills
  - name: web-search
    triggers: ["search", "web", "find", "lookup", "google"]
    tier: standard
    estimated_tokens: 800

  - name: git-operations
    triggers: ["git", "commit", "branch", "merge", "push", "pull"]
    tier: standard
    estimated_tokens: 700

  - name: database-query
    triggers: ["database", "sql", "query", "table", "schema"]
    tier: standard
    estimated_tokens: 900

  - name: test-runner
    triggers: ["test", "spec", "assert", "coverage", "jest", "pytest"]
```

```yaml
    tier: standard
    estimated_tokens: 500

  - name: documentation
    triggers: ["docs", "readme", "document", "explain", "comment"]
    tier: standard
    estimated_tokens: 600

  # Optional skills (first to evict)
  - name: image-processor
    triggers: ["image", "picture", "screenshot", "diagram", "png"]
    tier: optional
    estimated_tokens: 1000
    compression_priority: 1

  - name: calendar
    triggers: ["calendar", "meeting", "schedule", "event", "agenda"]
    tier: optional
    estimated_tokens: 500
    compression_priority: 2
```

SKILL.md Schema

The full skill definition loaded when a skill is activated:

```
# [Skill Name]

## Identity
- Name: [kebab-case identifier]
- Tier: [core | standard | optional]
- Version: [semver]

## Triggers
- [trigger word 1]
- [trigger word 2]
- [trigger word N]
```

```
## Description
[What this skill does, in 1-3 sentences]

## Dependencies
- [other-skill-name]
- None

## Configuration
- [key]: [value]

## Tools Required
- [Tool 1 description]
- [Tool 2 description]

## Prompt Template
[The actual prompt or instructions used when this skill is active]

## Examples
[Input/output examples showing expected behaviour]
```

Constraints

Field	Required	Constraint
Name	Yes	Must match catalogue entry name
Tier	Yes	Must match catalogue entry tier
Version	Yes	Semantic versioning (MAJOR.MINOR.PATCH)
Triggers	Yes	Must be superset of catalogue triggers
Description	Yes	Max 500 characters

Field	Required	Constraint
Dependencies	No	Must reference valid skill names
Configuration	No	Key-value pairs only
Tools Required	No	Informational (not enforced)
Prompt Template	Yes	The core content loaded into context
Examples	No	Recommended for complex skills

Tier Behaviour

Property	Core	Standard	Optional
Activation priority	First	Second	Third
Eviction order	Last	Middle	First
Max simultaneous	3	2	1
Budget allocation	60%	30%	10%
Auto-deactivate	20 turns	10 turns	5 turns

Trigger Design Guidelines

1. **Minimum 2 triggers** per skill (avoids single-point activation failures)
2. **Maximum 10 triggers** per skill (more causes false positives)

3. **No single-character triggers** (too generic)
4. **Minimise overlap** between skills (if unavoidable, tier priority resolves conflicts)
5. **Include synonyms** (e.g., “search”, “find”, “lookup” for the same skill)
6. **Include domain terms** (e.g., “kubernetes”, “k8s”, “kubectl” for a K8s skill)

Key Sources

- Chapter 12: Skill Architecture for AI Agents (catalogue structure, activation patterns)
- Chapter 10: Token Budget Architecture (budget allocation per type)
- Terraphim knowledge graph configuration (role-based skill definitions)
- OpenClaw skill loading patterns (production implementation)

Appendix D: Knowledge Graph Schema

This appendix documents the schema formats used for knowledge graphs throughout this book, with concrete examples from the Terraphim knowledge graph configuration.

Thesaurus Format

Terraphim uses a JSON-based thesaurus format for defining role-specific vocabularies. Each role gets its own thesaurus file, containing term-to-concept mappings with synonym group identifiers.

```
{
  "name": "Terraphim AI Developer",
  "data": {
    "learning": { "id": 1, "nterm": "knowledge" },
    "knowledge": { "id": 1, "nterm": "knowledge" },
    "lesson": { "id": 1, "nterm": "knowledge" },
    "insight": { "id": 1, "nterm": "knowledge" },
    "discovery": { "id": 1, "nterm": "knowledge" },
    "setup": { "id": 2, "nterm": "configuration" },
    "configuration": { "id": 2, "nterm": "configuration" },
    "install": { "id": 2, "nterm": "configuration" },
    "deployment": { "id": 2, "nterm": "configuration" },
    "initialise": { "id": 2, "nterm": "configuration" }
  }
}
```

Field Definitions

Field	Type	Description
`name`	String	Role name – determines which synonym set is active
`data`	Object	Term-to-concept mapping
`data[term]`	Object	A searchable term (surface form)
`data[term].id`	Integer	Synonym group identifier – terms with the same `id` are treated as synonyms
`data[term].nterm`	String	Normalised term (canonical form) – the concept this term maps to

How Synonym Groups Work

All terms sharing the same `id` are compiled into the same Aho-Corasick automaton pattern group. When any term in the group is matched, the system treats it as a match for the canonical `nterm`.

Example: searching for “lesson” matches the `knowledge` concept (id: 1), which also matches documents containing “learning”, “insight”, or “discovery”.

Role Definitions

Each role specifies a focus area, embedding strategy, and search depth:

Role	Focus	Embeddings	Depth

Role	Focus	Embeddings	Depth
Terraphim AI Developer	Rust-based AI assistant development	local	deep
OpenClaw Developer	TypeScript gateway/agent platform	local	standard
Last30Days Skill Developer	Web research skill development	none	basic
System Operator	System administration and operations	none	minimal
Learning Capture	Personal knowledge capture	local	deep
Terraphim Engineer (Default)	General engineering	local	deep

Role Configuration

```
roles:
  - name: "Terraphim AI Developer"
    focus: "Rust-based AI assistant"
    embeddings: local
    depth: deep
    thesaurus: "terraphim-ai-developer.json"
    haystacks:
      - path: "~/terraphim-ai"
        type: "code"
      - path: "~/.config/terraphim/docs"
        type: "documentation"
```

