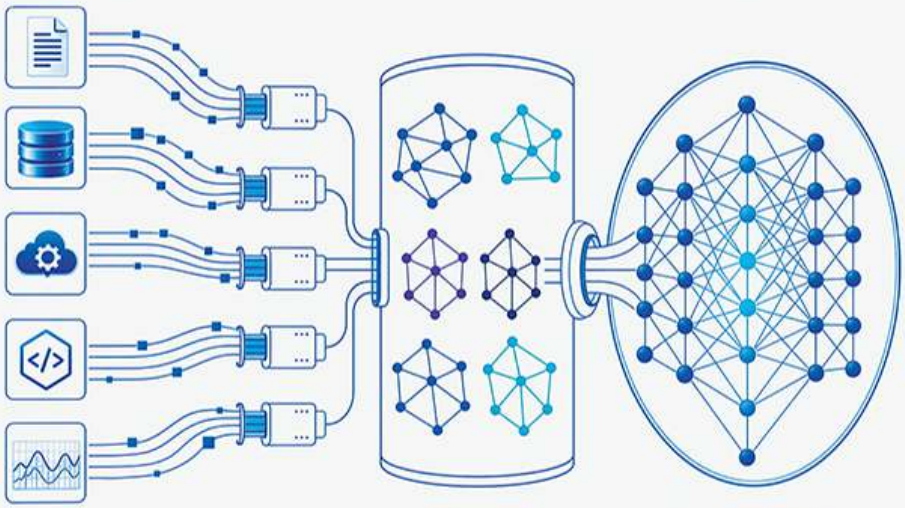
<packt>

Data Strategy for LLMs

Build reliable AI applications with RAG, fine-tuning, evaluation, and context engineering





Foreword by Yash Sheth, Co-founder & CTO, Galileo (acquired by Cisco)

Eslam Kamal | Rany ElHousieny

Data Strategy for LLMs

Build reliable AI applications with RAG, fine-tuning, evaluation, and context engineering

Eslam Kamal
Rany ElHousieny



Data Strategy for LLMs

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty

Relationship Lead: Sanjana Gupta

Project Manager: Shashank Desai

Content Engineer: Tiksha Abhimanyu Lad

Technical Editor: Seemanjay Ameriya

Indexer: Hemangini Bari

Production Designer: Vijay Kamble

Growth Lead: Ankur Mulasi

First published: November 2026

Production reference: 1231026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80611-699-7

www.packtpub.com

To Clearwater Analytics and its leadership, for giving me the place to learn what this book teaches by building it.

— Rany

To my family, for their encouragement, and to my colleagues, mentors, and leaders at Microsoft and Meta, for their support and the shared lessons that shaped this book.

— Eslam

Foreword

Over the past decade, I have witnessed the landscape of artificial intelligence shift from the brittle heuristics of early speech recognition to the fluid, generative capabilities of modern large language models (LLMs). In the early days of conversational AI, we were often obsessed with the "black box"—the model architecture itself. We spent countless hours tuning hyperparameters and designing complex neural layers, often treating the data as a static input for benchmarking purposes. However, as the field has matured, we have arrived at a critical realization: while model architectures evolve with breathtaking speed, the enduring foundation of any reliable, production-grade application is its data and evaluation strategy.

From our early days building Galileo, we have always focused on the data-centric AI approach to help our customers build some of the most complex production-grade agents in the industry. Even with today's pace of model evolution, we can comfortably say that we are at "peak intelligence" for countless AI applications that are moving the industry forward in a meaningful way. But on the other side, there are even more POCs of AI applications that are not being productionized at all. This is primarily due to the lack of a strong data and context strategy for their specific use case.

In *Data Strategy for LLMs*, Rany ElHousieny and Eslam Kamal provide the industry with a timely and essential roadmap. Drawing from their deep experience in building AI agents at scale, they bridge the gap between theoretical potential and practical execution. This book is not merely a collection of techniques; it is a useful guide to the lifecycle of the data that defines LLM / Agent performance as we know it and as it will continue to be.

I encourage you to apply these learnings in mastering the craft of building robust AI systems. Whether you are an AI engineer, a business leader, or just someone trying to build something cool and useful with AI, the insights within these pages will sharpen your ability to implement LLMs in real-world scenarios with rigor and foresight. Mastering data strategy is no longer optional; it is the prerequisite for building intelligent applications.

Yash Sheth

Co-founder and CTO, Galileo(acquired by Cisco)

Contributors

About the authors

Eslam Kamal is an applied AI leader with 20 years of professional experience, including roles at Microsoft and Meta. His work spans natural language understanding, responsible AI, multimodal AI, and conversational AI systems. At Microsoft Research, he worked on language understanding. He later led applied science work across Microsoft Copilot, GitHub Copilot, and Azure Responsible AI. At Meta, he focused on conversational quality for Meta AI on Ray-Ban Meta glasses. He subsequently returned to Microsoft to lead applied AI for Dynamics 365 Contact Center. His experience has shaped a practical focus on how training and evaluation data influence model behavior and the reliability of deployed AI systems.

I would like to thank my wife, Manar, my children, Nada and Fares, and my parents for their patience, encouragement, and understanding throughout the writing of this book, especially during the evenings and weekends it took away from our time together. I am grateful to my co-author, Rany ElHousieny, for his collaboration and the shared effort that brought this book to life. My thanks also go to my friend, Hassan Fayek, for the valuable insights and honest feedback. I thank my leadership teams at Microsoft and Meta for supporting me throughout this writing journey. Finally, I am grateful to Tiksha Abhimanyu Lad at Packt for her editorial guidance, and to Shashank Desai and Sanjana Gupta for coordinating the project and supporting us throughout the publishing process.

Rany ElHousieny is an AI solutions architect and senior AI engineering manager with over two decades of experience in AI, NLP, and ML. He spent 20 years at Microsoft, including time at Microsoft Research. Currently, he manages generative AI engineering teams at Clearwater Analytics, building retrieval, context engineering, and multi-agent systems for investment management. He co-authored an AWS Machine Learning Blog post on Clearwater's generative AI work. On Medium and LinkedIn, he writes regularly about AI systems architecture, RAG, multi-agent orchestration, context engines, and knowledge graphs. He also created the Agentic Repos

framework that *Chapter 11* builds on and was the technical reviewer of *LLM Engineer's Handbook* (Packt).

I would like to thank Sandeep Sahai, Souvik Das, Apurva Shetye, and Bharat Mabbu at Clearwater Analytics for their bold vision, which put AI into our everyday engineering work early and kept us ahead of the industry. Much of what this book teaches came from building under that direction. Thanks also to my Clearwater colleagues whose published work I cite in these chapters, and to my co-author, Eslam Kamal, for picking our collaboration back up after more than a decade. At Packt, I am grateful to Tiksha Abhimanyu Lad for her careful editing, to Shashank Desai and Sanjana Gupta for keeping the project on track, and to our technical reviewers, whose comments made every chapter better. Most of all, thank you to my family for their encouragement while I wrote this book.

About the reviewer

Nithin Kakani is a Staff Data Engineer at Walmart Global Tech, where he builds large-scale data platforms spanning AdTech, ML pipelines, and associate data systems. His work focuses on the data infrastructure that powers production machine learning and AI systems, from ingestion and pipeline design to privacy-aware architecture. He is the author of an open-source privacy framework for AI systems and a peer-reviewed paper on privacy-preserving AI architectures for multi-tenant data platforms. Nithin brings a practitioner perspective to technical review, grounded in building and operating the kinds of systems this book describes.

Table of Contents

Preface

Free benefits with your book

Part 1: Building the Data Foundation

Chapter 1: Getting Started with LLMs: From First API Call to Data Strategy

Technical requirements

How this chapter fits into the book •

Getting started: setting up OpenAI API

Environment setup •

Option 1: Google Colab (recommended for beginners) •

Option 2: Automated Local Setup (Optional for advanced users) •

For macOS users •

For Windows users •

For both Windows and Mac users: •

Calling the OpenAI API •

Understanding our core AI function •

In-context learning

The journey: From basic LLM to Intelligent Assistant

Stage 1: Direct LLM Integration (Chapter 1) •

Stage 2: Data Collection (Chapter 2): Building Your Knowledge Foundation •

Stage 3: Data Preparation (Chapter 3): From Raw Data to Vector-Ready Knowledge •

Stage 4: Retrieval-augmented generation (Chapters 4 and 5): building and refining retrieval systems •

Stage 5: Synthetic Data Generation (Chapter 6): Enhancing Your Knowledge Base •

Stage 6: Post-training and Model Customization (Chapters 7 and 8): Changing Model Behavior Through Data •

Stage 7: Evaluation Data (Chapter 9): Measuring Whether the System Works •

Stage 8: Data Quality and Human Oversight (Chapter 10): Keeping Production Systems Reliable •

Stage 9: Context Engineering (Chapter 11): Delivering the Right Data at Runtime •

Summary

References

Chapter 2: Data Sources

Technical requirements

Using web data

Types of web data •

Example web data collections •

Common Crawl •

Wikipedia •

Project Gutenberg •

OpenWebText •

BookCorpus •

Challenges in collecting web data •

Best practices for collecting web data •

Acquiring domain-specific content

- Types of domain-specific content •
  - *Medical domain data* •
  - *Financial domain data* •
  - *Legal domain data* •
  - *Scientific research data* •
  - *Software development data* •
- Benefits of domain-specific content •
- Challenges and considerations •
- Best practices for using domain-specific content •
- Applications of domain-specific content •

Processing HTML and PDF documents

- Challenges of parsing HTML documents •
- Challenges of parsing PDF documents •
- HTML and PDF processing techniques •
  - *HTML parsing* •
  - *PDF parsing* •
  - *Optical character recognition (OCR)* •
  - *Data extraction* •
- Best practices for processing HTML and PDF •

Leveraging public datasets

- Benefits of public datasets •
- LLM training datasets •
- LLM benchmarking datasets •
- Public dataset selection •

Exploring licensed data

- Benefits of licensed data •

Common types of licensed data •

Guidance for using licensed data •

Selecting relevant data sources

Data sources for assistant use cases •

Personal assistant •

Domain-specific assistant •

Enterprise assistant •

Content moderator •

Best practices when selecting data sources •

Summary

References

Chapter 3: Data Preparation: From Raw Data to Vector-Ready Knowledge

---

Vectorization and embedding fundamentals: the language of AI •

Choosing your first embedding model: OpenAI's text-embedding-3 series •

Your first embedding: A practical example •

A quick word on other embedding models •

The indexing pipeline: building our knowledge catalog •

The code: Step-by-Step implementation •

Chunking •

The naive approach: fixed-size chunking •

A smarter approach: semantic chunking •

LLM-driven chunking: the agentic approach •

The final layer: hierarchical and context-aware chunking •

Graph database integration: connecting the dots •

What is a graph database? •

Vector database optimization: from prototype to production •

The Goldilocks problem: optimizing chunk size and overlap •

Beyond the text: metadata enrichment •

Data ingestion and indexing pipeline: getting it all together •

Putting theory into practice •

Summary •

Further reading and references •

Part 2: Delivering Knowledge at Inference Time

Chapter 4: Building Your First RAG System: From Theory to Implementation

The two halves of a RAG system: indexing and retrieval

The retrieval Pipeline: finding and using knowledge

Coding our RAG function •

Understanding the Q1/Q2 Example •

Common pitfalls in first-time RAG builds

Graduating to a framework

Why use a Framework? •

Managed RAG and enterprise search options

How to choose (practical guidance) •

Cost and ops trade-offs (at a glance) •

Choosing a vector database (Top 5)

Cloud-Native Vector options (AWS, Elastic, MongoDB) •

Applying RAG to the four case studies

Summary

Chapter 5: Data Strategy for Advanced RAG: When Your Data Tells

You What You Need

Technical requirements

Detecting and correcting temporal data drift

Adding keyword exact match retrieval

Adding a second source exposes volume and schema mismatches

Handling multi-hop failures when answers span multiple systems

Building your data architecture

- Deciding what goes in the schema •
  - *Handling access control* •
  - *Storing entity links for multi-hop retrieval* •
- Choosing a chunking strategy •
  - *Structure-aware chunking* •
  - *Contextual chunking* •
  - *Chunking code* •
- Designing index architecture •
- Understanding what it costs •
- Establishing governance •
- Routing and self-correcting retrieval •
- Retrieving code •
- Comparing the four use cases •

Summary

References and further reading

Part 3: Shaping LLM Behavior with Data

Chapter 6: Synthetic Data Generation

What is synthetic data

- Why synthetic data has become essential •
- Visualizing the role of synthetic data •
- Enterprise example: HR policy queries •
- Synthetic data as a strategic complement •
- Domain example: content moderation challenges •
- Visualizing complementary roles in training •

Question–answer generation

- Why QA generation matters •
- Techniques for synthetic QA generation •
- Enterprise example: HR policy FAQs •
- Domain example: medical knowledge expansion •
- Implementing QA generation in practice •

Utilization of structured data

- Structured data in enterprise systems •
- Transforming structured data •
- Enterprise example: HR policy conversion •
- Enterprise example: financial data transformation •
- Building synthetic QA from structured data •

Preference alignment data

- Synthetic preference alignment in action •
- Enterprise example: aligning HR responses for clarity •
- Domain example: moderation messages with the right tone •
- Generating preference pairs programmatically •

Data augmentation for robustness

- Techniques for Robustness •
- Enterprise example: overcoming OCR challenges •
- Personal assistant example: handling speech-to-text errors •

Implementing text augmentation techniques •

Multimodal data augmentation

Why multimodal augmentation matters •

Techniques for multimodal augmentation •

Domain example: content moderation across modalities •

Building multimodal augmentation workflows •

Human-in-the-loop strategies

Why human oversight matters •

Hybrid approaches •

Visualizing human-in-the-loop workflows •

Enterprise example: legal review for compliance QA •

Summary •

References

Chapter 7: Understanding LLM Post Training

Technical requirements

The role of post training

Exhausting inference-time controls

Structuring instructions for behavior •

Using few-shot examples •

Retrieval as a behavioral control •

Using tools for actions and computation •

When is post training required? •

The post-training spectrum

Supervised fine-tuning (SFT) •

Low-Rank Adaptation (LoRA) •

Preference Alignment and Reinforcement Learning •

Comparing techniques in practice •
Reading the Spectrum •
Applying supervised fine-tuning
Adapting models with LoRA
Optimizing agentic behavior with RL
Managing data scarcity and overfitting
Integrating post-training into learning loops
Summary
References

Chapter 8: LLM Model Customization

Understanding model customization
Customization as a System-Level Process •
Data as the foundation of customization •
Dimensions of Domain Specialization
Knowledge and terminology •
Workflows and decision making •
Policies and compliance requirements •
Building customization data
Knowledge data •
Behavioral data •
Preference and feedback data •
Synthetic data •
Data quality requirements •
Designing the customization workflow
Understanding the customization life cycle •
Connecting data to the customization workflow •

Starting with inference-time controls •
Expanding through retrieval and grounding •
Scaling the customization data •
Escalating to post-training •
Using evaluation as a gating function •
Implementing the Customization Workflow
Defining target behaviors •
Curating customization data •
Selecting few-shot examples •
Assembling inference-time controls •
Integrating retrieval and grounding •
Deploying the customized assistant •
Customization patterns for AI assistants
Summary
References
Part 4: Building and Maintaining Reliable LLM Systems
Chapter 9: Task-Specific Evaluation Data
Technical requirements
Using LLM-as-a-judge
Step 1: building the gold dataset •
Step 2: defining two prompts to compare •
Step 3: running both prompts through the evaluation function •
Step 4: reading the regression report •
Evaluating the RAG pipeline
Step 1: loading the ChromaDB from Chapter 3 •