```
  - name: "System Operator"
    focus: "System administration"
    embeddings: none
    depth: minimal
    thesaurus: "system-operator.json"
    haystacks:
      - path: "/etc"
        type: "config"
```

Synonym Group Reference

A production knowledge graph typically defines 15-25 synonym groups covering the domain vocabulary. The following groups illustrate the pattern:

ID	Canonical Term	Synonyms	Domain
1	Knowledge	learning, lesson, insight, discovery	General
2	Configuration	setup, install, deployment, initialise	DevOps
3	Git	gitea, github, repository, version control	SCM
4	AI Assistant	terraphim, agent, claude, kimi	AI
5	Tool	skill, capability, function, feature	Platform
6	Storage	memory, persistence, database	Infrastructure

ID	Canonical Term	Synonyms	Domain
7	Context Management	arscontexta, onecontext, context, progress	CE
8	Retrieval	search, query, find, lookup	Search
9	Failure	error, mistake, bug, issue, problem	Debugging
10	Documentation	docs, guide, manual, readme	Docs
11-13	Languages	Rust/Cargo, JavaScript/Node, Python	Languages
14	Testing	test, spec, assertion, coverage	QA
15	Security	auth, encryption, vulnerability, CVE	Security
16	Performance	latency, throughput, optimisation, benchmark	Performance
17	Build	compile, bundle, package, release	Build
18	Execution	run, execute, invoke, spawn	Runtime
19	Debugging	debug, trace, log, inspect, breakpoint	Debugging
20	Repair	fix, patch, hotfix, workaround	Maintenance

Graph Schema (Property Graph Format)

For systems using property graphs (Neo4j, Memgraph), the following schema defines the node and relationship types:

```
// Node types
(:Concept {id: INT, label: STRING, definition: STRING})
(:Term {surface_form: STRING, language: STRING})
(:Document {path: STRING, type: STRING, last_modified: DATETIME})
(:Role {name: STRING, focus: STRING, depth: STRING})

// Relationship types
(:Term)-[:IS_SYNONYM_OF {group_id: INT}]->(:Concept)
(:Concept)-[:RELATES_TO {type: STRING, weight: FLOAT}]->(:Concept)
(:Document)-[:MENTIONS {position: INT, count: INT}]->(:Concept)
(:Role)-[:HAS_VOCABULARY]->(:Concept)
(:Role)-[:SEARCHES]->(:Document)
```

Graph Schema (RDF/Turtle Format)

For systems using triplestores, the equivalent schema in Turtle notation:

```
@prefix ce: <http://context-engineering.org/schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

ce:Concept a rdfs:Class ;
    rdfs:label "A domain concept in the knowledge graph" .

ce:Term a rdfs:Class ;
    rdfs:label "A surface form (searchable text) for a concept" .

ce:isSynonymOf a rdf:Property ;
    rdfs:domain ce:Term ;
    rdfs:range ce:Concept ;
```

```
    rdfs:label "Maps a surface form to its canonical concept" .

ce:relatesTo a rdf:Property ;
    rdfs:domain ce:Concept ;
    rdfs:range ce:Concept ;
    rdfs:label "Semantic relationship between concepts" .