- Step 2: building the RAG golden dataset •
- Step 3: running the full pipeline and scoring •
- Step 4: reading the summary •
- Step 5: running RAGAS automated metrics •
- Step 6: reading the combined report •

Evaluating and using Chapter 6's synthetic data

- Faithfulness •
- Answerability •
- Diversity •
- Preference pairs and the position bias check •

Evaluating the fine-tuned model from Chapter 7

Evaluating live production data

- Applying a production evaluation platform •

Building task-specific evaluation datasets

- Understanding why task-specific evaluation matters •
- Sourcing data for evaluation sets •
- Defining the evaluation dataset schema •
- Creating the golden dataset •
- Prioritizing quality over quantity •
- Determining evaluation set size •
- Example: building starter sets for the four assistant types •

Selecting evaluation metrics by assistant type

- Applying general metrics across all assistants •
- Seeing the difference in the notebook •
  - *Step 1: defining the multi-metric judge •*
  - *Step 2: scoring prompt B responses •*
  - *Step 3: scoring Prompt A responses •*

Step 4: comparing basic score vs. multi-metric score •

Step 5: interpreting what the extra metrics caught •

Personal assistant metrics (case study 1) •

Domain-specific assistant metrics (case study 2: medical diagnosis) •

Enterprise knowledge assistant metrics (case study 3) •

Content moderation system metrics (case study 4) •

Metric selection framework •

How to evaluate evaluation platforms

Core capabilities •

Integration and developer experience •

Production readiness •

Evaluating costs at scale •

Differentiation •

Lock-in and portability •

From criteria to decision •

Summary

References

Chapter 10: Data Quality and Human Oversight

Identifying data quality problems

Diagnosing and correcting data failures

Tracing failures to their sources •

Correcting the root cause •

Rebuilding affected data •

Scaling human oversight

Using automated judges for scale •

Prioritizing human review •

Calibrating the quality process •
Storing LLM data
Separating source and derived data •
Managing vector indexes •
Selecting data infrastructure •
Governing and monitoring data quality
Enforcing data quality gates •
Tracking metadata and provenance •
Data versioning •
Monitoring data quality •
Closing the data quality loop
Summary
References

Chapter 11: Context Engineering: Data Strategy at Agent Runtime

Technical requirements
The junior new session
Understanding the knowledge map
Organizing into three tiers •
Reading what a signpost has to say •
The index writes itself •
Walking it by hand •
Encoding the same walk in Python •
Facts that expire
Keeping the dead file •
Rules the agent follows
Whether it actually obeys •

Representing people and systems
Growing junior to senior
Generate, then govern •
Understanding when tokens become critical •
The discipline of context engineering
Applying the pattern across domains
Applying the same tiers to an engineering repository •
Revealing the search the finance example hid •
Using existing tools and what they cannot build •
Running the pattern in production •
How to build the brain (the knowledge base) •
Maintaining session continuity •
The forward-deployed engineer •
Starting your own map •
Applying the pattern across the four use cases
Summary
References
Chapter 12: Unlock the Code Bundle and the PDF Version
Unlock this book's free benefits in three easy steps
Other Books You May Enjoy
Index

Preface

This book has two authors. Where we use "I," it refers to the author sharing that experience. This preface is written in Rany's voice.

Most teams who build applications with large language models start with the model. They compare benchmarks, context windows, prices, and release notes. These choices matter, but they rarely create a lasting advantage because the same foundation models are available to competitors and the leading model can change quickly.

The data surrounding the model is different. It determines what the system knows, what evidence it can retrieve, how it should behave, and how the team measures whether it is working. Source documents, training examples, prompts, retrieved context, tool outputs, preference signals, evaluation cases, and production feedback turn a general model into a useful system.

This distinction became clear to me while building enterprise AI systems at Clearwater Analytics. Our work combined retrieval, domain adaptation, evaluation, and specialized agents. In one project we published on the AWS Machine Learning Blog, we fine-tuned GPT-J-6B with domain adaptation using Amazon SageMaker JumpStart and public help content, then compared it with the base model and a larger commercial model using retrieval. The work showed how training data, evaluation questions, retrieval design, latency, and cost must be considered as one system. Clearwater's Crystal assistant also achieved efficiency gains between 25 and 43 percent. The same article reported that Clearwater increased assets under management by more than 20 percent without increasing operational headcount (`https://aws.amazon.com/blogs/machine-learning/how-clearwater-analytics-is-revolutionizing-investment-management-with-generative-ai-and-amazon-sagemaker-jumpstart/`). These were reported production results, not benchmark improvements.

Working on those systems also changed how I thought about where the results came from. Swapping a base model was often a configuration change followed by testing. Improvements to chunking, labeling, filtering, or

evaluation could have a larger effect because they changed what the model received and what the team could measure. The progression happened quickly. In July 2024, I was writing about retrieval-augmented generation and the role of external knowledge in improving accuracy and freshness (https://www.linkedin.com/pulse/retrieval-augmented-generation-rag-langchain-refining-rany-adgmc/). By May 2025, my focus had moved to multi-agent orchestration and CWIC Flow, where specialized agents use tools, skills, memory, and governed access to data (https://cwan.com/resources/blog/multi-agent-systems-and-cwic-flow-pioneering-ai-orchestration/). Each new capability increased the cost of weak data. An agent that retrieves the wrong document may act on it. A fluent answer may still fail a domain-specific evaluation. The models and frameworks kept changing, but the need to select, structure, test, and maintain the right data remained.

Eslam Kamal reached the same conclusion from another part of the stack. We worked together at Microsoft Research on language understanding across languages and domains. Eslam later helped create Microsoft's language services. He then led applied science work across Microsoft Copilot, GitHub Copilot, and Azure Responsible AI.

At Meta, Eslam focused on conversational quality for Meta AI on Ray-Ban Meta glasses. This work required the AI assistant to interpret multimodal context and respond naturally in real time. Then, he returned to Microsoft to lead applied AI for Dynamics 365 Contact Center. Across many task-oriented multimodal assistants, he encountered the same recurring challenge: obtaining the data needed to teach the intended behavior, measure failures, cover difficult cases, and support decisions in production.

These two paths shape this book. One begins with enterprise applications, retrieval, evaluation, and agents. The other begins with model development, synthetic data, post-training, and multimodal systems. They converge on the same conclusion: the quality of an LLM-powered system depends on the data lifecycle around the model as much as, and often more than, the model itself.

Control over the data lifecycle does not imply legal ownership of every record. Some data is licensed or controlled by customers. Some data carries privacy, security, or regulatory restrictions. A collection of documents is not

automatically a durable advantage either, since documents become stale, permissions change, and competitors may have access to similar sources.

The durable advantage comes from the disciplined process around the data. It comes from knowing which sources to trust, how to prepare them, how to respect access permissions and compliance requirements, and how to deliver the right evidence at the right time. It also comes from generating examples for cases that real traffic does not sufficiently cover, building evaluations that represent the application, and feeding production failures back into the system. This process is difficult to copy because it develops inside the organization's workflows and improves through repeated use.

This book approaches that work as one connected lifecycle. Source selection affects preparation; preparation affects retrieval; retrieval affects the answers during evaluation; and evaluation, human review, and production feedback create new data for the next iteration. The focus is on managing data throughout this lifecycle to build and maintain reliable LLM-powered systems.

Examples drawn from four LLM applications connect these ideas to practical scenarios across selected chapters: a personal productivity assistant, a domain-specific assistant, an enterprise knowledge assistant, and a content moderation system. Their different needs for authority, freshness, privacy, coverage, safety, and human review show how the same data decision can affect different systems in different ways.

The code examples in this book use concrete models, APIs, databases, and libraries because practical work requires specific tools. Those tools will change, but the fundamental engineering questions will remain: What data does the system need? Where does it come from? How should it be transformed? Who is allowed to use it? How will the system retrieve it? How will you know whether the results are correct? What happens when the data changes?

Models will continue to improve, and those improvements will be available to almost everyone. Your lasting advantage comes from the data and the operating discipline around it. This book is about building that advantage.

Who this book is for

This book is for AI and machine learning engineers, data engineers, and data scientists who build and operate LLM-powered applications. It also helps technical leads, architects, and AI product managers understand how data decisions affect application behavior and reliability. Advanced students and business domain experts who collaborate with AI teams can use it to understand the data requirements and tradeoffs involved in developing these systems.

Basic familiarity with large language models and generative AI is recommended. To follow the practical examples, readers should understand basic machine learning concepts, such as training and evaluation, and be comfortable reading and modifying Python code. Prior experience with fine-tuning or retrieval systems is not required.

What this book covers

Chapter 1, *Getting Started with LLMs: From First API Call to Data Strategy*, takes you from API setup to a working model call, shows how supplying context changes the answer, and introduces the data strategy developed throughout the book.

Chapter 2, *Data Sources*, covers the sources used for model training and LLM applications, including web data, domain-specific content, public datasets, and licensed data.

Chapter 3, *Data Preparation: From Raw Data to Vector-Ready Knowledge*, explains how raw content becomes usable through extraction, cleaning, chunking, embeddings, and graph-based representations.

Chapter 4, *Building Your First RAG System: From Theory to Implementation*, guides you through building a retrieval-augmented generation (RAG) system using the prepared knowledge base. You will implement semantic search, assemble retrieved content into a prompt, and generate answers grounded in that context.

Chapter 5, *Data Strategy for Advanced RAG: When Your Data Tells You What You Need*, examines what happens when retrieval meets production data, including stale documents, identifiers that challenge semantic search, source imbalance, and questions that require evidence from several systems.

Chapter 6, *Synthetic Data Generation*, shows how synthetic data can extend coverage, clarify behavioral expectations, and create examples that would be difficult or unsafe to collect.

Chapter 7, *Understanding LLM Post Training*, explains when post-training is justified and introduces supervised fine-tuning, parameter-efficient adaptation, preference optimization, and reinforcement learning, connecting each approach to its data requirements and operational tradeoffs.

Chapter 8, *LLM Model Customization*, brings prompting, retrieval, tools, evaluation, synthetic data, and post-training together as parts of a system-level customization process.

Chapter 9, *Task-Specific Evaluation Data*, treats evaluation as a data asset designed around application requirements and risks rather than as a mere

score.

Chapter 10, Data Quality and Human Oversight, traces quality failures to their sources and shows how automated checks, human oversight, correction, versioning, and monitoring work together.

Chapter 11, Context Engineering: Data Strategy at Agent Runtime, develops the Agentic Repo approach to context engineering, showing how to organize project knowledge, preserve decisions across sessions, and select trustworthy context for an agent's next action.

To get the most out of this book

The practical code examples use Jupyter notebooks that you can run in Google Colab or a local Python environment. Local execution requires Python 3.10 or newer. The repository includes setup scripts for Windows and macOS/Linux that create a shared environment and install the core dependencies, including the OpenAI Python library, ChromaDB, and PyTorch. You will need an internet connection to install packages, download models, and access external data and APIs. Google Colab requires a Google account. *Chapter 1* walks you through the setup. The API-based examples require an OpenAI API key and available API credit or billing access. Running the bundled examples with the default model selections incurs small API usage charges. Costs depend on the selected models and how often you run the examples. Follow the repository's API-key configuration instructions and monitor your usage while experimenting.

The small-model training examples in *Chapter 7* are designed to run on a CPU; a GPU is optional for these examples but can accelerate training. Larger models may require additional memory and GPU resources. Some notebooks install additional packages as needed. Run each notebook's setup cells before its examples and follow instructions for loading sample data or reusing outputs from earlier chapters. If your output differs from the book, check the *What changed since the book was printed* section of the repository README.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. Downloading the

bundle is recommended so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com.**
2. Click your **profile picture** and select **Your Orders.**
3. Find this book and click **Download Code.**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book.
3. Sign up or log in to your free Packt account.
4. Upload your proof of purchase and download the code bundle locally.

Usage note: You are free to use and modify this code for personal learning and noncommercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to the *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter

handles. For example: "We will be using the function `ask_ai()` that we built in the previous section."

A block of code is set as follows:

```
response = ask_ai("Your prompt here")
print(response)
```

Any command-line input or output is written as follows:

```
cd chapter_01/Jupyter_Notebooks/
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Click on **Create new secret key** and give it a descriptive name."

> **../../EPUB/NOTE**
>
> Warnings or important notes appear like this.

> **../../EPUB/TIP**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit http://authors.packt.com/.

Share your thoughts

Once you've read *Data Strategy for LLMs*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



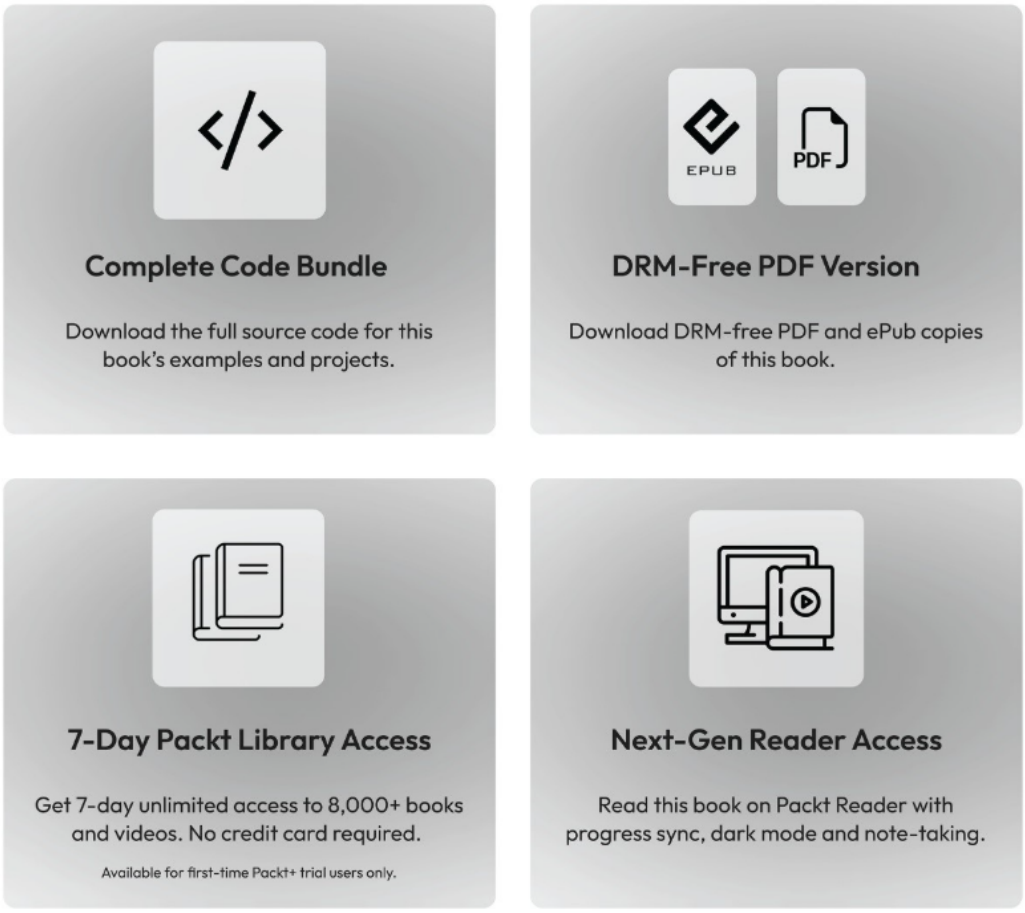
https://packt.link/r/1806116995

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here is a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website do not require an invoice.

Part 1

Building the Data Foundation

The first part of this book establishes the data foundation for LLM-powered applications. You will set up your development environment, make your first model call, and explore how supplying context changes the model's responses. You will then examine different data sources and learn how to select them for your application. Finally, you will prepare that data for retrieval through chunking, embeddings, and metadata, and store it in a searchable knowledge base. By the end of this part, you will have a working baseline and the prepared data needed to build a retrieval layer.

This part of the book includes the following chapters:

- *Chapter 1, Getting Started with LLMs: From First API Call to Data Strategy*
- *Chapter 2, Data Sources*
- *Chapter 3, Data Preparation: From Raw Data to Vector-Ready Knowledge*

1

Getting Started with LLMs: From First API Call to Data Strategy

I started building **machine learning** systems in the early 2000s. What I figured would be hard was the math and the compute. Turns out the hard part was always the data. Most real projects would have a handful of source systems feeding training data, and they rarely agreed on anything. One would spell customer state one way, another would use a different key for the same entity, and there would be a field that the model needed that only existed in the system nobody owned. That data-mapping work was the work, not the algorithm. Two decades on, I am watching the same pattern play out on every serious **large language model** (**LLM**) project. A classifier gives you a probability and a wrong answer is pretty obvious in production. An LLM gives you a paragraph of confident prose, and the wrong piece in it is usually a single sentence that the user has to spot themselves.

In 2024, we published a piece with AWS engineering on how Clearwater built CWIC (Clearwater Intelligent Console) Specialists and Crystal, using SageMaker JumpStart to fine-tune GPT-J-6B and the Llama family behind a few of the agents in them. The numbers in that article were real. Assets under management grew by more than 20% without adding operational headcount [13]. Crystal efficiency gains ranged from 25% to 43% across the workflows we measured [1]. What I did not see clearly at the time, and took me most of 2024 to figure out, was that fine-tuning was not what made the systems work. The data layer underneath was. Swap the fine-tuned model for a larger generic one with solid **Retrieval Augmented Generation** (**RAG**) on top, and most of the quality holds, provided the data side of the pipeline functioned well, the corpus was curated cleanly, and the retrieval context was populated with the

right metadata at ingest time. I wrote about the fine-tuning. This book is mostly about the part I had not written about yet.

That is what this chapter is about. You start with the smallest useful LLM application: setting up API access, making a direct model call, and then adding context so the model can use information supplied at runtime. The Clearwater work behind CWIC and Crystal is where the value of context and data became clearest to me [2]. The chapter also introduces the four assistant use cases that return throughout the book and maps the nine stages from data collection to context engineering. By the end, you will have a working baseline and a clear view of how the remaining chapters build on it.

Every chapter of this book covers four types of AI assistants. I chose them because they each break on a different part of the data layer, and between them, they cover most of what I have seen fail in real deployments:

- **The Personal Productivity Assistant**: Extreme personalization with heterogeneous private data
- **The Domain-specific Assistant**: Expert-level accuracy with specialized knowledge domains
- **The Enterprise Knowledge Assistant**: Navigating data silos with strict access controls
- **The Content Moderation System**: Real-time decisions at massive scale with evolving policies

All four begin with a model and a prompt, but the data decisions underneath are not the same at all. What works for the personal assistant will slow down the enterprise system, and the content moderation setup runs at a scale the other three do not require. You will see that pattern come up in every chapter. The code for each runs in a Jupyter notebook if you want something concrete to follow along with.

Here is what you will do, in order:

- Set up API access and an environment that runs the book's notebooks
- Make a first direct call to the model, with nothing sitting between you and the API
- Work through in-context learning, and what changes when you supply what the model is missing

- Walk the nine stages that carry a bare model call to a production assistant

Technical requirements

This is a hands-on book, and I recommend running the code as you read. Each chapter has a Jupyter notebook that works in Google Colab or in a local Python environment. The chapter explains the lines that matter to the data strategy, while the notebook contains the complete runnable example and additional code notes. You can find all the notebooks in the book's repository. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*. Run the notebooks, change the inputs, and watch how the model's output changes. Where one of my articles provides useful background without interrupting the chapter, I include it in the references.

../../EPUB/NOTE

If you run into setup problems, Claude Code or OpenAI Codex can help you inspect errors and work through the commands. Google Colab is another option when you want to avoid a local Python setup because it runs the notebook in your browser. I created a Claude Code course for you if you want a starting point and included it in the *References* section [3]. You do not need a coding assistant to complete the chapter. The code is here to make the data concepts concrete.

How this chapter fits into the book

The book runs eleven chapters. The book presents the chapters in this order because each one feeds into the next, and the RAG and fine-tuning work later in the book gets a lot harder if the data foundations here are not solid first.

I have seen it go wrong in a specific way. Teams jump into retrieval before they know what they actually have, and the problems they end up chasing were already baked in by *Chapter 2*. This chapter gives you the baseline model call; the next chapters add the data collection, preparation, and retrieval work around it.

../../EPUB/NOTE

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Getting started: setting up OpenAI API

For this book, I am using OpenAI's API. The reason I chose OpenAI is that it is well-documented, and I have used it enough to know where the edge cases are. There are other options, such as Anthropic's Claude, Google's Gemini, and local models with Ollama. The code in this book can be adapted to work with any of them. But LLM calls are not really the point here. The point is the data strategy around them.

Here is the step-by-step setup process, which I also documented in an article [11]:

1. **Create your OpenAI account**: Go to `https://platform.openai.com` and select **Sign up**. If you already have an account, select **Log in**. Follow the account creation steps shown on the screen.
2. **Complete account verification**: you can sign up with Google, Microsoft, or email. Complete any verification steps OpenAI presents when you create your first API key. I recommend using your primary development account for consistency.

3. **Add billing information**: OpenAI API usage is billed by model and token usage. Prices change, so check the current model and pricing information at `https://developers.openai.com/api/docs/models` before running the examples. The notebook does not hardcode a chat model. The notebook checks the models available to your account and selects one that responds successfully.

> **../../EPUB/NOTE**
>
> You will have to add money to your credit balance to be able to run the examples in this book. If you did not add credit, you will receive an error when you call the APIs. Go to `https://platform.openai.com/settings/organization/billing/overview` to check your current balance. The page is the OpenAI billing portal where you can see what you have available and add/buy credits when you need to.

4. **Go to API keys.** Once your account is set up, open `https://platform.openai.com/api-keys` to manage your API keys.



Figure 1.1: API Keys section in your OpenAI dashboard

5. **Create your API key**: Click on **Create new secret key** and give it a descriptive name like `Book Examples` or `Development Testing`:

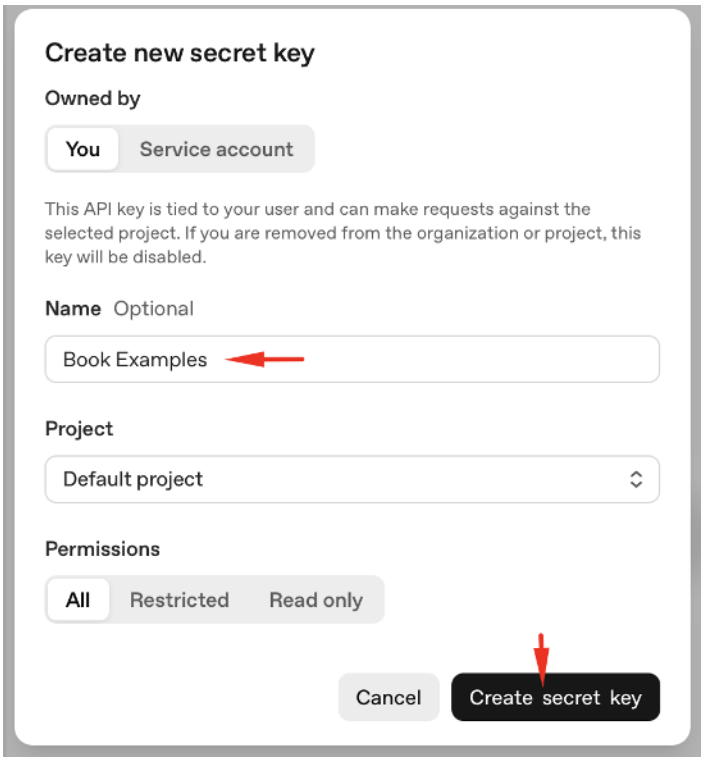


Figure 1.2: Creating a new API key

6. **Copy your API key**: Your API key will start with `sk-` - copy the entire string and store it securely. Please note that you can view this key only once, so save it immediately.

../../EPUB/NOTE

Never commit API keys to version control or share them publicly. You can use environment variables to keep them secure. However, for this book, you will use a simple approach of assigning the API key to a variable. The setup cells in the chapter notebook, `Chapter_01_Notebook.ipynb`, show the secure version, loading the key from a `.env` file through environment variables.

Copy your API key to a safe place. You will need it when setting up the environment.

Environment setup

I've learned through the years and from feedback from learners that the best way to understand how an LLM application works is to build the smallest version yourself. To follow along with the code, you have two main paths.

- If you don't have Python installed, or prefer not to set up a local environment, use Option 1: Google Colab. It's the fastest way to get started and runs entirely in your browser.
- If you are an experienced developer with Python already on your machine, you can use Option 2: Local Development.

Throughout this chapter, you will build a working example that starts with a direct model call and then adds context to the prompt. To follow along, you will need two things: access to an LLM API and a development environment.

You will need to clone the repository from GitHub to your machine:

```
git clone https://github.com/PacktPublishing/Data-Strategy-for-LLMs.git
```

If you do not want to use Git, you can just download the notebook you are using and upload it to Google Colab. Which one makes sense depends mostly on your setup. There are two options:

Option 1: Google Colab (recommended for beginners)

If you are new to Python, or just do not want to deal with setting up a local environment, Google Colab is the easier path. You open it in a browser and it runs. Nothing to install. If you need a Python refresher first, I wrote one [9].

Here is how you get started with Colab:

1. **Google Account**: you need a Google account to access Google Colab. If you don't have one, you can create it for free at `https://accounts.google.com/signup`.
2. **Accessing Google Colab**: Open a web browser and go to `https://colab.research.google.com/`. You will be prompted to sign in with your Google account:

3. **Open the Chapter_01_Notebook.ipynb notebook from GitHub**: After signing in, open the notebook. Most of the time you will have the Open notebook screen below by default. If you do not want to clone the repository or download the notebook, you can still upload or open it directly from GitHub and then put the full path to the `https://github.com/PacktPublishing/Data-Strategy-for-LLMs/blob/main/chapter_01/Jupyter_Notebooks/Chapter_01_Notebook.ipynb` notebook as shown here:

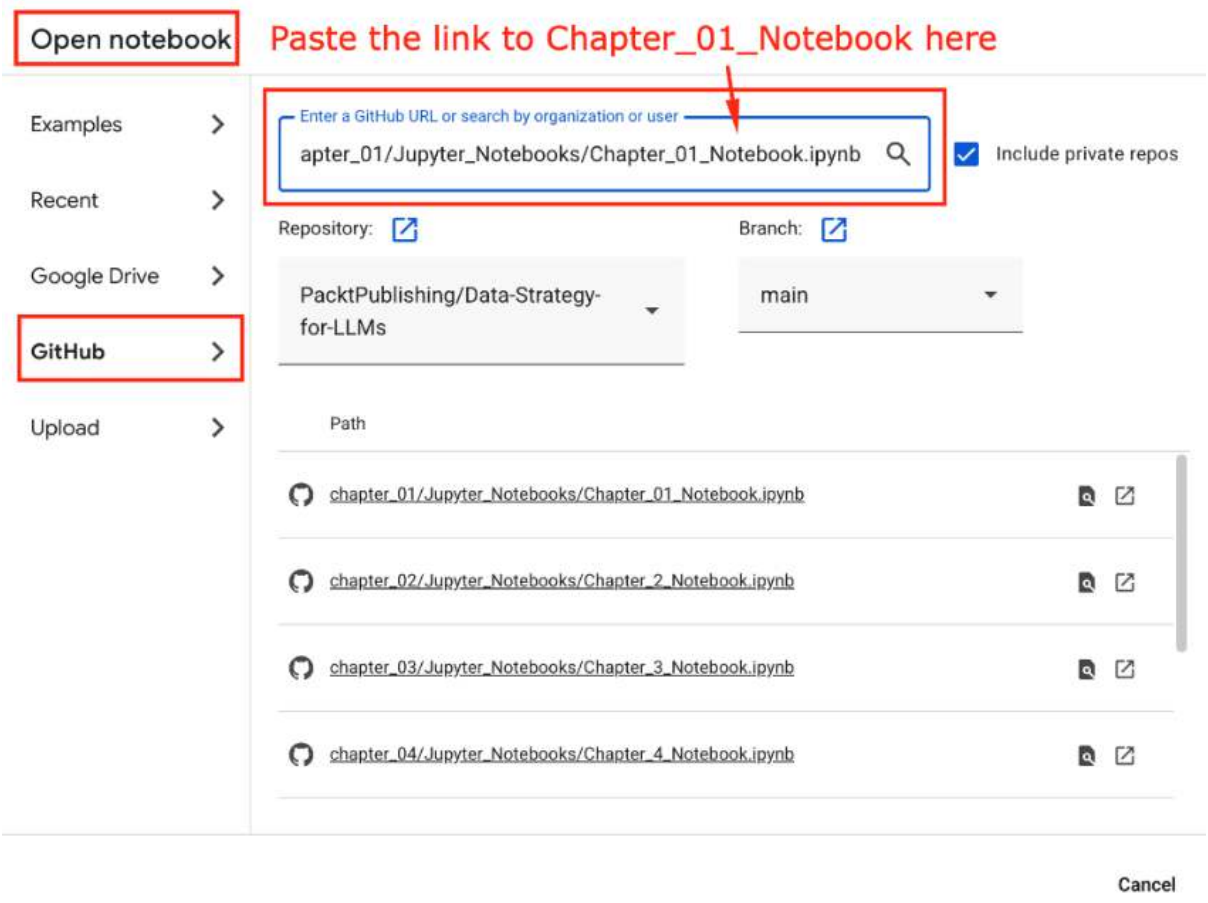


Figure 1.3: Google Colab Open Notebook

If you did not have the previous Open notebook screen by default, choose **File** | **Open** or upload notebook:

../../EPUB/NOTE

If you are new to Colab, you can read the "**Welcome to Colab**" guide to get started or read my article [4].

You can clone or download the notebook locally and upload it to Google Colab. You will find a Jupyter Notebook file named `Chapter_01_Notebook.ipynb` that contains the code you will use in this chapter.

4. First, download/clone the notebook from the GitHub repository or clone the entire repository as explained before. After that, you can open:

```
git clone https://github.com/PacktPublishing/Data-Strategy-for-LLMs.git
```

5. Upload the notebook to your Colab environment and run it to follow along with the code examples in this chapter. This approach is the easiest way to get started if you do not have previous experience or do not want to set up a local environment. If you open the notebook in VS Code or another code editor, you may also see an **Open in Colab** button, as shown in *Figure 1.4*.

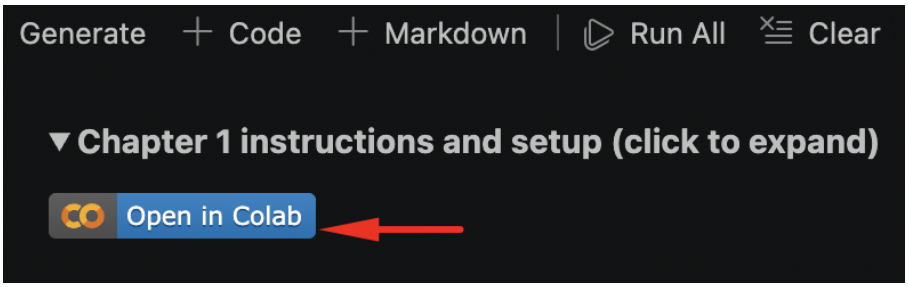


Figure 1.4: Click on the "Open in Colab" Icon in the Notebook

6. When you upload the notebook successfully, click **Run all** at the top of the notebook:

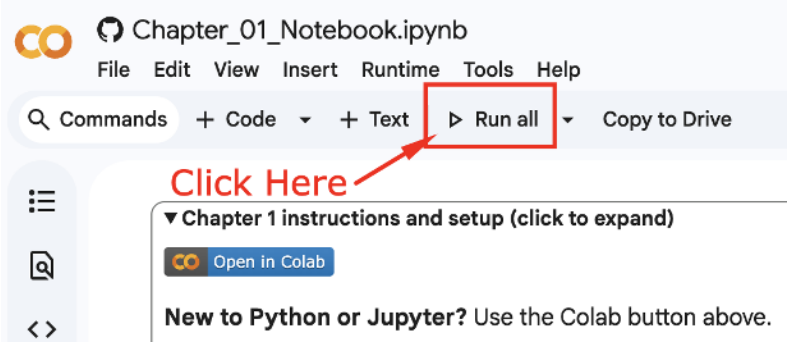


Figure 1.5: Google Colab notebook after upload - select Run all to execute

It will give you a warning; you can accept it to continue:

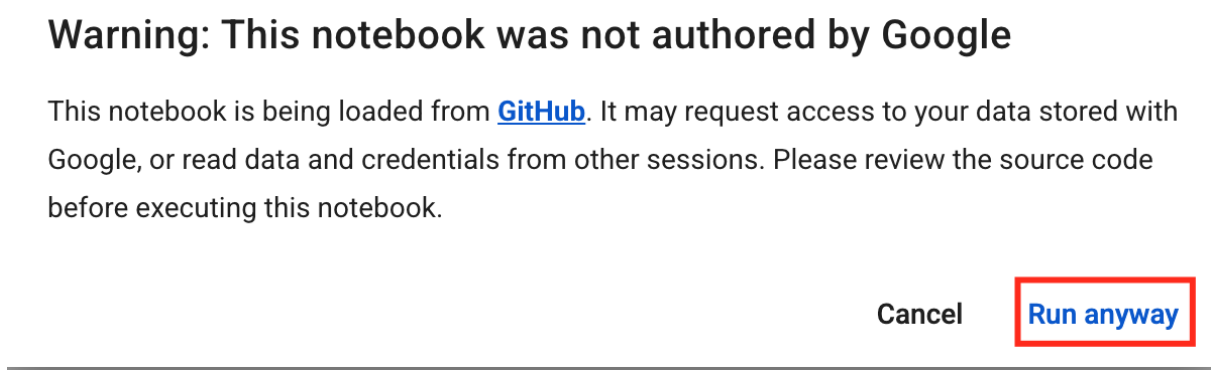


Figure 1.6: This is a warning to review the Notebook

Colab only asks to connect Google Drive when a notebook mounts it. This book's notebook does not, so you will not normally see the prompt below. If a notebook ever asks, grant access only when you expect it.

One of the cells will ask you to enter your API key. Please enter your API key there to save it to the .env file:



Figure 1.7: A Prompt to Enter Your OpenAI API Key

As shown in *Figure 1.7*, your typing stays hidden, so the key does not appear on screen or remain in the notebook output. The cell checks the format before it accepts anything, and it gives you three tries if the paste

goes wrong. Once it accepts the key, it writes it to a .env file, so restarting the kernel does not send you back here.

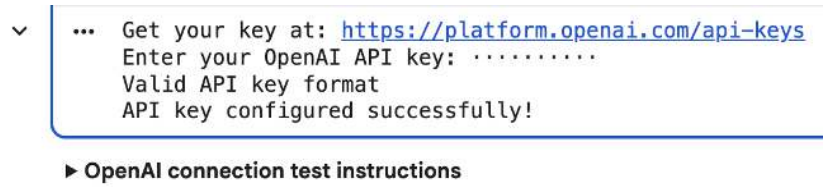


Figure 1.8: Successful entry of your OpenAI API key

Now, you can jump to the Calling OpenAI API section.

Option 2: Automated Local Setup (Optional for advanced users)

For local setup, I have written scripts that automate the whole thing. You do not need to manually install Python, configure environments, or track down packages. Just run the script and it handles all of that. The scripts are in the setup directory of the GitHub repository.

Here are the instructions to set up these files:

All the necessary setup files (`setup_mac.sh`, `setup_windows.ps1`, and `requirements.txt`) are in the setup directory of the book's GitHub repository.

1. Clone or download the entire book repository to your machine.

```
git clone https://github.com/PacktPublishing/Data-Strategy-for-LLMs.git
```

This is what we see:

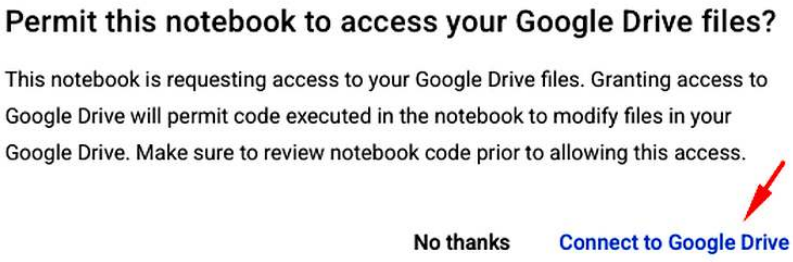


Figure 1.9: Cloning the book code repository

2. Navigate into the repo directory:

```
cd Data-Strategy-for-LLMs/setup
```

3. Follow the instructions for your operating system below.

For macOS users

From inside the root directory, run the following commands:

```
# Make the script executable
chmod +x setup_mac.sh

# Run the script
./setup_mac.sh
```

The script will guide you through installing Homebrew and Python if they are not already present, and then set up your project directory.

```
Github_Repo % cd Data-Strategy-for-LLMs/setup
setup % chmod +x setup_mac.sh

setup % ./setup_mac.sh
Setting up Data Strategy for LLMs book environment...
Checking for Homebrew...
```

Figure 1.10: Setup Commands on Mac

You will be asked to enter your OpenAI API key that you obtained in the previous section:

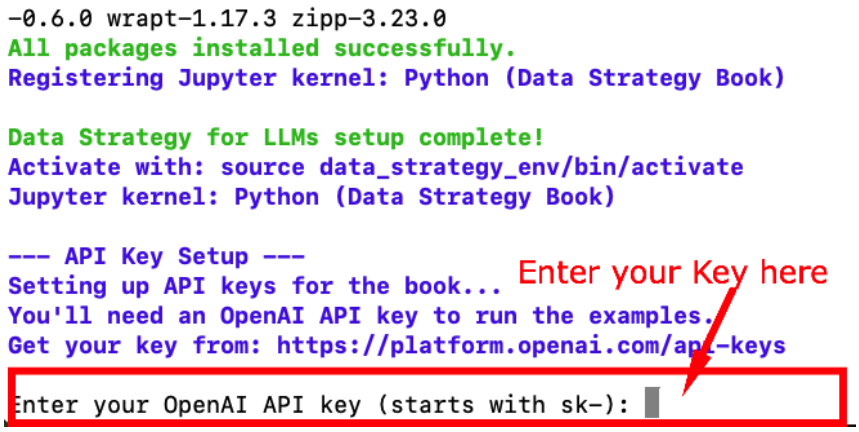


Figure 1.11: Add your OpenAI API Key to the setup

For Windows users

Open PowerShell or Windows Terminal and navigate to the root directory of the cloned book repository. Run the setup script:

```
.\setup\setup_windows.ps1
```

The script uses Windows Package Manager (winget) to install a supported version of Python if one is not already available. If Windows requests permission during the installation, approve the prompt. The script then creates the environment, installs the required packages, and asks for your OpenAI API key:

```
Enter your OpenAI API key (starts with sk-): sk-
```

Figure 1.12: Enter Your API Key

For both Windows and Mac users:

After the script finishes, go to the chapter_01 Notebooks folder to start the Notebook.

1. Go back from setup to root:

   ```
   cd ..
   ```

2. Open the Jupyter Notebooks folder in chapter_01:

   ```
   cd chapter_01/Jupyter_Notebooks/
   ```

3. You can start your IDE from there or open the Notebook from your favorite IDE. You can use Visual Studio Code and start it with `code`:

   ```
   (setup >
   (setup > cd ..
   (Data-Strategy-for-LLMs > cd chapter_01/Jupyter_Notebooks
   (Jupyter_Notebooks > code .
   ```

   Figure 1.13: Change directory to the chapter notebook

   When you open the notebook, perform the following steps:

   The setup script registers the environment as a Jupyter kernel named "Python (Data Strategy Book)." Your VS Code might also suggest the same environment directly as `data_strategy_env` with the Python version next to it. Both point to the same environment the setup script created. VS Code changes its kernel picker from version to version, so the names and the layout you see might not match my screenshots exactly. Do not let that confuse you. Just pick the name closest to the

one in the book, either `Python (Data Strategy Book)` or `data_strategy_env`, and you are good to go:

1. Open the Command Palette (Mac: *Cmd+Shift+P*, Windows: *Ctrl+Shift+P*).
2. Run Developer: Reload Window. You can also press *Cmd+P* (Windows: *Ctrl+P*) and type `>Developer: Reload Window`:



Figure 1.14: Developer: Reload Window

3. After reloading, select **Select Kernel** (top-right):



Figure 1.15: Select Kernel

When you open the notebook, make sure to select the correct environment, Python (Data Strategy Book):



Figure 1.16: Select Jupyter Kernel

Choosing Jupyter Kernel opens the list of environments VS Code can find on your machine. Pick the one the setup created for this book. The Python version and the exact name can differ from what the screenshot shows, so match on the environment name rather than the version number:

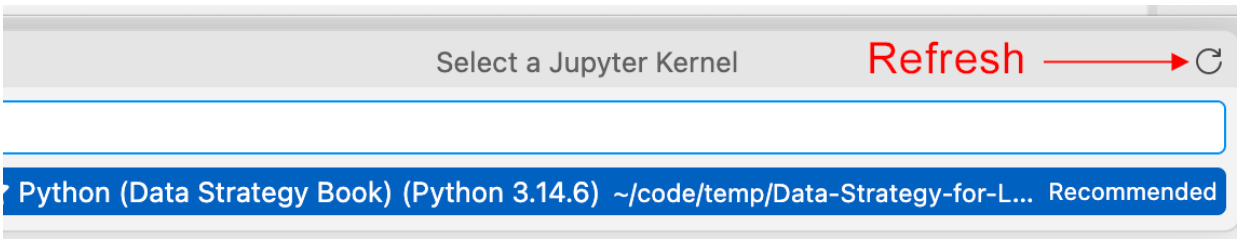


Figure 1.17: Select the Python Data Strategy environment (Note: Python version and name can vary)

You can open the notebooks in Jupyter Notebook or JupyterLab. Start with the `Chapter_01_Notebook.ipynb` notebook. Its first cells fix the Jupyter kernel setup automatically and then guide you through the rest of the chapter.

After selecting the environment, run all the cells:

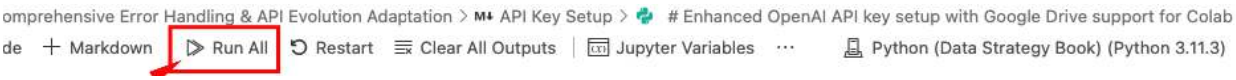


Figure 1.18: run all cells

../../EPUB/NOTE

If you are still facing issues, you should use Google Colab to follow along with the code examples in this chapter.

Calling the OpenAI API

When you run all the cells, it will create two global variables:

- `OpenAI_API_Key`: This will keep your key to use later.
- `MODEL`: The name of the model the notebook selected. The notebook includes a function that checks which models your key can use and selects one.

The following code snippet is a direct, minimal call using the modern OpenAI client: you build a client with `openai.OpenAI(api_key=OpenAI_API_Key)`, request a chat completion with the globally selected `MODEL`, and print the assistant's text via `response.choices[0].message.content`. I use this pattern only to explain the simplest form to call OpenAI APIs. I wrote a short walkthrough of this first call as well [12]. The implementation of the higher-level `assistant.ask_ai(...)` is in the cells above this one. It has proper error checking. You can go through

the full code, if you like, in the notebook. However, all you will need is to call `ask_ai("prompt")`:

```
import openai;
OpenAI_client = openai.OpenAI(api_key=OpenAI_API_Key)

response = OpenAI_client.chat.completions.create(
    model=Model,
    messages=[{
            "role": "user",
            "content": "Tell me a joke"}])

print(response.choices[0].message.content)
```

✓ 1.1s

```
Sure! Here's a joke for you:

Why don't scientists trust atoms?
Because they make up everything!
```

Figure 1.19: Successful Output from the LLM Call

If the cell runs successfully, you should see a joke printed in the output as shown in the preceding screenshot.

The previous code does not have any error handling or retry logic. In production, you should always implement error handling and retry logic. You will find better error handling and retry logic in the cells before this cell. I want to use the simplest way to get to the main point of the book, data strategy.

If this call fails, it could mean that you did not add credit to your OpenAI account. Please add credit to your OpenAI account to be able to run the examples in this book. Now, you can put it in a function to call it in the rest of the book:

```python
import openai

def ask_ai_simple(content, api_key=OpenAI_API_Key):
    return openai.OpenAI(api_key=api_key).chat.completions.create(
        model=Model,
        messages=[{"role": "user", "content": content}]
    ).choices[0].message.content

# Usage:
print(ask_ai_simple("Tell me a joke"))
```

✓ 0.8s

```
Sure! Here's one for you:

Why don't scientists trust atoms?
Because they make up everything!
```

Figure 1.20: The Simplest LLM Call

The previous code is the simplest form you can reach the goal of calling the LLM with a prompt. The main `ask_ai()` is similar but wrapped with lots of error handling. Through the remainder of the book, you will use `ask_ai("prompt")` to call the AI.

Understanding our core AI function

The `ask_ai` function is what you use for the rest of the book to talk to the model. You call it with a prompt and get back text. I built it to keep all the API plumbing out of the way so you can stay focused on the data side.

The function handles all the API mechanics. What you pass in, you get back as text. This arrangement keeps the focus on what the book is actually about: the data strategy around those calls.

../../EPUB/NOTE

Scope of this function: How the OpenAI API works internally is not what this book covers. There are good resources if you want to go deeper on that side, starting with the official API documentation and the Python SDK [7][8]. You will use OpenAI here, but the same data strategy applies to Anthropic, Google, or any local model.

Throughout the rest of the book, every `ask_ai` call we make is a data strategy exercise. The call itself is trivial. What matters is the information we put into the prompt and why it produces better answers than a bare question would.

What you need to know is simply how to call it:

```
response = ask_ai("Your prompt here")
print(response)
```

In the Notebook, search for `ask_ai("tell me a joke")` to find the call to the function. You can change the prompt and try different things.

In the `ask_ai` function, you are sending the entire prompt to the LLM. This is a simple way to get started, but it is not the best way to use LLMs. You will now divide this prompt into a context and a question. The LLM will use this context to answer the question. This process is called **in-context learning**. You can understand that next.

In-context learning

Sending the prompt straight to the model works when the question is general enough that training data covers it. Most of the interesting things we want to build do not fall into that category. In-context learning is the fix: you include the relevant facts or documents in the prompt itself, and the model uses them when generating the response:

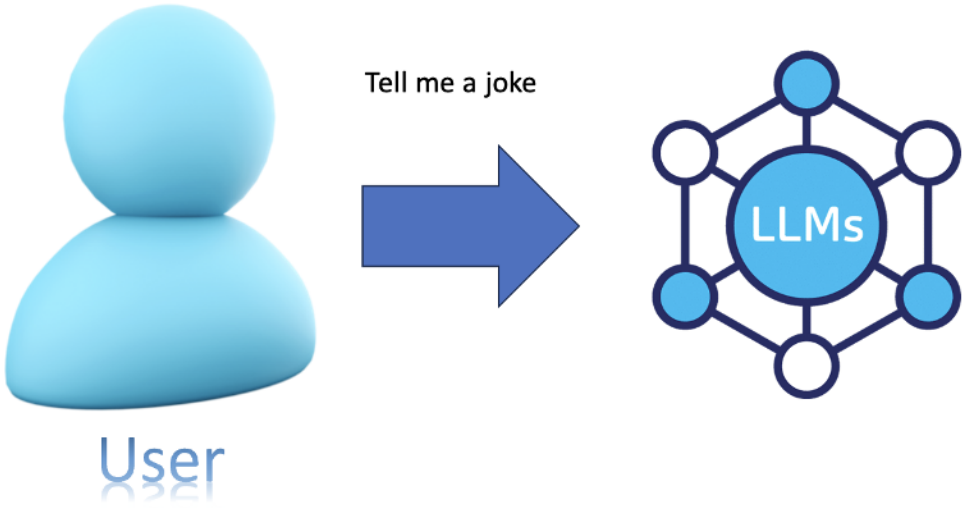


Figure 1.21: Sending the Prompt Directly to the LLM

In my experience building AI systems, the part that actually moves the needle is what you put in the context window. A model trained on the internet does not know your internal data. It does not know what happened last week. That gap is what the context window is for.

That grounding is the starting point, not the whole design. Once you build systems that choose, organize, refresh, and deliver context at runtime, you have moved from prompt engineering into context engineering. *Chapter 11* returns to this idea and treats context as a data layer with sources, structure, evidence, and maintenance [16].

I am spending time on this here because almost everything in the next few chapters is a variation on it. Retrieving documents and injecting them into a prompt, supplying examples to steer behavior, adding a user's history and preferences: same mechanism, different data.

The quality of in-context learning comes entirely from what you put in. Irrelevant context sends the model in the wrong direction. Focused, accurate context makes the response noticeably better. This is why the chapters ahead spend so much time on data collection in *Chapter 2* and data preparation in *Chapter 3*. They determine the quality of context your assistant has to work with, which is what *Chapter 4* retrieves for the context.

RAG builds on this. A query comes in, the system finds the documents most relevant to it, and those get added to the prompt before the model sees the question. The model answers from that retrieved material. If the retrieval is accurate, the answer tends to be too. If it is not, the model falls back on training data and often gets it wrong. The better your data strategy, the more useful that information is, and the more reliable the responses become.

You can discover this in practice. After you have built your basic AI assistant, you will explore how to use in-context learning to improve response accuracy and relevance. You will use the following function: the function `ask_ai()` that you built in the previous section to provide context to the model. You can ask a question that the AI has no information about in its training data: "*Who is Rany ElHousieny?*"

```
print(ask_ai("Who is Rany ElHousieny?"))


# Output:
As of my last update in October 2023, I do not have specific information about an individual named
Rany ElHousieny.
```

Here is the LLM response without context:



Figure 1.22: LLM response without context - demonstrating the limitation of models when asked about information not in their training data

The model has no information about Rany in its training data, so it says so. Refusing the unsupported answer is the right behavior. Now, watch what happens when you add context. You can provide context within your prompt:

```
# Now let's add context to help the model understand who Rany ElHousieny is
context = """
Rany ElHousieny is an AI solutions architect and engineering manager with over 20 years of experience
in AI, NLP, and machine learning. He has led groundbreaking projects at Microsoft, where he advanced
NLP technologies and the Language Understanding Intelligent Service (LUIS). Currently, he drives
GenAI innovation at Clearwater Analytics, focusing on AI-driven financial and investment management
solutions. He is also an author of multiple articles on AI systems architecture and ethical AI
deployment.
"""

prompt = f"""
Context: {context}

Question: Who is Rany ElHousieny?

Please provide a comprehensive answer based on the context provided.
"""

print(ask_ai(prompt))

# Output:
Rany ElHousieny is an experienced professional in the field of artificial intelligence, particularly
in natural language processing (NLP) and machine learning, ...
```

In the Notebook, you can search for `print(ask_ai("Who is Rany ElHousieny?"))` and change the context and the question to try it out.

../../EPUB/NOTE

This output illustrates an important point: it came from a run with an old model, and I kept it on purpose. Run the same cell today and a newer model may give you something better, if it has picked up the name since. Try it with your own name while you are in the notebook. If you have published anything under it, the model may know you too, and if you have not, it will tell you so. That is how public

> information behaves: it arrives on its own schedule, without me touching anything, and it will keep moving under me. What the model still will not do is tell me which parts of its answer hold for my situation. The context I supply is the part I control, and it is the only part that behaves the same tomorrow as it does today.

Here is a version that will not change. Suppose you have a policy at `Rany.ai` that keeps prompt logs from client engagements for 400 days, because the annual client audit covers twelve months and reconciliation adds another five weeks. That number is not a standard. It is ours, it lives in an internal document, and no part of it has ever been on the public internet.

```
print(ask_ai("How long do we retain prompt logs at Rany.ai?"))

# Output:
I can't reliably confirm Rany.ai's current prompt-log retention period from the information available
to me. Please check Rany.ai's latest Privacy Policy or security documentation, or contact
support@rany.ai for the applicable retention period.
```

Two things in that answer matter. The model was right to refuse the number, and it refused cleanly, which is the behavior you want. Then, it invented a support address. There is no `support@rany.ai`. The model could not produce the fact you asked for, so it produced the shape of an answer instead, and the invented part reads with exactly the same confidence as the honest part.

Now, put the policy in the prompt:

```
context = """Rany.ai retains the prompt logs from client engagements for 400 days. The retention
window covers the twelve-month client audit cycle plus a five-week reconciliation period, so the logs
stay available until the audit for that period closes."""

prompt = f"""Context: {context}

Question: How long do we retain prompt logs at Rany.ai?

Please provide a comprehensive answer based on the context provided."""

print(ask_ai(prompt))

# Output:
Rany.ai retains prompt logs for 400 days. This period covers the 12-month client audit cycle plus a
five-week reconciliation period, ensuring the logs remain available until the relevant audit closes.
```

The policy specifies 400 days, with the reason attached. Nothing about the model changed between those two cells. The only difference is that the second one carried the policy. Ask that same model what a consulting company should do in general. That model will tell you thirty days or less, which is sound advice and wrong for us by more than a year. A team that followed it would delete the logs long before the audit that needs them. You would not catch that in review. The logs would already be gone.

The prompt structure above is simple but powerful: you provide context first, then ask your question, and instruct the model to use the provided context. Throughout this book, you will use this same pattern. The only difference is that instead of manually writing the context, RAG systems will automatically retrieve and inject the most relevant information.

Here is how RAG transforms this manual process into an automated, intelligent system:

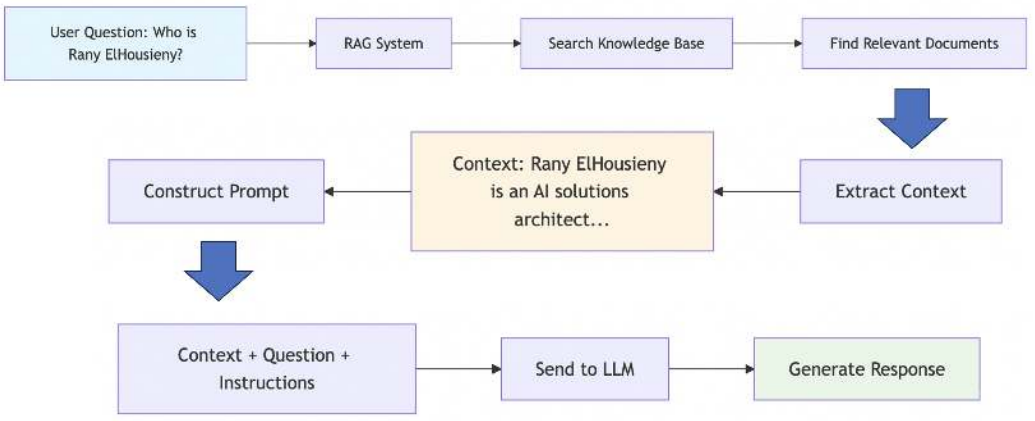


Figure 1.23: RAG System Automating Context Retrieval - From manual context to intelligent document search

Retrieval happens in the middle steps. Instead of manually providing the context, the RAG system searches documents and knowledge bases, finds relevant information, and uses it to construct the context section of the prompt.

Prompt engineering is a topic on its own and not what this book covers. I wrote a guide on prompt engineering frameworks, including ReAct, if you want to go deeper [5]. Our concern here is the data feeding those prompts, not the prompt structure itself.

This process is in-context learning. The model had no answer on its own, we gave it the relevant text, and it used that text to respond. RAG does this same thing at scale. At Clearwater we built an assistant called Crystal that works this way across thousands of documents, retrieving the relevant documents and adding them to the prompt before the model sees the question [1]. I wrote a longer walkthrough of how in-context learning works with LangChain [6], and a broader LangChain guide with examples if you want more [10].

This same principle applies to all four AI assistant use cases you will build throughout this book. The Personal Productivity Assistant uses this for calendar and email data. A user asks about tomorrow's meetings, and the system pulls the calendar entries and puts them in the prompt. The Domain-specific Assistant does the same with financial filings. An analyst asks about Apple's latest SEC filing and the system retrieves that document. The Enterprise Knowledge Assistant searches internal policy documents when someone has a question about company rules. Content Moderation works the same way but pulls policy guidelines and precedent cases so the system evaluates content consistently instead of making one-off judgment calls.

Same pattern every time. Find whatever is relevant to what the user asked, drop it in the prompt, and the model reasons from that instead of guessing. Whether the answer is any good comes down to whether you retrieved the right material, which is basically the rest of this book.

With your development environment ready, you can use the same model call as the starting point for the rest of the book. The next section condenses the eleven chapters into nine stages, showing what each stage adds to the assistant and the data behind it.

The journey: From basic LLM to Intelligent Assistant

Getting from a basic API call to a production assistant is not one big step. It is a sequence of smaller ones, and each step changes what your data needs to look like. The chapters in this book follow that same sequence. Let me walk you through the nine-stage progression that every successful AI assistant goes through.

Stage 1: Direct LLM Integration (Chapter 1)

Every AI assistant starts here. This is exactly what you are building in this chapter. You make direct API calls to an LLM with static prompts and no external data sources. This is where most tutorials end, but it is just the foundation. At this stage, your assistant can only work with information it was trained on, with no access to your specific data or context. This is the ask_ai function we created in this chapter.

For our **Personal Productivity Assistant**, this stage looks like:

```
# Basic LLM call
response = client.chat.completions.create(
     model=MODEL,  # selected at runtime by the notebook setup, never hardcoded
    messages=[
        {"role": "system",
         "content": "You are a helpful productivity assistant."},
        {"role": "user",
         "content": "Help me write an email to my team about the quarterly review."}
    ]
)
```

Ask it about your company's quarterly results and it draws a blank. All it has is what was in the training set.

Stage 2: Data Collection (Chapter 2): Building Your Knowledge Foundation

The model does not know your company. It does not know your tools. We started Crystal on a base generative pre-trained transformer (GPT) model, and when we asked it about Helios, it told us Helios was a figure from Greek mythology. Helios is one of our internal tools, named after the mythological figure, but no public model would ever know that. Darrel Cherry told this same story in his RAG article on the Clearwater engineering blog [15]. Internal documentation is not in anyone's training data. *Chapter 2* is about collecting the data that fills that gap, everything from web scraping to API integrations to internal documents nobody has organized yet.

Stage 3: Data Preparation (Chapter 3): From Raw Data to Vector-Ready Knowledge

Chapter 3 is the preparation step. You cannot feed raw documents straight into a retrieval system and expect good results. Text has to become vectors. Documents have to be split into chunks that keep the context intact. You

vectorize the text so the machine can search it. You chunk the documents, and if the chunks are too small you lose context, too large and retrieval gets noisy. I have broken production systems both ways. Then, there is the vector store itself, ChromaDB for the notebook examples in this book, though Pinecone and Weaviate could show up in production deployments.

Stage 4: Retrieval-augmented generation (Chapters 4 and 5): building and refining retrieval systems

Chapter 4 is where you actually build the retrieval pipeline. The data from *Chapter 3* is sitting in a vector store, and now you need a system that pulls the right chunks and puts them into the prompt before calling the model. You go through semantic search, ranking, and the parts that break first in production.

Chapter 5 picks up where *Chapter 4* left off and starts adding the complications. Old documentation outranks new documentation because the old stuff was written better. Error 502 and error 503 look identical to an embedding model. Jira has dozens of times more chunks than Confluence and drowns it out in retrieval. Each describes a real failure we had to fix, and each one traces back to metadata we skipped at ingest time.

Stage 5: Synthetic Data Generation (Chapter 6): Enhancing Your Knowledge Base

There is a gap problem that every RAG system hits eventually. Users ask reasonable questions and the system returns nothing because the knowledge base simply does not cover that area. You can wait for someone to write the missing documentation, or you can generate it. *Chapter 6* covers that second option. You use the model to produce targeted content for the areas where retrieval fails, and then verify it actually helps before adding it to the index.

Stage 6: Post-training and Model Customization (Chapters 7 and 8): Changing Model Behavior Through Data

At some point RAG is not enough and you need to change the model itself. *Chapter 7* covers when that point arrives and introduces post-training methods, such as supervised fine-tuning and preference alignment. *Chapter 8* then shows that training is only one way to customize. It treats customization as a system-level process: inference-time controls come first, then retrieval, tools, and synthetic data, with post-training added when evaluation shows

the earlier layers are not enough. Together, the two chapters show when to change the model itself and what data each type of customization needs.

Stage 7: Evaluation Data (Chapter 9): Measuring Whether the System Works

Changing model behavior is only useful if you can tell whether the change helped. *Chapter 9* treats evaluation as a data problem. You will build representative test sets, define meaningful measures, and study failure cases to determine whether retrieval, customization, and generation are actually improving. Dan Siddall wrote up the framework we built at Clearwater for scoring LLM output [14]. This stage replaces "it seems better" with evidence you can act on.

Stage 8: Data Quality and Human Oversight (Chapter 10): Keeping Production Systems Reliable

A successful evaluation is not the finish line. Once the system is in production, its data changes, new failure patterns emerge, and some decisions still require human judgment. *Chapter 10* covers how to identify weak, incomplete, or outdated data, capture feedback, and turn production failures into measurable improvements. This stage makes data quality a continuous operational practice rather than a one-time preparation task.

Stage 9: Context Engineering (Chapter 11): Delivering the Right Data at Runtime

The final stage brings the data strategy into the environment where agents do their work. *Chapter 11* covers how to decide what belongs in the context window, when to retrieve information, how context should change during a task, and what the agent should retain or discard. Context engineering brings together instructions, retrieved knowledge, memory, tools, and user input so the agent receives the right information at each step.

Each stage builds on the one before it. The data is collected in *Chapter 2*, prepared in *Chapter 3*, retrieved in *Chapters 4 and 5*, and expanded in *Chapter 6*. *Chapters 7 and 8 use that foundation for post-training and model customization. Chapters 9 and 10* measure, monitor, and improve the system in production. *Chapter 11* brings the same data discipline into the context window, where agents need the right information at the right moment. By the end, you will

have moved from a model that knows nothing about your domain to an assistant that can use your data accurately and remain reliable in production.

Summary

You have completed the setup. You can call the model. The more important lesson came from the in-context learning examples: the same model produced a different answer once we supplied the missing information.

The four assistants map to real problems I have seen. Productivity needs calendar and email data the model was never trained on. Domain needs current SEC filings that change quarterly. Enterprise needs internal documentation that nobody keeps current. Moderation needs policy consistency across edge cases that no training set covers. Different data, different retrieval, but the same basic lesson: the model can only work with the information the system gives it.

The `ask_ai` function is deliberately bare. You will use it for the rest of the book, and the point is never the function call. The in-context learning demo showed why. Same function, same model, completely different answer once you added the right context to the prompt.

The nine stages span all eleven chapters. Data collection feeds preparation, and prepared data supports retrieval. Synthetic data helps close the gaps, while post-training and customization change how the model behaves. *Chapter 9* then introduces evaluation, using representative datasets and measurable criteria to determine whether those changes actually improve the system. Chapters 10 and 11 take the system into production through data quality, human oversight, and context engineering. The bare model call you just wrote is only the starting point. Over the next 10 chapters, you will surround it with the data, retrieval, evaluation, and runtime context needed for a production AI assistant.

In this chapter, we connected to the model and generated a response. What you do not yet have is the information that would make those answers useful in your specific domain. That is *Chapter 2*, where you focus on data sources and collection strategies: identifying, accessing, and curating the raw materials that will power your assistant.

References

- [1] Cherry, D., Siddall, D., ElHousieny, R., Redondo Sanchez, P., & Ganapathy, P. (2024, December 13). *How Clearwater Analytics is revolutionizing investment management with generative AI and Amazon SageMaker JumpStart*. AWS Machine Learning Blog. https://aws.amazon.com/blogs/machine-learning/how-clearwater-analytics-is-revolutionizing-investment-management-with-generative-ai-and-amazon-sagemaker-jumpstart/
- [2] Sklaroff, J. (2024, September 23). *Constructing the Ultimate Gen AI Chat/Copilot Experience (Part 1)*. Clearwater Engineering Blog (Medium). https://medium.com/cwan-engineering/constructing-the-ultimate-gen-ai-chat-copilot-experience-part-1-55c2e07a1feb
- [3] ElHousieny, R. (2026, April 12). *The Claude Code Course: From Zero to AI-Powered Developer*. Level Up Coding (Medium). https://levelup.gitconnected.com/the-claude-code-course-from-zero-to-ai-powered-developer-fd2d1f86e923
- [4] ElHousieny, R. (2023, October 7). *Working with Google Colab for Python and Machine Learning: A Comprehensive Guide*. LinkedIn Pulse. https://www.linkedin.com/pulse/working-google-colab-python-machine-learning-guide-rany
- [5] ElHousieny, R. (2024, August 21). *AI Prompt Engineering and ReACT Framework. LinkedIn Pulse*. https://www.linkedin.com/pulse/ai-prompt-engineering-react-framework-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-ofhlc
- [6] ElHousieny, R. (2024, July 1). *In-context Learning with LangChain: Revolutionizing AI Interaction*. LinkedIn Pulse. https://www.linkedin.com/pulse/in-context-learning-langchain-revolutionizing-ai-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-r0oxc
- [7] OpenAI. *API Documentation. OpenAI Platform*. https://platform.openai.com/docs
- [8] OpenAI. *OpenAI Python SDK. GitHub* https://github.com/openai/openai-python
- [9] ElHousieny, R. (2023, August 6). *Python Basics Crash Course. LinkedIn Pulse*. https://www.linkedin.com/pulse/python-basics-crash-course-

quick-refresher-before-rany

- [10] ElHousieny, R. (2024, June 10). *Mastering LangChain with Examples: The Future of AI Development*. Level Up Coding (Medium). https://levelup.gitconnected.com/mastering-langchain-with-examples-the-future-of-ai-development-180bea2a6d27
- [11] ElHousieny, R. (2023, July 7). *Getting Started with the ChatGPT API: Getting API Key. LinkedIn Pulse*. https://www.linkedin.com/pulse/getting-started-chatgpt-api-key-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0
- [12] ElHousieny, R. (2023, July 8). *Unlocking Conversations: Your First ChatGPT API Call. LinkedIn Pulse*. https://www.linkedin.com/pulse/unlocking-conversations-your-first-chatgpt-api-call-rany
- [13] Amazon Web Services. *Generative AI Customer Stories. AWS Generative AI*. https://aws.amazon.com/ai/generative-ai/customers/
- [14] Siddall, D. (2024, August 20). *A Cutting-Edge Framework for Evaluating LLM Output. Clearwater Engineering Blog (Medium)*. https://medium.com/cwan-engineering/a-cutting-edge-framework-for-evaluating-llm-output-edab53373514
- [15] Cherry, D. (2024, May 24). *Customizing Generative AI Solutions with RAG*. Clearwater Engineering Blog (Medium). https://medium.com/cwan-engineering/customizing-generative-ai-solutions-with-retrieval-augmented-generation-86121761b8e3
- [16] ElHousieny, R. (2026, April 23). *What a Context-aware Knowledge Engine Is, and How to Build One Yourself*. LinkedIn. https://www.linkedin.com/pulse/what-context-aware-knowledge-engine-how-build-one-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-taaic

2

Data Sources

The effectiveness of any LLM application is defined not just by the model, but also by the data it learns from and interacts with. Assuming a fixed baseline model, **data sources, diversity, and quality are the dominant factors that influence LLM application quality**. Beyond initial LLM training, many product-level improvements come from improvements in training or inference-time data that target specific customer problems. High-quality data teaches the model how to behave in real use cases.

Data sources play a central role in LLM accuracy and relevance. This chapter examines the major data sources including web-scale data, domain-specific content, public datasets, licensed data, and other curated data sources. These sources serve different purposes. Training data teaches the model core knowledge. Retrieval data provides external knowledge at inference time. Evaluation data measures application quality, and you must isolate it from training data. Each source comes with distinct strengths and limitations, and the same source might be appropriate for one purpose but not another.

The goal of this chapter is to help you select, acquire and assess data sources based on the needs of your LLM application. Through comparisons of different data sources and practical acquisition examples, you will learn how to assess their relevance, quality, coverage, freshness, and operational constraints before consuming them. By the end of the chapter, you should have a clear mental model of how different data sources support training, retrieval, and evaluation. You should also understand how those choices influence LLM performance and reliability.

This chapter covers the following key topics:

- **Using web data** for topic coverage and fresh information

- **Acquiring domain-specific content** for specialized industries and fields
- **Processing HTML and PDF documents** to extract useful information
- **Leveraging public datasets** for LLM training and evaluation
- **Exploring licensed data** and its benefits, limitations, and usage considerations
- **Selecting relevant data sources** for different LLM applications

Technical requirements

This chapter builds on the shared development environment introduced in *Chapter 1*. If you have completed the setup in *Chapter 1*, no additional software installation is required. The examples in this chapter use the same Python environment and Jupyter notebooks introduced earlier. This chapter focuses on acquiring and working with different data sources and requires only a basic familiarity with Python and the LLM application concepts introduced in *Chapter 1*.

To follow the examples in this chapter, you should have:

- The development environment configured as described in *Technical Requirements* of this chapter.
- Access to the book's GitHub repository and the accompanying Jupyter notebooks
- Internet access for examples that retrieve data from public websites, APIs, and datasets

The practical examples in this chapter use libraries and APIs for accessing sources such as web content, public datasets, research repositories, and document formats. Some examples retrieve live data from external sources, so their exact outputs might change over time. The source code for this chapter is available in `Chapter_2_Notebook.ipynb` in the book's GitHub repository.

Using web data

Web data includes a wide range of online content, such as webpages, articles, social media posts, and forum discussions. This data is commonly used to educate LLMs about fundamental world knowledge and trending topics.

While most LLM systems draw on existing models that have already been trained on large-scale web data, collecting and incorporating web data is still valuable. Additional web data enables better understanding of specific topics and incorporates information from restricted webpages that are not publicly available.

Additionally, web data is critical for augmenting LLM knowledge with fresh information. It is usually used within **retrieval augmented generation (RAG)** pipelines, where external content supplements the LLM internal knowledge. In RAG, web data improves response accuracy, freshness, and grounding in authoritative sources. This integration is useful for many applications, such as question answering, text generation, and conversational AI systems.

In this section, you define the main types of web data and go over popular web data collections for LLM training and LLM applications. We also examine the challenges associated with collecting and preparing web data including noise, highly variable quality, and operational complexity. These challenges are common in real-world systems. You close the section with the best practices associated with using web data while building and operating LLM-based systems.

Types of web data

This section defines the key types of web data that are commonly used for training LLMs. Getting familiar with these web data types will help you with proper collection and usage of web data for different LLM applications. The following is a list of the most important web data types:

- **Webpages** are the most common form of web data. They include news articles, blogs, and other websites covering topics, such as science, technology, news, and culture. These sources provide broad text data for building general knowledge and language understanding.
- **Social media** provides comprehensive and diverse information. Platforms, such as Facebook, X (formerly Twitter), and LinkedIn offer a wealth of user-generated content that can be used to train LLMs on natural language conversations, sentiment analysis, and trend detection.

- **Forums and discussion boards, such as** Reddit and Quora, provide deep, community-driven content on various general topics. Specialized forums, such as IGN boards and XDA forums contain specialized knowledge on niche topics. This content is useful for building domain-specific LLM applications.
- **Online reviews** include product reviews, movie reviews, and other online reviews. They are suitable for training LLMs in shopping and opinion mining applications.

Now that you have explored the various types of web data, you can delve into specific examples of web data collections that are commonly used in LLM training and applications across the industry and open-source communities.

Example web data collections

In this section, you can find some specific web data collections that can be easily used for LLM training and inference. You can examine them one by one.

Common Crawl

Common Crawl is a non-profit organization that provides a large corpus of webpages. This dataset is useful for training language models, as it contains vast text data from various websites and webpages.

You can collect news articles from Common Crawl as a use case example for your personal assistant. Common Crawl is particularly useful for building personal assistants that require awareness of public news and trends.

For instance, you can use `requests` to retrieve URLs from the Common Crawl index, then download and parse the raw webpages for relevant content (for example, topics such as "artificial intelligence" and "machine learning").

Here is an example piece of code that shows exactly that. You can find the full code of this example and all other examples in the book's GitHub repository under `Chapter_2_Notebook.ipynb`. The output of this code is a corpus of archived news articles from the public web:

```
import requests, json
from bs4 import BeautifulSoup
from datetime import datetime

class CommonCrawlNewsCollector:
    def __init__(self):
```

```
        self.cc_index_server = "https://index.commoncrawl.org"
        self.news_domains = ['techcrunch.com',
                             'theverge.com', 'wired.com']
        self.ai_keywords = ['artificial intelligence',
                            'machine learning', 'gpt', 'llm']
        self.articles = []

    def search_news_urls(
       self, domain, index_name="CC-MAIN-2024-10", limit=100
    ):
        r = self.session.get(
            f"{self.cc_index_server}/{index_name}-index",
            params=params, timeout=60
        )
        if r.status_code == 200:
            return [json.loads(line)
            for line in r.text.splitlines() if line.strip()]
        return []

    def is_likely_ai_news_article(self, url, title="", content=""):
        ...  # URL patterns + AI keyword scoring; return (is_ai, score)

    def collect_ai_news_articles(
        self, domains=None, index_name="CC-MAIN-2024-10", max_per_domain=5
    ):
        for domain in (domains or self.news_domains):
            results = self.search_news_urls(
                domain, index_name, 300)
            for meta in results:
                ok, score = self.is_likely_ai_news_article(
                    meta.get("url", ""))
                if not ok:
                    continue
                ... # GET Common Crawl WARC bytes, check status 206
                ... # parse HTML with BeautifulSoup, extract title + <p> text
                ... # apply word_count/ai_score thresholds
                self.articles.append({
                    "domain": domain,
                    "url": meta.get("url", ""),
                    "timestamp": meta.get("timestamp", ""),
                    "title": title,
                    "content": content,
                    "word_count": word_count,
                    "ai_score": score,
                    "collected_at": datetime.now().isoformat()
                })
        self.articles.sort(key=lambda x: x.get("ai_score", 0),
                           reverse=True)
        return self.articles
```

```
# Chapter_2_Notebook.ipynb: Collecting News Articles from Common Crawl
```

Wikipedia

Wikipedia is a free online encyclopedia that contains millions of articles on various topics. It is a high-quality asset for training language models, providing vast, high-quality text data on multiple subjects.

Here is a use case example for your domain-specific assistant that extracts medical text from Wikipedia. Wikipedia contains high-quality articles on technical domains such as medicine, law, and finance. This example shows how to use the `wikipedia-api` package to retrieve an article related to cardiology.

Please note the `cardiology` keyword in the example code. The output from this code is a title and summary preview suitable for fine-tuning or indexing for retrieval:

```
import wikipediaapi

USER_AGENT = "MyColabBot/1.0 (https://example.com/contact) Python/3.x"

wiki = wikipediaapi.Wikipedia(
    user_agent=USER_AGENT,
    language='en'
)

page = wiki.page("Cardiology")

if page.exists():
    print("Title:", page.title)
    print("Summary:", page.summary[:500])
else:
    print("Page not found")