ce:hasRole a rdf:Property ;
    rdfs:domain ce:Concept ;
    rdfs:range ce:Role ;
    rdfs:label "Concept is part of a role-specific vocabulary" .
```

As discussed in Chapter 22, the choice between property graphs and triplestores is an implementation detail. Both representations encode the same semantic information. The important decisions are about the ontology quality, synonym coverage, and relationship completeness – not the storage format.

Key Sources

- Terraphim knowledge graph configuration (production implementation)
- SKOS (Simple Knowledge Organisation System) specification
- Neo4j graph data modelling documentation
- RDF 1.1 Turtle specification

Appendix E: Context Health Checklist

This appendix provides a practical checklist for assessing the health of a context engineering system. Use it during periodic reviews (weekly or per-sprint) and at quality gates before major releases.

Structural Health

These checks assess whether the foundational context artefacts are complete and consistent.

Domain Model

- [ ] Domain model exists and is versioned
- [ ] All key domain terms are defined with unambiguous definitions
- [ ] Synonym groups are documented (term X = term Y in this context)
- [ ] Relationships between concepts are explicitly typed (not just "related to")
- [ ] Domain model has been reviewed by a domain expert (not just the engineer who wrote it)
- [ ] No orphan terms (terms defined but not used in any artefact)
- [ ] No undefined terms (terms used in artefacts but not in the domain model)

Knowledge Graph

- [ ] Knowledge graph is populated and queryable
- [ ] Term coverage: >80% of domain model terms appear in the KG
- [ ] Synonym coverage: key terms have at least 2-3 synonyms defined
- [ ] Relationship coverage: key entity pairs have typed relationships
- [ ] Role-based graphs are configured for each active context role
- [ ] KG build time is acceptable (<1s for graphs under 100K terms)
- [ ] Search results are deterministic (same query returns same results)

Vocabulary

- [ ] Vocabulary file exists and is maintained
- [ ] Industry-standard terms are included (not just internal jargon)
- [ ] Acronyms are expanded in the vocabulary
- [ ] Vocabulary is validated against actual usage in codebase/documents

Operational Health

These checks assess whether the context system is functioning correctly in production.

Retrieval Quality

- [ ] Retrieval accuracy is measured (% of queries returning relevant results)
- [ ] Precision at top-5 results is tracked
- [ ] Zero-result queries are logged and investigated
- [ ] Retrieval latency is within acceptable bounds (<100ms for deterministic, <2s for semantic)

Context Freshness

- [ ] Knowledge graph was updated within the last sprint/iteration
- [ ] No stale entries (documents referenced but no longer existing)
- [ ] New code/documents added since last KG update are identified
- [ ] Temporal metadata (last_updated, source_date) is present on KG entries

Drift Detection

- [ ] Baseline context snapshot exists
- [ ] Drift is measured at regular intervals
- [ ] Vocabulary drift: new terms appearing in usage that are not in the KG
- [ ] Assumption drift: conventions established in recent sessions that contradict earlier ones
- [ ] Relationship drift: entity connections that have changed meaning

- [ ] Drift threshold triggers are defined (e.g., >15% = warning, >30% = critical)

Governance Health

These checks assess whether the context management process is sound.

Change Management

- [ ] Context changes are tracked (who changed what, when, why)
- [ ] Impact analysis is performed before significant changes
- [ ] Changes to the domain model require review/approval
- [ ] Vocabulary additions follow a defined process (not ad-hoc)

Quality Gates

- [ ] Pre-LLM validation is automated (syntax, semantics, grounding)
- [ ] Post-LLM validation checks outputs against KG
- [ ] Gate pass rates are tracked over time
- [ ] Failed gates produce actionable feedback (not just “FAILED”)

Documentation

- [ ] Context architecture is documented (which systems, how they connect)

- [ ] KG schema is documented (node types, relationship types, properties)
- [ ] Role definitions are documented (who uses which vocabulary)
- [ ] Runbook exists for common context operations (rebuild KG, add terms, fix drift)

Agent-Specific Health

These checks apply when AI agents are consuming the context.

Agent Context Configuration

- [ ] Each agent role has a defined context scope (what it can see)
- [ ] System prompts reference the knowledge graph for grounding
- [ ] CLAUDE.md (or equivalent) is maintained and current
- [ ] Skill definitions include context requirements
- [ ] Token budget is defined and monitored per agent

Multi-Agent Coherence

- [ ] Shared vocabulary is consistent across agents
- [ ] Handover documents are generated between agent sessions
- [ ] No contradictory conventions established by different agents
- [ ] Agent outputs are validated against the shared domain model

Feedback Loop

- [ ] Failed agent actions are captured for learning
- [ ] Agent queries with zero results are logged
- [ ] Context gaps discovered by agents are fed back to the KG
- [ ] Agent performance metrics are tracked over time

Scoring Guide

For each section, count the number of checked items and calculate a percentage:

Score	Status	Action
90-100%	Healthy	Continue monitoring
70-89%	Adequate	Address gaps in next sprint
50-69%	At Risk	Prioritise context improvements
Below 50%	Critical	Stop feature work, fix context foundations

Recommended Review Cadence

Check Category	Frequency	Typical Duration
Retrieval quality spot-check	Daily (automated)	0 min (CI/CD)
Drift detection	Weekly (automated)	0 min (scheduled)
Full structural health	Per sprint	30-60 min
Full governance review	Monthly	1-2 hours

Check Category	Frequency	Typical Duration
Agent coherence audit	Per major release	2-4 hours

Key Sources

- Chapter 14: Validation and Quality Gates (KLS protocol, drift detection)
- Chapter 21: Enterprise Context Governance (gated lifecycle, health metrics)
- Chapter 7: Terraphim Graph Embeddings (deterministic search properties)
- Chapter 18: Multi-Agent Context Management (coherence patterns)

Appendix F: Resources and Tools

This appendix compiles the tools, frameworks, standards, and references cited throughout this book.

Knowledge Graph Tools

Tool	Type	Description	Chapters
Terraphim	Deterministic search engine	Aho-Corasick automaton-based search with role-based knowledge graphs. Sub-millisecond matching across 1.4M+ patterns.	6, 7, 9, 19
Neo4j	Graph database	Property graph database with Cypher query language. Market leader for enterprise knowledge graphs.	5, 19, 21
Memgraph	Graph database	In-memory property graph compatible with Cypher. Optimised for real-time queries.	19, 21
Apache Jena	RDF framework	Java framework for building Semantic Web and Linked Data applications. Supports SPARQL.	5, 19
TrustGraph	Graph store	Apache Cassandra-backed graph store designed for billion-node knowledge graphs with trust scoring.	19, 21

Embedding and RAG Tools

Tool	Type	Description	Chapters
Pinecone	Vector database	Managed vector database for similarity search. Cloud-hosted.	6
Weaviate	Vector database	Open-source vector database with built-in vectorisation.	6
Chroma	Vector database	Lightweight embedding database for AI applications.	6
LangChain	RAG framework	Framework for building LLM applications with retrieval pipelines.	19
LlamaIndex	RAG framework	Data framework for connecting LLMs to external data sources.	19

AI Models and APIs

Model/Service	Provider	Relevance	Chapters
Claude	Anthropic	Primary LLM for context-engineered agents. 200K token context window.	10, 13, 15
GPT-4	OpenAI	128K context window. Token budget comparison baseline.	10

Model/Service	Provider	Relevance	Chapters
BERT	Google	Base embedding model (768 dimensions). Reference for embedding discussion.	6
text-embedding-3	OpenAI	Embedding model (up to 3072 dimensions).	6
MedGemma	Google	Medical domain LLM. Used in KG grounding case study.	9

Standards and Specifications

Standard	Organisation	Description	Chapters
RDF 1.1	W3C	Resource Description Framework for expressing data as triples.	5, 19, App D
OWL	W3C	Web Ontology Language for formal ontology definitions.	5, 19
SKOS	W3C	Simple Knowledge Organisation System for vocabularies and thesauri.	5, 8, App D

Standard	Organisation	Description	Chapters
SPARQL	W3C	Query language for RDF data.	5, 19
Cypher	Neo4j/GQL	Graph query language for property graphs.	5, 19
JSON-LD	W3C	JSON-based Linked Data serialisation format.	19
Turtle	W3C	Terse RDF Triple Language. Compact RDF serialisation.	19, App D

Algorithms

Algorithm	Description	Chapters
Aho-Corasick	Multi-pattern string matching in linear time. Foundation for Terraphim's deterministic search.	6, 7
PageRank	Graph centrality algorithm for ranking nodes by connectivity. Applicable to KG result ranking.	7, 21
Cosine Similarity	Vector similarity measure used in embedding-based search.	6
TF-IDF	Term frequency-inverse document frequency for classical text ranking.	6

Development Tools

Tool	Purpose	Chapters
Claude Code	AI coding assistant with CLAUDE.md instructions and version-dependent AGENTS.md support.	13, 16
OpenAI Codex	Coding agent using AGENTS.md for project guidance; discovery and scope are client-specific.	13
OpenClaw	AI assistant platform with layered memory, skill system, and autonomous heartbeat loops.	17
Quarto	Scientific publishing system. Used to produce this book.	–

Project Instruction Formats

Chapter 13 compares shared instructions with client-specific discovery. Official documentation checked on 9 October 2026:

- AGENTS.md format
- Claude Code memory and instruction files
- Codex AGENTS.md discovery
- Cursor rules
- Cascade AGENTS.md guidance

- GitHub Copilot repository instructions

Context Engineering Frameworks

Framework	Description	Chapters
Progressive Disclosure	UI design pattern applied to context: show only what is needed, reveal more on demand.	10, 12
Constitutional Context	Files that define AI identity and must never be compressed or dropped.	10, 17
KLS (Knowledge Layer Schema)	Custom validation protocol for AI-generated content across conceptual, structural, and semantic dimensions.	14
Token Budget Architecture	Systematic approach to allocating context window tokens across categories.	10, App B
Hybrid Pipeline	Retrieval architecture combining deterministic, semantic, and graph methods.	19
FPF and Systems Engineering DPF	Selected distinctions, revision 0c6ade2, 8 October 2026: intended use, evidence and consequences. Editorial adaptation, not framework conformance.	Introduction, 4

Research Papers and Articles

Reference	Relevance	Chapters
Karpathy, A. (2025). “Vibe Coding” and LLM coding guidelines.	Operational discipline for AI-assisted development.	3, 13
Cutler, J. (24 September 2026). “TBM 441: AI, the Loss of Positive Friction, and What to Do About It”.	Deliberate judgement and source traceability at information handoffs.	3
Hrishi & Prithvi (Anthropic). Build-review codon loops.	Multi-agent coordination with context walls.	15
FLAURA Trial. Osimertinib 80mg for EGFR-mutated NSCLC.	Reference for KG grounding safety case study.	9
“Some Simple Economics of AGI” analysis. Verification bottleneck.	Economics of AI verification at scale.	1, 22
LLM-as-a-Judge best practices.	Evaluation methodology: ensemble judging, bias cataloguing, calibration.	14, 22
Context Graphs: From RAG to Autonomous Learning.	Eight-step progression from basic RAG to self-describing context stores.	5, 19

Clinical References

Reference	Relevance	Chapters
SNOMED CT	Systemised Nomenclature of Medicine. 1.4M+ clinical terms used in Terraphim healthcare demo.	6, 7, 9

Reference	Relevance	Chapters
FLAURA Trial	Phase III clinical trial establishing Osimertinib 80mg daily for first-line EGFR-mutated NSCLC treatment.	9

Key Sources

- Individual chapter “Key Sources” sections
- Terraphim AI technical documentation
- W3C Semantic Web standards
- Neo4j graph data science documentation

Appendix G: Spec-Driven Verification

Chapter 14 argued that every validation gate needs a criterion, that the criteria come from executable specifications, and that the specifications are only meaningful once a reconciled domain model (Chapter 4) fixes what their words mean. This appendix carries the mechanics that the chapter summarised: a layered verification model from the practitioner literature, the Gherkin specification as the bridge between the domain model and automated tests, and the Spec-Kit toolkit that operationalises the workflow. It is reference material; the argument is in Chapter 14 and the practice in Chapter 23.

Gherkin, the notation BDD scenarios are written in, builds each scenario from three core step keywords: Given states a precondition, When an action, Then the expected outcome; And and But add clauses, and Feature, Scenario and Background structure the file around them.

The Swiss Cheese Model of Verification

Ankit Jain, writing in Latent Space ("How to Kill the Code Review", 2025), articulates why the two-layer model of Chapter 14 matters for AI-native development. His argument, with the history and the industry figures as he presents them: human code review became standard practice only in the last decade or so, and under AI-assisted

generation it is now the bottleneck, because teams merge far more pull requests while the time spent reviewing them grows almost as fast. The figures are his and are not reproduced here; the conceptual point stands without them. The solution is not faster review – it is layered verification that moves human judgement upstream from "Did you write this correctly?" to "Are we solving the right problem?"

Jain proposes a five-layer Swiss Cheese model, where each layer catches failures that the others miss:

Layer 1: Compare multiple options. Ask three agents to implement independently. Rank by verification pass rate, smallest diff, or fewest new dependencies. This is a competition pattern – the agents do not collaborate, they compete, and the best output wins.

Layer 2: Deterministic guardrails. Tests, type checkers, contract verification – pass/fail artefacts, not opinions. This maps directly to the pre-LLM validation layers of Chapter 14: syntax checking, consistency validation, and knowledge graph grounding. These guardrails enforce what is computable without intelligence.

Layer 3: Humans define acceptance criteria. BDD specifications authored by humans before any code is generated. The specification is the primary artefact, not additional documentation. The agent implements; the BDD framework verifies; the human never reads the implementation unless verification fails. This is the spec-driven development layer – and it depends on the context engineering upstream to ensure the specs themselves are correct.

Layer 4: Permission systems as architecture. Narrow the filesystem scope per task. Trigger automatic human review for sensitive changes: authentication logic, database schema modifications, new dependencies. This is scope management, not intelligence – a structural constraint that limits blast radius regardless of agent confidence.

Layer 5: Adversarial verification. Separate the coding agent from the verification agent architecturally. The agent that writes the code never reviews its own output. An optional red-team agent attempts to break what the first agent built. This connects to the build-review codon loops of Chapter 18, where context-walled agents evaluate each other's work with fresh eyes.

No single layer catches everything; the layers are arranged so that each covers the others' blind spots: if a wrong assumption passes through the domain model (Layer 3 missed it), the deterministic guardrails (Layer 2) might catch the resulting type violation. If the type system is too permissive, the adversarial agent (Layer 5) might find the flaw. Defence in depth means that failures must penetrate every layer simultaneously to reach production.

BDD Specifications as the Bridge Artefact

Behaviour-Driven Development specifications connect the domain model to automated tests. People can review their wording against the model, and test runners execute the scenarios through step definitions.

A BDD scenario has a precise structure:

```
Feature: Fund Transfers

  Scenario: Insufficient funds
    Given a user with 50 GBP in their account
    When they attempt to transfer 100 GBP
    Then the transfer is blocked
    And an InsufficientFunds error is returned
    And the account balance remains 50 GBP
```

The scenario's nouns, including "user", "account", "GBP", "transfer" and "InsufficientFunds", need definitions in the domain model. Its relationships, such as a user having an account, must agree with that model. The expected balance of 50 GBP must follow from a documented invariant. Resolve ambiguous terms before treating the scenario as an acceptance test.

Trace this scenario to requirement R-TR-01 in Chapter 4 and to the account holder's concern about losing money through rejected transfers. The transfer service shall fulfil that requirement; the scenario supplies evidence that the implementation does so for these conditions. Keep separate requirement links for blocking the transfer and returning the specified error. Stakeholder validation must also consider whether the service meets the underlying need across the intended operating conditions. A passing example alone cannot settle that question.

This is where context engineering directly feeds spec-driven development. The domain model provides the vocabulary; the BDD spec uses that vocabulary to express testable behaviour; the

validation layers execute those tests against the generated implementation. The human's job shifts from reviewing code to reviewing specifications – a fundamentally different activity that operates at a higher level of abstraction.

The workflow becomes:

1. **Context engineering** records stakeholder concerns, needs and requirements alongside the domain model (entities, relationships, invariants)
2. **Humans allocate requirements to services and write BDD specs** using domain model vocabulary, retaining the links to stakeholder needs
3. **Agents implement** against the specs (humans review the specs by default, and the code only on sensitive paths, failed gates, high-impact changes or audit samples)
4. **Automated verification** runs the BDD scenarios, type checks, and contract validation
5. **Findings feed back** to requirements, service allocation or implementation as appropriate; stakeholders validate that the resulting services address their needs

Human time concentrates on steps 1 and 2 – understanding the domain and expressing requirements. Agent time concentrates on step 3. Verification is automated. This is the practical expression of the two-layer model: context engineering ensures the specs are right; spec-driven development ensures the code matches the specs.

Tools like Spec-Kit (GitHub's open-source spec-driven development toolkit) operationalise this workflow with a five-phase process: constitution (project-wide rules), specification (user scenarios), plan (technical design), tasks (executable work items), and implementation. The parallels to this book's approach are direct – the constitution maps to CLAUDE.md (Chapter 13), the specification phase maps to the BDD layer described here, and the planning phase maps to the domain model work in Chapters 3-4. The key difference is that Spec-Kit assumes context is given (starting from a constitution), while context engineering treats context as the thing that must be engineered first.

Key Sources

- Ankit Jain, "How to Kill the Code Review" (Latent Space, 2025): the Swiss Cheese verification model and the review-bottleneck argument; the industry figures in that article are his and are not reproduced here
- Dan North, "Introducing BDD" (2006): Behaviour-Driven Development origins
- Spec-Kit (GitHub), spec-driven development toolkit documentation
- Chapters 4, 13, 14, 18 and 23 of this book

Link Executable Specifications to Decision Evidence

Attach a stable case identity and requirement identifier to each executable acceptance example. Record the observed result separately from the expected result, then reference that result from the assessed candidate’s decision trace. Appendix H provides illustrative templates; Chapters 15–17 separate evaluation, action policy and evidence lineage. A passing fixture verifies its specified behaviour. Stakeholder validation still needs to assess whether the selected cases address the concern in the intended operating context.

Appendix H: Evaluation and Decision Record Templates

These compact records support the maintenance case introduced in the introduction and Chapter 4, and developed in Chapters 15–17. They are original teaching templates, not ISO, OpenTelemetry or Terraphim protocol schemas. Values and outcomes are synthetic illustrations. The templates show required relationships; they are not an executed evaluation pack or a complete executable schema.

Keep identifiers stable and content revisioned. A requirement identifies agreed behaviour; a case exercises it; a result records an observation; a trace joins that observation to authority and action; an experiment record preserves the attempt and its lineage. Additional production controls depend on the intended environment.

Requirement-to-Evidence Matrix

Requirement	Stakeholder concern	Allocated service	Verification evidence	Stakeholder validation question
R-USE-01	Service owner: distinguish restoration from an attempted repair	Maintenance	Completion and required delivery records for the identified job and run	Does the recorded outcome establish the restoration the owner needs?

Requirement	Stakeholder concern	Allocated service	Verification evidence	Stakeholder validation question
R-BOUND-01	Other teams and risk owner: contain shared-resource exposure	Execution	Scope, per-run and cumulative quota checks; withhold when evidence is unavailable	Do the limits protect affected work under concurrent demand?
R-EVAL-01	Service owner: avoid harmful promotion	Evaluation	Frozen triggering and protected cases; candidate-specific results	Do these cases cover the operational harms that matter?
R-POL-01	Operator: do not act without adequate evidence	Policy/admission	Unknown, unavailable and missing-evidence decisions withhold promotion	Is the escalation route usable during an incident?
R-TRACE-01	Reviewer: reconstruct acceptance	Trace	Requirement, service, digest, policy and evidence links resolve	Can the reviewer explain the action from the retained record?
R-DATA-01	Privacy owner: limit evidence exposure	Evidence	Access denial, redaction and retention-expiry checks	Can authorised review proceed without unnecessary disclosure?

Requirement	Stakeholder concern	Allocated service	Verification evidence	Stakeholder validation question
R-CHANGE-01	Maintainer: prevent stale approval	Promotion	Changed candidate or governing policy requires reassessment	Does renewed review protect working behaviour without hiding delays?

These allocations describe responsibilities, not a requirement for separate deployed services. The matrix should be agreed with the stakeholders; mechanically complete links cannot establish that the chosen requirements meet their concerns.

Frozen Case

The expected outcome belongs in the reference store, withheld from the subject and from an optimiser tuning against protected cases. Give a judge only the evidence its rubric requires; do not accidentally include the answer key in its input envelope.

```
# Illustrative record, not a completed evaluation.
case_id: protected-01
case_revision: fixture-v1
requirement_ids: [R-EVAL-01, R-CHANGE-01]
service_id: evaluation
split: protected
input_ref: fixtures/protected-01/input.txt
input_digest: <digest-of-input-bytes>
observations_required: [tool_result, candidate_output]
reference_ref: restricted-labels/protected-01.yaml
reference_revision: adjudicated-v1
```

Specify what constitutes a pass, who adjudicated the reference and how ambiguous cases are resolved. The split name alone does not enforce access restrictions.

Result Record

Separate the observed output, grading and execution status. A timeout is not a negative label and is not a pass. Record an unavailable immutable model identity explicitly.

```yaml
# Illustrative protected-regression result.
result_id: result-protected-01-rule-b
case_id: protected-01
case_revision: fixture-v1
candidate_digest: <digest-of-rule-b>
output_ref: evidence/output-protected-01.json
output_digest: <digest-of-output-bytes>
execution_status: completed
grader_revision: deterministic-check-v1
rubric_revision: protected-behaviour-v1
resolved_model_id: not_applicable
verdict: fail
finding: protected_behaviour_regressed
provenance: synthetic_fixture
```

For model grading, add the configured identity, resolved identity where available, sampling settings and error details. Report independent case counts separately from repeated calls on each case. Missing observations remain missing; a fallback does not count as assessed evidence.

Decision Trace

This example records a refusal after protected-case failure. The rationale is a checkable statement, not raw hidden reasoning. Digest placeholders must be replaced by computed values before a real integrity check is possible.

```
# Illustrative refusal; no promotion was authorised.
trace_id: trace-rule-b-refused
parent_trace_id: null
task_id: maintenance-example
stakeholder_id: service-owner
concern_id: harmful-promotion
requirement_ids: [R-EVAL-01, R-POL-01, R-TRACE-01]
service_id: admission
candidate_digest: <digest-of-rule-b>
policy_revision: admission-v1
scope: synthetic-maintenance-rule
assessment_refs: [result-protected-01-rule-b]
model_recommendation: not_applicable
policy_decision: deny
requested_action: promote
authorised_action: none
rationale: Protected case failed; promotion criteria unmet.
alternatives: [retain_current_rule, revise_candidate]
action_id: null
outcome_ref: null
access_policy: example-reviewers-only
retention_policy: example-retention-v1
```

An allowed trace additionally binds the approved action, validity interval and candidate to the execution check. Link the eventual action and outcome after they occur. Keep the original decision immutable and append the outcome or a superseding decision.

Integrity metadata, key identity and replay protection require a defined implementation contract beyond this sketch.

Experiment Lineage

Retain rejected experiments alongside successful ones. A lesson proposed, a lesson applied and an observed improvement are distinct states.