# Chapter_2_Notebook.ipynb: Extracting Medical Text from Wikipedia
```

A comment at the bottom of each code snippet always points to the corresponding full code in the chapter's notebook. You can use these pointers to try the complete code examples yourself.

Project Gutenberg

Project Gutenberg is a digital library that contains over 60,000 free e-books. These books are mostly classics and are available in various formats, including

text files. This dataset is useful for training language models on literary styles, genres, and language structures.

Here is a code example for your content moderation use case. It identifies obsolete or biased content in classic literature. Project Gutenberg's classic books can help train moderation systems to detect outdated, biased, or sensitive content. Some older literature includes culturally insensitive language or themes that would be inappropriate in modern contexts.

This example filters such texts for moderation analysis. The output from this code is a preview of lines containing potentially sensitive terms for contextual review:

```
import requests

biased_terms = ['savage', 'negro', 'oriental']

def flag_sensitive_lines(file_path):
    flagged = []
    with open(file_path, encoding='utf-8') as f:
        for line in f:
            if any(term in line.lower() for term in biased_terms):
                flagged.append(line.strip())
    return flagged

URL = "https://www.gutenberg.org/files/1342/1342-0.txt"
FILE_PATH = "gutenberg_sample.txt"

resp = requests.get(URL, timeout=60)
if resp.status_code == 200:
    with open(FILE_PATH, "wb") as f:
        f.write(resp.content)
else:
    ...  # handle download errors or retry logic

sensitive = flag_sensitive_lines(FILE_PATH)
...  # optionally dedupe/sort or write flagged lines to a report

for preview in sensitive[:5]:
    print(preview)

# Chapter_2_Notebook.ipynb: Identifying Obsolete or Biased Content in Classic Literature
```

OpenWebText

OpenWebText is a dataset of webpages scraped from the internet. It is similar to Common Crawl but has some data collection and processing differences.

This dataset is useful for training language models on various topics and styles.

OpenWebText is particularly useful for building personal assistants that require awareness of general web content and trends, without relying on a live web crawl. For instance, you can use the datasets library from Hugging Face to stream and filter documents from the `OpenWebText` corpus based on user-specified topics (for example, "climate change" or "tech news"). This approach avoids the need for URL lookups and HTML parsing, as the dataset already contains cleaned text extracted from webpages.

Here is a piece of code that extracts text snippets from OpenWebText given a selected list of topics of interest, such as "climate change." The output is a corpus of relevant text snippets sourced from the OpenWebText dataset:

```
from datasets import load_dataset
import csv, re

CONFIG = {"TOPICS": ["climate change", "tech news"],
          "MAX": 200, "SCAN": 120_000, "CHARS": 420,
          "CI": True, "WW": False}

def topic_to_regex(term, whole_word=False):
    ...  # compile regex using CONFIG["CI"] and optional word boundaries

def make_snippet(text, span, width=CONFIG["CHARS"]):
    ...  # compute centered snippet and return (snippet, start, end)

def done(seen):
    ...  # return True when all topics reach CONFIG["MAX"] or seen >= CONFIG["SCAN"]

pats = {t: topic_to_regex(t, CONFIG["WW"]) for t in CONFIG["TOPICS"]}
ds = load_dataset("Skylion007/openwebtext", split="train", streaming=True)
writer = csv.writer(open("openwebtext_snippets.csv", "w",
                    newline="", encoding="utf-8"))
...  # write CSV header row:
     # ["topic","snippet","doc_id","char_start","char_end"]

seen = 0
for ex in ds:
    if done(seen):
        break
    seen += 1
    txt = ex.get("text") or ""
    for topic, pat in pats.items():
        ...  # skip topic if already at CONFIG["MAX"]
        m = pat.search(txt)
```

```
        if not m:
            continue
        snip, a, b = make_snippet(txt, m.span())
        writer.writerow([topic, snip, seen, a, b])
...  # optionally print progress and final summary
     # (counts per topic, output path)

# Chapter_2_Notebook.ipynb: Collecting News Articles from OpenWebText
```

BookCorpus

BookCorpus is a dataset of books and texts that can be used for training language models. The dataset contains an extensive collection of books, including fiction and non-fiction works. This dataset is useful for training language models on literary styles, genres, and language structures.

Here is a use case example for your enterprise assistant that builds a BookCorpus-like dataset from internal reports. You can create BookCorpus-inspired datasets from internal corporate documentation, such as handbooks, HR policies, and internal technical manuals.

This approach is especially valuable for building enterprise assistants that answer internal queries. The input to this code is a collection of markdown or text-based internal documents. The output is an internal corpus for indexing or fine-tuning for question-answering tasks:

```
import os
import glob

DOCS_DIR = "../data/internal_docs"

def collect_internal_docs(directory):
    """Return list of full-text contents of *.md files in directory."""
    corpus = []
    for file_path in glob.glob(os.path.join(directory, "*.md")):
        with open(file_path, encoding="utf-8") as f:
            text = f.read()
            ...  # optional: normalize whitespace / remove YAML front matter
            corpus.append(text)
    return corpus

if __name__ == "__main__":
    docs = collect_internal_docs(DOCS_DIR)
    if docs:
        print(f"Found {len(docs)} markdown files in {DOCS_DIR}.")
        print(docs[0][:500])  # preview first 500 chars of first doc
        ...  # optionally: write combined corpus to a file
```

```
        ...  # optionally: build train/val split or tokenize
    else:
        print(f"No markdown files found in {DOCS_DIR}.")
        ...  # optionally: prompt user to upload or set a different directory

# Chapter_2_Notebook.ipynb: Internal BookCorpus-like Dataset from Internal Reports
```

Each of these **data collections** has clear strengths and weaknesses. They can be used independently or combined to support different AI applications. At the same time, it is important to recognize that these collections also introduce non-trivial challenges.

In the following sections, you examine these challenges and outline best practices for addressing them when building **LLM-based systems**.

Challenges in collecting web data

Collecting web data introduces a set of practical challenges. These challenges surface quickly once LLM systems move from small experimentation to real-world production and use:

- The first issue is **data quality and relevance**. Web data is noisy by nature. The web contains duplicate and low-quality content with high frequency of grammar and spelling mistakes. Additionally, most of the web content would have little relevance to your target application. Without targeted filtering, this noise will affect the model behavior and downstream application performance negatively. Correct preparation of web data requires intentional pre-processing with cleaning, normalization, and filtration steps. These steps are essential to overcome the challenges. The next chapter continues the data preparation process by covering chunking, metadata enrichment, vectorization, and indexing.
- The second challenge is the **data volume and scale**. The web contains huge amounts of content, but such scale comes at a cost. Collecting, processing, and storing large volumes of data require a solid infrastructure that can scale without becoming slow or too expensive. During LLM training, **more training data translates into longer training cycles**. To keep early model experiments fast, work with smaller datasets and delay full-scale training to release-candidate model snapshots.

- The third challenge is **imbalance of topics and perspectives**. While the web content is comprehensive, topic and perspective representations are not uniform. Certain topics and opinions show disproportionate representation while others are sparse. These imbalances can surface in the form of biased outputs in your LLM application.
- The fourth issue is **data accessibility**. Many websites implement **anti-scraping techniques,** restrict **their terms of service** and add **copyright limitations**. To be compliant with these constraints, your crawler must be aware of the different licenses and follow polite crawling practices, especially when building commercial systems.
- Finally, **web data is highly dynamic**. Webpages are updated all the time. The constant updates make it difficult to keep web datasets consistent and duplicate-free and complicate the separation of training and evaluation data.

Awareness of these challenges and related solutions helps you collect and use web data more safely and effectively.

Best practices for collecting web data

At this stage, you explored the challenges associated with web data collection. You cannot solve these challenges with a single technique. They require a set of techniques that work together to control data sourcing, processing and maintenance over time.

The starting point is to **define a clear scope for data collection**. Narrow your collection to a specific set of sources and topics. Narrowing the collection will help limit the noise and reduce the risk of collecting large volumes of low-value data. Prefer a defined scope over unplanned wide coverage, especially at an early model development stage when scope is not discussed before.

Next step is **source selection**. Rely on reputable and well-understood websites to improve baseline data quality and reduce downstream filtering costs. Source selection is even more critical when web data is used directly in RAG pipelines, where low-quality content or wrong information can easily show up in final model outputs.

Next is **selecting the right data collection tools**. There are many techniques and solutions for collecting web data, from public datasets to crawling

libraries to building your own crawlers. The right tool depends on many factors such as target scale, budget and data refresh frequency.

On another dimension, **legal and operational constraints** must be treated as first-class concerns. Respecting website **terms of service**, **robots.txt** directives, and **copyright limitations** is not optional. These constraints in many cases define the validity of the data sources for your application way before technical considerations.

If the solutions discussed so far seem to be too complex, **curated datasets are much simpler than large-scale crawling** for training and evaluation purposes. When a curated data set matches your application's purpose and domain, it saves a lot of time and avoids the hassles associated with web crawling. You should keep crawling for live retrieval during inference time.

The risk with curated datasets is whether they are a good fit for your application. Curated datasets do not exactly match a domain or target behavior and usually miss the coverage of edge cases that matter for your application. When that is the case, the model might look strong in offline evaluation but fails on long tail cases that are difficult to represent in evaluation data.

After data collection, **filtering and normalization** are key techniques that remove duplicate, irrelevant and poor-quality content. This directly addresses the quality and relevance challenges discussed earlier. These two steps ensure that more data amplifies useful signals rather than accumulating noise.

Finally, **web data collection systems must account for change**. Web content evolves quickly with continuous feeds of user-generated content and website restructuring activities. Effective data collection pipelines include monitoring and adaptation mechanisms, along with storage and management strategies such as versioning and hashing that support reprocessing and content refresh cycles when needed.

Taken together, these practices transform web data from an unreliable raw input into a usable and maintainable cornerstone of LLM-based systems. However, broad web coverage is not sufficient when an application depends on specialized terminology or industry-specific knowledge. In these cases, web data needs to be complemented with domain-specific content.

Acquiring domain-specific content

Domain-specific content is important for building LLM-based business applications. Generic web data teaches the model core language understanding and reasoning. Domain data teaches the model the vocabulary and rules of real work in a specific industry or organization.

Domain content comes from many sources, including technical manuals, internal documents, industry reports, research papers, and regulatory material. Each source contributes different information. Some sources provide stable facts. Others capture processes and operational corner cases. When done right, the sources work together to improve relevance and precision of model responses.

Internal documentation is probably the most challenging source to use effectively. Internal documentation contains a mix of outdated and recent materials. If you ingest everything that is available, you introduce contradictions. **Filtration is essential** to remove noise and ensure factual correctness.

Expert-generated material is complementary in some business domains. This material helps resolve ambiguity in internal documentation and provides targeted clarification where sources conflict or drift over time. Expert-generated material can also introduce new knowledge in a controlled way without inheriting the full historical volume of other sources.

In the following sections, you can break **domain-specific content** into categories and examine how each category influences model behavior.

Types of domain-specific content

Domain-specific content spans a broad set of materials tailored to the needs and constraints of a particular industry, function or organization. This content is useful for training LLMs to understand and respond accurately to domain queries. Compared to general-purpose data, it provides highly focused information, improves precision, and enables models to operate closer to real user intents.

Different types of domain-specific content contribute different information. Each type of content shapes model behavior in a different way, and each

comes with its own tradeoffs around freshness, reliability and coverage. You can examine some of them here:

- **Internal documents** such as reports, memos, specifications and policy documents encode how an organization operates. These sources describe business processes, internal terminology, regulatory interpretations, and decision constraints that rarely appear in public data. When curated and filtered correctly, *internal documentation* grounds models in organization-specific reality. When used naively, it can introduce contradictions and outdated assumptions.
- **Emails and internal communication** provide contextual signals rather than canonical truth. They cover how people discuss issues, escalate problems, and negotiate decisions in practice. This content helps models learn tone, informal language and abbreviations. At the same time, it requires special handling, as it often mixes speculation, incomplete information, and transient states.
- **Meeting notes and transcripts** capture decision-making in motion state. They surface tradeoffs and rationale that might never be formalized in other sources. This content is valuable for reasoning over decision-making process beyond just the final decisions. However, meetings are noisy and time-bound, which limits its usefulness without strong contextual filtering.
- **Wikis and internal knowledge bases** represent curated organizational memory. They consolidate best practices, FAQs, architectural guidance, and operational playbooks. Because they support reuse, they tend to be provide more structure and stability than other internal sources. When kept current, they become a high-quality backbone for **domain grounding**.

In practice, no single content type is sufficient on its own. **Effective domain grounding comes from combining multiple sources** and applying different curation and validation strategies depending on an understanding of the role of each source.

Medical domain data

Medical domain data encompasses a wide range of critical information sources for research, clinical practice, and education in the healthcare field.

These data sources include academic articles, clinical notes, **electronic health records** (**EHRs**), and reference materials. Each type of data provides unique insights and plays a key role in teaching your LLM the necessary medical knowledge for better model performance. The following is a list of key sources of medical domain data:

- **PubMed** articles and biomedical literature
- Clinical notes and EHRs
- Medical textbooks and reference materials

This example code fetches medical articles from PubMed given a search query:

```
from typing import List, Dict, Optional
from Bio import Entrez, Medline

def fetch_pubmed_abstracts_json(
    query: str, max_results: int = 5,
    email: str = "your_email@example.com",
    api_key: Optional[str] = None,
    sort: str = "most+recent"
) -> List[Dict]:
    """Search PubMed and return results as structured JSON."""
    Entrez.email = email
    if api_key:
        Entrez.api_key = api_key

    with Entrez.esearch(
        db="pubmed", term=query, retmax=max_results, sort=sort
    ) as h:
        ids = Entrez.read(h).get("IdList", [])
    if not ids:
        return []

    with Entrez.efetch(
        db="pubmed", id=",".join(ids), rettype="medline", retmode="text"
    ) as h:
        records = list(Medline.parse(h))

    results = []
    for rec in records:
        ...  # extract pmid, title, abstract, authors, journal, year
        ...  # extract DOI from AID or LID fields if available
        results.append({...})  # append structured dict
    return results

if __name__ == "__main__":
    data = fetch_pubmed_abstracts_json(
        "cardiology heart disease", max_results=3,
```

```
        email="your_real_email@domain.com"
    )
    import json
    print(json.dumps(data, indent=2, ensure_ascii=False))

# Chapter_2_Notebook.ipynb: Acquiring Medical Domain Data
```

Here is what the output data looks like in JSON format:


```
[
  {
    "pubmed_id": "40802773",
    "title": "Transcriptome fingerprinting of aberrant fibroblast activation…",
    "abstract": "Aberrant activation of fibroblasts is a pivotal component of…",
    "authors": [
      "Cinato M",
      …
    ],
    "journal": "Science advances",
    "year": "2025",
    "doi": "10.1126/sciadv.adx0968"
  },
…
```


Financial domain data

Financial domain data encompasses a variety of information sources for training and fine-tuning LLMs and their applications in the financial sector. These data sources include news articles, market data, **Securities and Exchange Commission** (SEC) filings, annual reports, and financial databases. Each type of data provides unique insights and plays a different role in enhancing the accuracy, relevance, and performance of LLMs in understanding and generating financial content. The following is a list of key sources of financial domain data:

- Financial news articles and market data
- **Securities and Exchange Commission** filings and annual reports
- Financial databases and analytics platforms

This example code retrieves SEC company facts of Apple Inc. from the U.S. Securities and Exchange Commission Data APIs:

```
import pandas as pd
import json, datetime as dt
import requests
```

```
TICKER = "AAPL"
START = "2024-01-01"
END = dt.date.today().isoformat()
CIK = "0000320193"
USER_AGENT = "ColabDemo/1.0 (youremail@example.com)"

def get_stock_ohlcv_json(ticker: str, start: str, end: str) -> list[dict]:
    ...  # Convert start/end to start_ts/end_ts
    response = requests.get(
        f"https://query1.finance.yahoo.com/v8/finance/chart/{ticker}",
        params={"period1": start_ts, "period2": end_ts, "interval": "1d"}
    )

    ...  # sort ascending, rename OHLCV columns, format date
    return df.to_dict(orient="records")

def get_sec_company_facts(cik: str, user_agent: str) -> dict:
    url = f"https://data.sec.gov/api/xbrl/companyfacts/CIK{cik}.json"
    r = requests.get(url, headers={"User-Agent": user_agent}, timeout=30)
    r.raise_for_status()
    return r.json()

ohlcv = get_stock_ohlcv_json(TICKER, START, END)
print(json.dumps(ohlcv[:5], indent=2))

try:
    facts = get_sec_company_facts(CIK, USER_AGENT)
    ...  # select USD metric from facts["facts"]["us-gaap"],
         # sort by end/instant, keep recent slice
    ...  # print compact JSON with metric name and observations
except requests.HTTPError:
    ...  # SEC policy requires a valid User-Agent with email

# Chapter_2_Notebook.ipynb: Acquiring Financial Domain Data
```

Here is what the output data looks like in JSON format:

```
# Example JSON: SEC Company Facts (compact slice) for CIK 0000320193
{
  "metric": "NetIncomeLoss",
  "observations": [
    {
      "fy": 2025,
      "fp": "Q1",
      "form": "10-Q",
      "start": "2024-09-29",
      "end": "2024-12-28",
      "value": 36330000000
    },
…
```

Legal domain data

Legal domain data includes a diversity of information sources that are vital for developing and fine-tuning LLMs and their applications in the legal field. These sources encompass court opinions and legal briefs, statutes and regulations, law review articles, and legal commentary. Each type of data offers unique perspectives in improving the precision, relevance, and functionality of LLMs in interpreting and generating legal content.

The following is a list of key sources of legal domain data:

- Court opinions and legal briefs
- Statutes and regulations
- Law review articles and legal commentary

This example code acquires legal snippets from the U.S. Supreme Court's public slip-opinion index.