```
# Illustrative experiment record.
experiment_id: maintenance-rule-attempt-02
parent_experiment_id: maintenance-rule-attempt-01
candidate_digest: <digest-of-rule-b>
context_revision: maintenance-context-v1
case_set_revision: frozen-suite-v1
result_refs: [result-trigger-01-rule-b, result-protected-01-rule-b]
decision_trace_ref: trace-rule-b-refused
status: rejected
lesson_status: proposed
lesson: Preserve protected behaviour when changing the trigger rule.
application_evidence_refs: []
```

Missing evidence produces an unresolved decision. A changed candidate produces stale approval requiring reassessment. A stable but incorrect judge remains an evaluator defect, even when every repeated verdict agrees. The beneficial candidate is eligible only when required evidence, current scope and current approval are all present. These are expected teaching outcomes; an implementation must record its actual checks separately.

Appendix I: Experiment Reference

This appendix summarises six author-reported operational studies and designs from the OpenClaw and Terraphim systems discussed in Chapters 7, 9 and 20. It is an index to the evidence and its limits, rather than a substitute for the underlying records. The ledgers and experiment directories were unavailable in the editorial checkout; none of the numerical results has been independently reproduced for this edition. A reported observation, a proposed mechanism and a verified runtime guarantee remain different kinds of claim.

Experiments are numbered E1 to E6 and are cited throughout the book as, for example, “Experiment E3, Appendix I”.

E1: Cron Reliability Study

Where it appears: Chapter 20 (scheduled jobs and their failure modes).

Design. The account reports a forensic analysis of recorded runs of the author’s scheduled-job fleet across two registry eras: 1,586 runs in a legacy registry era (February–May 2026) and 469 runs in a later era (August–September 2026), 2,055 runs in total. The chapter derives requirements for status, phase identity, timestamps and deliverable evidence from the reported failure taxonomy; the underlying classification procedure is not reproduced here.

Headline results.

Finding	Value
Run-level success, legacy era	75.9%
Run-level success, current era	51.4%
Dominant failure class	Timeouts: 178 of 228 errors in the stated error subset (78%)
Best job (stateless news aggregator)	93.2% success
Worst job (multi-stage morning briefing)	13.7% success
Success clustering	P(success | success) = 0.85; P(success | failure) = 0.10; odds ratio 52

Failure-mode taxonomy. Five recurring modes were identified, each with a detection signature: (1) resource-exhaustion fast-fail — during a disk-full window, all nine jobs failed in a median of 1.6 seconds across six days; (2) the successful error — a job reporting status error while its deliverables were fully produced; (3) the placeholder success — a job reporting status ok while writing a 744-byte template with no content; (4) silent fallback — a voice-generation job delivered working audio from a secondary provider for four consecutive days while the primary credential rotted; (5) phase-localised breakage — extraction and consolidation phases of one pipeline failed at different times, inverting after a rework.

Limits. The two eras differ in time and job composition, so the drop from 75.9% to 51.4% does not isolate the effect of any redesign. Runs from the same job are not independent observations, so the odds ratio of 52 describes clustering, not causation.

Source. PR #132, cron reliability account, commit `4ed3d3809e6965cafed9c46dccaac4b25560d9f8`. The original run ledger was not inspected for this edition.

E2: Prediction-Scoring Loop (Operational Self-Calibration)

Where it appears: Chapter 14 (validating the system's own predictions).

Design. The account describes a month of operation in which nine scheduled jobs fed a daily memory consolidation, and each consolidation ended with scored predictions for the next day: a stated probability and, once the day closed, a binary outcome, written to a durable ledger. Probabilities were recorded before outcomes; hindsight re-grading was the failure mode this design was built to prevent. The inspected PR does not establish the complete ledger window or observation protocol.

Headline results.

Metric	Value
Hit rate	96.4% correct
Mean stated confidence	35.3%
Mean Brier score at start	0.572 (a coin-flip baseline scores 0.25)
Predictions stated at 0–5% confidence	Came true 100% of the time (n = 42)

The account interprets this pattern as inverted calibration. Event probability, confidence in a selected label and thresholded correctness must be distinguished before that interpretation can be confirmed.

Diagnostic hypotheses. The account proposes failure salience as one explanation for pessimistic forecasts. Alternatives include inverted event definitions, incomplete observations and a changing class mix. It also describes a narrative contradicting data: the memory narrative claimed extraction jobs formed clusters of “two to four days maximum”, while the run ledger contained ten-day success streaks with a reported lag-one autocorrelation of 0.74.

The calibration loop. The account reports a mean Brier score changing from 0.572 to approximately 0.10 over three weeks, alongside three interventions: (1) a recovery-boost factor — after a fix, the second consecutive clean cycle earned a probability uplift (+15 percentage points in this system), as a local heuristic; (2) consecutive-day confirmation — a “recovery” or “regression” claim requires two consecutive observations, after single data points were repeatedly misclassified; (3) per-class scoring — engagement predictions failed structurally while infrastructure predictions scored near-perfectly, and a blended average concealed this.

Limits. The published record does not provide enough underlying ledger detail to reconcile every aggregate value or to establish that scoring itself caused the improvement; the case motivates an inspectable prediction ledger rather than a transferable performance guarantee.

Source. PR #132, operational self-calibration account, commit `4ed3d3809e6965cafed9c46dccaac4b25560d9f8`. The original prediction ledgers were not inspected for this edition.

E3: Skill-Router Evaluation (exp-002, in-distribution)

Where it appears: Chapter 12 (routing as context selection).

Design. A production OpenClaw deployment compared its trigger-substring router (single-shot matching with an unconditional fallback) against a gated structure (a yes/no gate, then choice and compound tags). Both were run over a 140-case held-out set containing a negative class and compound labels — an evaluation the original router had never had; its entire prior validation corpus was one happy-path example.

Results.

Metric	Single-shot baseline	Gated router
Overall accuracy	32.9%	99.3%
False positives on negative cases	50/50	0/50
Compound exact-set correctness	0/25	25/25
Mean Brier score, as reported	0.2569	0.0158
Positive-class accuracy	51.1%	98.9%

Findings. (1) The negative class was the highest-value gate: knowing what not to load mattered more than any matching improvement; the baseline routed capability questions such as “can you parse JSON?” to the JSON skill at 0.9 confidence. (2) Compound requests were silently dropped by the single-skill output schema: “fetch the URL, parse JSON and extract the text” routed to exactly one skill while the other two operations vanished. (3) Confidence shape mattered as much as accuracy: the baseline showed a flat 0.5 confidence on every fallback route, simultaneously over-confident on negatives and under-confident on genuine fallbacks.

Residual ambiguity. The account identifies one disputed evaluation-set label — a composed-skill input structurally identical to a compound input, labelled oppositely. It was documented rather than over-fitted away.

Limits. The underlying experiment directory was unavailable in the editorial checkout, so these figures are author-reported and not independently reproduced.

Source. PR #133, in-distribution routing account, commit `ab7510feb89b54345e92032c53903f8590afd9be`. The original cases and scoring outputs were unavailable.

E4: Skill-Router Evaluation Under Distribution Shift (exp-002, out-of-distribution)

Where it appears: Chapter 25 (confidence under distribution shift).

Design. A fresh 100-case set spanning eight domains unfamiliar to the router, scored by both the gated router and the single-shot baseline of Experiment E3.

Results. The gated router achieved 70% accuracy against the baseline's 22%, while its mean Brier score was slightly worse: 0.2475 against 0.2388. Against its in-distribution Brier of 0.0158 (E3), the gated score worsened approximately 16-fold. Erroneous decisions carried confidences from 0.85 to 0.95, and compound exact-set correctness collapsed to 33% — the hardest-hit slice. Paraphrase and non-native phrasing broke the gate more often than the matcher.

Limits. The compound subset denominator and the confidence-event definition are not recorded in the inspected material, so these figures cannot support a complete uncertainty estimate or calibration diagnosis. The reported change motivates evaluation under distribution shift; accuracy alone does not describe confidence quality. Brier score combines more than calibration, so these aggregate values cannot isolate calibration error.

Source. PR #133, distribution-shift account, commit `ab7510feb89b54345e92032c53903f8590afd9be`. The original out-of-distribution cases were unavailable.

E5: Consolidation Pipeline Review

Where it appears: Chapter 11 (when compression becomes the failure mode).

Design. A critical internal review of the author's autonomous memory-consolidation pipeline after a month of nightly operation, during which the pipeline appended roughly 10KB of "daily consolidation" per day while its own status reports claimed memory saturation.

Findings. (1) Novelty filtering was absent: each day's summary restated the previous day's facts in new words, and the dedup ratio was never measured — only size. (2) Recorded lessons had a near-zero application rate: dozens of "lessons learned" entries accumulated while the same mistakes recurred weeks later. (3) Identified problems had no closure mechanism: the system repeatedly documented the same three open issues without ever scheduling or executing a fix. The pipeline compressed faithfully — old memory stayed small — while adding low-value bulk that size metrics could not distinguish from signal.

Remediation design. Measurement at the compression boundary: a novelty gate before append; consolidation reports that include information metrics (novelty ratio, lesson application rate, open-issue closure) alongside byte metrics; and periodic audits of what compression made unfalsifiable.

Limits. These are reported diagnostic observations from one deployment; they do not establish that a particular metric corrected the pipeline.

Source. PR #133, consolidation review, commit `ab7510feb89b54345e92032c53903f8590afd9be`. The PR's YAML values of 0.35

novelty and 0.22 lesson application illustrate a proposed report; they are not established measurements in the inspected record.

E6: Continuity Loop Handoff Study

Where it appears: Chapter 18 (durable handoffs between sessions).

Design. The author's account describes the Continuity Loop pattern in Terraphim deployments: at session end the agent writes a structured handoff file (`memory/handoffs/PENDING.yaml`) carrying an executive summary, next actions and open loops, with a 24-hour expiry window; at session start a new instance loads the handoff automatically only if it is younger than the expiry window, then archives it to a permanent ledger. A sibling-agent deployment in the same workspace used the primitive to coordinate agents that do not share memory, under an explicit contract: isolated workspaces, one designated shared surface for handoff files only, a structured schema, and handoffs written for a stranger with zero conversation history.

Findings. Expiry bounds automatic reuse; it does not guarantee automatic resumption weeks later, and after a longer absence the receiving agent needs an explicit ledger review and renewed confirmation of relevance and authority. Knowledge graphs can represent facts and commitments. The handoff contract makes ownership, deadlines and acknowledgement operationally explicit; expiry exposes a need for review without granting new authority.

Limits. The pattern is documented from a single deployment family; the guarantees are conventional (file semantics) rather than enforced.

Source. PR #133, Continuity Loop and sibling-agent contract, commit `ab7510feb89b54345e92032c53903f8590afd9be`. This edition does not claim a separately observed deployment trial.

Appendix J: Glossary

Key terms used throughout this book, organised alphabetically. Terms marked with (KG) appear in Terraphim knowledge graph synonym groups. Terms marked with (Ch N) reference the chapter where they are primarily defined.

A

Aho-Corasick Automaton (Ch 7)
: A finite-state pattern matching algorithm that finds all occurrences of a set of patterns in a single scan through the text. Used by Terraphim for deterministic term matching across millions of patterns in sub-millisecond time. Linear time complexity O(n) regardless of pattern count.

AI Sandwich (Ch 22)
: A firm structure where AI handles generation (measurable work), humans handle verification (unmeasurable work), and AI handles final polish. The middle layer does not scale, creating a structural bottleneck.

Artefact (KG)
: A versioned, reviewable output of a development process – code, documents, configurations, domain models. In context engineering, artefacts include knowledge graphs, vocabularies, and skill definitions.

C

Cascade Pattern (Ch 19)

A hybrid pipeline routing pattern that tries the cheapest retrieval method first and escalates to more expensive methods only when confidence is low. Optimises for cost.

CLAUDE.md (Ch 13)

A project-level configuration file that defines rules, conventions, and context for AI coding assistants. Acts as persistent context that survives across sessions.

Codifier's Curse (Ch 22)

The phenomenon where codifying expertise into AI systems destroys the tacit knowledge that was never explicitly documented. The system works until it encounters edge cases where that tacit knowledge mattered.

Compression (Ch 11)

Techniques for reducing context size while preserving meaning. Includes summarisation, extraction, and structured representation.

Connectivity-Based Ranking (Ch 7)

A scoring method where documents are ranked by the strength and number of their connections to query terms in the knowledge graph. More connections through typed relationships yield higher scores.

Context Drift (Ch 14)

The gradual misalignment between a system's knowledge representation and the actual state of the domain. Types include

vocabulary drift, assumption drift, relationship drift, and goal drift.

Context Engineering (Ch 2)
: The discipline of managing the information that AI systems use to generate responses. Encompasses knowledge representation, retrieval, validation, and governance.

Context Graph (Ch 19)
: A triples-based representation of data designed for AI retrieval, using graph structure, semantic-web formats and ontology definitions.

D

Deterministic Precision (Ch 6, Ch 7)
: Synonym for **100% accuracy for known terms**. The property of a search system that guarantees every vocabulary-matched term is found with zero false negatives and zero false positives. Achieved by Terraphim through Aho-Corasick automata operating on a pre-defined vocabulary. The scope is limited to terms that exist in the knowledge graph; for unknown terms, hybrid pipelines (Ch 19) provide coverage.

Deterministic Search (Ch 6)
: Search where the same query always produces the same results. Contrasted with probabilistic/semantic search where results may vary.

Domain Model (Ch 4)

A formal representation of the concepts, relationships, and rules within a specific problem domain. The foundation for knowledge graphs and context engineering.

Drift Detection (Ch 14)

Monitoring for changes in context alignment over time. Includes vocabulary drift (new terms appearing), assumption drift (conventions changing), and relationship drift (connections shifting).

E

Embedding (Ch 6)

A numerical vector representation of text, used for semantic similarity search. Terraphim's "graph embeddings" are deterministic alternatives that preserve full graph structure rather than compressing into vectors.

Ensemble Judging (Ch 22)

Using multiple LLM judges and aggregating their scores to reduce single-model bias in automated evaluation.

Explainability (Ch 7)

The ability to trace exactly why a search result was returned – which terms matched, at what positions, through which synonym expansions, and along which relationship paths. A key advantage of deterministic search over vector similarity.

F

Fan-Out Pattern (Ch 19)

A hybrid pipeline routing pattern that runs multiple retrieval methods in parallel and merges results. Documents found by multiple methods receive boosted scores. Optimises for recall.

G

Grounding (Ch 9)

Verifying that AI-generated claims are supported by facts in the knowledge graph. A safety gate that prevents hallucination.

GraphRAG (Ch 19)

Retrieval-augmented generation that uses entities, relationships and graph-derived summaries as context.

H

Haystack (KG)

In Terraphim, a collection of documents indexed for search. Each role can search across different haystacks (local folders, repositories, documentation).

Hook (Ch 16)

A script or function that executes automatically in response to events (pre-commit, post-tool-use, etc.). Used to automate context validation, knowledge capture, and safety checks.

Hybrid Pipeline (Ch 19)

A retrieval architecture that combines deterministic, semantic, and graph-based methods, routing queries to the most

appropriate strategy based on their characteristics.

K

KLS (Knowledge Layer Schema) (Ch 14)
: A structured validation protocol for AI-generated content. Validates across conceptual, structural, technical, semantic, and style dimensions.

Knowledge Graph (Ch 5)
: A data model for organising knowledge as entities (nodes) connected by typed relationships (edges). Distinct from the database that stores it.

L

LLM-as-Judge (Ch 22)
: Using a large language model to evaluate the quality of AI-generated outputs against defined rubrics. Offers ~80% of human evaluation accuracy at ~1% of cost.

M

Measurability Bias (Ch 22)
: The tendency for AI automation to target tasks that are easy to verify, not necessarily tasks that are simple. Creates selection

bias where measurable work is automated while unmeasurable but critical work is neglected.

Missing Junior Loop (Ch 22)
: The structural problem caused by AI automating junior engineering roles, which historically served as the training pipeline for developing senior expertise.

O

Ontology (Ch 5)
: A machine-readable specification of concepts, their hierarchies, and relationships. More formal than a vocabulary or taxonomy. Key standards include OWL and SKOS.

OntologyRAG (Ch 19)
: Retrieval-augmented generation that uses an ontology to constrain or expand retrieval through defined classes and relationships.

P

Property Graph (Ch 5, Ch 19)
: A graph model where nodes and edges can have properties (key-value pairs). Used by Neo4j, Memgraph. Queried with Cypher. Contrasted with RDF triplestores.

R

RAG (Retrieval-Augmented Generation) (Ch 19)
: A technique where relevant documents are retrieved and included in an LLM prompt to ground responses in factual context.

RDF (Resource Description Framework) (Ch 19)
: A W3C standard for expressing data as subject-predicate-object triples. Uses URIs for global identification. Serialisation formats include Turtle, JSON-LD, and N-Triples.

Role-Based Graph (Ch 7)
: A knowledge graph where the same entities can have different synonym edges and relationships depending on the active context role. Enables the same term to mean different things in different contexts.

Rolegraph (KG)
: Terraphim's implementation of role-based graphs. Each rolegraph handles ingestion and ranking via Aho-Corasick automata, with search inference in 5-10 nanoseconds.

S

Semantic Enrichment Pipeline (Ch 19)
: A hybrid pattern that uses deterministic search to establish a precise foundation, then uses semantic methods to detect and fill conceptual gaps.

Skill (Ch 12)
: A self-contained capability that can be attached to an AI agent. Includes a prompt template, tool definitions, and context

requirements.

Synonym Group (Ch 7, Appendix D)
: A set of terms that are treated as equivalent for search purposes. All terms in a group map to a single canonical concept via a shared group identifier.

T

Temporal Context (Ch 19, Ch 22)
: Information about when facts were established, how long they have been corroborated, and how their validity changes over time. Freshness and recency are not the same as accuracy and precision.

Thesaurus (Appendix D)
: In Terraphim, a JSON file mapping surface forms (searchable terms) to canonical concepts via synonym group identifiers. Each role has its own thesaurus.

Token Budget (Ch 10)
: The allocation strategy for distributing available context window tokens across system prompt, user context, retrieved documents, and generation space.

Triple (Ch 5, Ch 19)
: The fundamental unit of knowledge graph data: Subject-Predicate-Object. Predicates encode richer information than simple verb-based connections.

V

Validation Gate (Ch 14)
: A checkpoint where AI-generated content must pass quality criteria before progressing. Gates can be automated (CI/CD) or manual (human review).

Vector Embedding (Ch 6)
: A dense numerical representation of text in high-dimensional space. Used for approximate nearest-neighbour search. Contrasted with Terraphim's deterministic graph embeddings.

Verification Bottleneck (Ch 22)
: The fundamental economic constraint where verification costs grow faster than generation costs, creating pressure to deploy unverified AI outputs.

Vocabulary (Ch 8)
: A human-readable collection of defined terms for a domain. Less formal than an ontology, more structured than ad-hoc terminology.

Key Sources

- Chapters 1-25 of this book
- Appendix D: Knowledge Graph Schema (thesaurus format, synonym groups)
- Terraphim AI technical documentation
- SKOS (Simple Knowledge Organisation System) specification

- “Some Simple Economics of AGI” (verification bottleneck terminology)

Evaluation and Decision Records

A **harness** controls evaluation cases, execution and result capture. A **grader** applies an assessment rule to an output and its evidence. A **reference label** is an independently specified expected assessment, with its review status recorded. **Calibration** relates stated probabilities to observed event frequencies for a defined workload. **Abstention** records that the assessment mechanism declines to select a substantive answer.

A **verdict** is an assessment result; a **decision** records a selected course of action under stated constraints. **Promotion** makes an assessed candidate eligible for a specified operational use through an authorised change process. **Provenance** records origins and transformations; **evidence lineage** links claims and decisions to their supporting observations. A **decision trace** joins the requirement, service, assessment, authority and observed outcome. A **belief** is a revisable representation of what an agent takes to be true; an **intention** is a commitment to a course of action. These modelling terms do not replace stakeholder concerns or requirements. Chapters 15–17 and Appendix H give the corresponding contracts.