```
def supreme_court_opinions(limit=5):
    term = date.today().year - (date.today().month < 10) - 1
    index_url = (
        f"https://www.supremecourt.gov/opinions/"
        f"slipopinion/{term % 100:02d}"
    )
    index = BeautifulSoup(
        requests.get(index_url, timeout=30).content,
        "html.parser"
    )
    records = []
    for link in index.find_all("a", href=True):
        ...  # Select opinion PDFs;
             # read case name, release date, and docket number.
        pdf_url = urljoin(index_url, link["href"])
        pdf = requests.get(pdf_url, timeout=30)
        pdf.raise_for_status()
        with pymupdf.open(stream=pdf.content, filetype="pdf") as document:
            excerpt = (" ".join(page.get_text()
                       for page in list(document)[:2]))

        records.append({
            "source": "us_supreme_court_slip_opinions",
            "case_name": case_name,
            "date_released": release_date,
            "docket_number": docket_number,
            "snippet": " ".join(excerpt.split())[:400],
            "url": pdf_url,
        })
```

```
        if len(records) == limit:
            break
    return records

records = supreme_court_opinions()
... # Write one record per line to ../data/legal_data_sample.jsonl

# Chapter_2_Notebook.ipynb: Acquiring Legal Domain Data
```

Here is what the output data looks like in JSON format:

```
{
  "source": "us_supreme_court_slip_opinions",
  "case_name": "People Not Politicians v. Onder",
  "court": "Supreme Court of the United States",
  "date_released": "2026-09-25",
  "docket_number": "26A388",
  "citations": [],
  "snippet": "Cite as: 609 U. S. ____ (2026) 1 Per Curiam ...",
  "url": "https://www.supremecourt.gov/opinions/25pdf/26a388_q86b.pdf"
}
```

Scientific research data

Scientific research data comprises a broad spectrum of information sources that are important for the scientific domain. These sources include research articles and academic papers, scientific databases and datasets, as well as conference proceedings and abstracts.

The following is a list of key sources of scientific research data:

- Research articles and academic papers
- Scientific databases and datasets
- Conference proceedings and abstracts

This code snippet fetches scientific abstracts from Crossref database using a search query.

```
import json, requests
from datetime import datetime

SEARCH_QUERY = "large language models safety"
FROM_DATE = "2024-01-01"
TO_DATE = datetime.utcnow().date().isoformat()
USER_AGENT = "QES-Research-Colab/1.0 (mailto:hassan@example.com)"

def fetch_crossref(query, from_date, to_date, rows=15):
    r = requests.get(
```

```
        "https://api.crossref.org/works",
        params={
            "query": query,
            "filter": f"from-pub-date:{from_date},"
                      f"until-pub-date:{to_date}",
            "rows": rows, "sort": "published", "order": "desc"
        },
        headers={"User-Agent": USER_AGENT}, timeout=30
    )
    ...  # parse r.json()["message"]["items"]; build list of dicts (title, authors, published, doi,
url, venue, abstract)
    return [...]  # normalized results

def fetch_arxiv(query, max_results=15):
    ...  # GET http://export.arxiv.org/api/query; parse Atom XML;
         # collect title/authors/published/url/summary

def main():
    items = []
    try:
        items.extend(fetch_crossref(
            SEARCH_QUERY, FROM_DATE, TO_DATE, rows=15))
    except Exception:
        ...  # handle Crossref errors
    ...  # optionally: fetch_arxiv(SEARCH_QUERY, max_results=15)
    ...  # and extend dedupe by (title, source), sort by published desc
    ...  # save to /data/science_results.json and pretty-print first 3 items

if __name__ == "__main__":
    main()

# Chapter_2_Notebook.ipynb: Acquiring Scientific Research Data
```

Here is what the output data looks like in JSON format:


```
  {
    "source": "Crossref",
    "title": "A New Way of Storing Traffic Camera Footage at Scale",
    "authors": [
      "Hoyoung Kim",
      …
    ],
    "published": "2025-08-31",
    "doi": "10.1145/3744649",
    "publisher": "Association for Computing Machinery (ACM)",
    "venue": "ACM Transactions on Multimedia Computing, Communications, and…
    "type": "journal-article",
    "url": "https://doi.org/10.1145/3744649",
    "abstract": "Storing large volumes of traffic video content in cloud storage is an expensive
```

```
undertaking, given the limited capacity of cloud storage and its…
  },
```

Software development data

Software engineering is a specific domain, and thus software development data has its own characteristics. This data includes repositories from platforms, such as GitHub, which provide access to open-source code, programming documentation and tutorials. Additionally, code review comments and feedback add insights into best practices and common pitfalls. Each type of data helps in understanding and generating software development content.

The following is a list of key sources of software development data:

- GitHub repositories and open-source code
- Programming documentation and tutorials
- Code review comments and feedback

This code snippet retrieves software development content from GitHub, PyPI and Stack Overflow.

```
import os, json, time, datetime as dt
import requests

GITHUB_TOPIC = "fastapi"
GITHUB_LANGUAGE = "Python"
REPO_LIMIT = 3
PYPI_PACKAGES = ["fastapi", "pydantic", "uvicorn"]
STACK_TAG = "fastapi"
STACK_LIMIT = 5
UA = "ColabSoftwareDataDemo/1.0 (+contact: your-email@example.com)"

def fetch_github_repos(topic, language, limit=3):
    r = requests.get(
        "https://api.github.com/search/repositories",
        params={
            "q": f"topic:{topic} language:{language}",
            "sort": "stars", "order": "desc", "per_page": limit
        },
        headers={"User-Agent": UA}, timeout=30)
    ...  # parse r.json()["items"]; fetch /languages per repo;
         # build dicts with name/url/stars/topics/languages_breakdown
    return [...]

def fetch_pypi_packages(packages):
    ...  # for each pkg: GET https://pypi.org/pypi/{pkg}/json;
```

```
            # extract latest_version, summary, license, uploaded_at
    return [...]

def fetch_stackoverflow_questions(tag, limit=5):
    r = requests.get(
        "https://api.stackexchange.com/2.3/questions",
        params={
            "order": "desc", "sort": "activity", "tagged": tag,
            "site": "stackoverflow", "pagesize": limit
        },
        headers={"User-Agent": UA}, timeout=30)
    ...  # normalize items to :
         # question_id/title/link/creation_date/score/is_answered/tags
    return [...]

github = fetch_github_repos(GITHUB_TOPIC, GITHUB_LANGUAGE, REPO_LIMIT)
result = {"sources": {"github": github}}
print(json.dumps(result, indent=2, ensure_ascii=False))
...  # save to /data/software_dev_data.json

# Chapter_2_Notebook.ipynb: Acquiring Software Development Data
```

Here is what the output data looks like in JSON format:


```
"stack_overflow": [
  {
    "question_id": 79734898,
    "title": "FastAPI OAuth2PasswordBearer JWT authorization not working in Swagger…
    "link": "https://stackoverflow.com/questions/79734898/fastapi-oauth2password…
    "creation_date": "2025-08-14T02:26:51Z",
    "score": 0,
    "is_answered": false,
    "tags": [
      "python",
      "jwt",
      "fastapi"
    ]
  },
…
```


Across these examples, the data sources vary by industry. The common underlying pattern is that domain-specific sources provide the specialized terminology and knowledge that general-purpose data does not fully cover. This translates into several benefits that apply across specialized LLM applications.

Benefits of domain-specific content

Domain-specific content offers several advantages for business applications. Across the different data types covered in this chapter, domain data ties directly to unique product value because it reflects real workflows, business constraints, and user intents within the product environment.

Relevance is the primary benefit. Domain content covers the decisions business workers make and the tasks they are trying to complete. Domain content reduces the need for analysis and guessing, and keeps the model anchored to the use cases that matter. **High relevance is the main differentiator of domain data** over broader web data in most enterprise applications.

Context relates to but not the same as *relevance*, especially in the context of *terminology*. Domain content tells the **meaning of terms, words and acronyms in this organization**, how concepts relate to each other, and the boundaries between them. Additionally, it carries implicit business rules that may not show up in authoritative documents. These hidden rules are imbedded in conversations, forms and exceptions for edge cases. Without that grounding, the model may produce fluent responses that do not reflect what the business wants.

On the other side of that, **accuracy of authoritative domain content protects your application** from over-reliance on hidden assumptions and stale information, especially when this content is regularly curated and maintained. In sensitive domains, small inaccuracies can compound into bad recommendations and incorrect actions. It is important to include both **authoritative content for accuracy and day-to-day content for context**.

These benefits highlight why domain-specific content is a must-have addition for business applications. At the same time, realizing the full value of domain content requires well -established processes and cadence that govern data collection and refresh. The next section covers the challenges and considerations that are associated with use of domain-specific content on a large scale.

Challenges and considerations

Domain content is high value but can introduce unintentional quality and privacy failures. LLM development teams usually optimize for *coverage* under

the assumption that more data leads to better models. In domain content, this instinct frequently backfires with potential failure modes such as conflicting answers originating from different information sources, outdated responses, and privacy failures such as exposing confidential employee information to an unauthorized user.

These failures do not come from the model structure. They emerge from the processes that work with domain data. In practice, the most critical challenges show up in two areas: **sensitive data protection**, and **integration of information sources**.

Domain content includes **sensitive and confidential information**. As coverage expands, the risk of surfacing correct information to unauthorized users increases. Without access controls in place, additional coverage will quietly increase the risk. **Security and authorization** checks must remain enforced end to end through the steps of data ingestion and retrieval.

Domain information does not live in a single clean data source. It spans documents, system records, and communications with different levels of **authority and freshness**. Integration of these information sources is not a simple ingestion problem. It is the problem of solving contradictions and improving consistency. If contradictions are not resolved, the model learns that all information sources are equal and answers based on whichever source is retrieved first. The integration problem is not about compatibility of formats between sources. It is **about achieving consistency by defining and ranking sources of truth** and preventing outdated and low-authority content from dominating responses.

Addressing these challenges is essential to preserving the benefits of domain-specific content without falling into traps. The next section focuses on best practices for managing domain data in a way that favors relevance over coverage.

Best practices for using domain-specific content

Effective use of domain-specific content is about correctness and freshness of high-value data, rather than accumulation of large volumes of data. The goal

is to maximize the alignment between the data that the LLM sees and real business operations.

The first practice is to **prioritize relevance over coverage**. Select domain data based on its representativeness of real workflows and decisions. Broad ingestion that prioritizes completeness tends to reintroduce the same contradictions and ambiguities that exist in other data sources. It is better to start with narrow and highly relevant scope and validate end-to-end impact, then expand gradually. A narrow, validated scope is much better than trying to ingest everything initially and correct later.

Related to relevance, you need to **define and rank sources of truth**. Not all domain content carries the same authority level. Policies, structured records and official documentation are more authoritative than meeting notes and historical conversations. Both are important to include in your data, but with explicit precedence rules that filter contradictions and override outdated information. Otherwise, the model will answer based on whatever data that happens to be retrieved at the time. Making authority explicit is a key data design decision for training and retrieval.

Next, necessary practice is **continuous curation**. Based on your business domain, content could become outdated quickly, especially in new and growing industries where processes and terminology evolve. Domain data is not static, and if it remains static for your application, it will gradually drift from being relevant to being useless. Continuous review and refresh cycles keep the system close to reality as of today, not to what it was when the data entered the system.

Finally, **evaluate domain content in production context** and recurrently, not just once and offline. The real value of domain data shows up in system behavior in high-value scenarios that contain edge cases and ambiguous requests. Feedback from these scenarios should feed back into data selection and refinement improvements. This feedback loop keeps the data relevant to current application requirements and preserves its value over time.

These practices together ensure that domain content contributes unique value while avoiding the potential failures associated with it. The next section concludes the discussion on domain content with an overview of its modern applications.

Applications of domain-specific content

Domain-specific content is used to train and evaluate LLMs across several scenarios. The common theme between them is specialization in which the model does not have to perform well at everything but is expected to be good at *specific and valuable scenarios*. This specialization can target an entire industry, a specific knowledge area, or the terminology and processes of an individual organization.

Recently, many AI startups were built around **industry-specific models**. This approach focuses on a single vertical and builds depth in its terminology and frequent tasks. This approach offers a direct path to product value because relevance is easier to achieve when the domain boundary is clearly defined. Common examples include **healthcare use cases,** such as medical diagnosis support and treatment planning, **financial applications** focused on analysis and forecasting, and **legal workflows** involving research and document drafting.

A deeper class of applications target **specialized knowledge areas** rather than an entire industry, driven by technical correctness and precision. **Scientific research, software engineering and security analysis** fall into this category.

Other teams invest in the application layer using **organization-specific fine-tuning**. Industry-specific models provide broad relevance across a vertical, but they are not sufficient on their own. Organization-specific customization adapts the model to a specific business by incorporating internal terminology and procedural conventions. Under the umbrella of organization-specific fine-tuning, there are additional techniques, such as *retrieval grounding* and *terminology management* that add incremental value. Industry-specific models and organization-specific fine-tuning enable LLMs to be more targeted and robust in real business deployments.

HTML and PDF documents are common on both the web and domain content. These formats are rich sources of information, but their structure introduces additional complexity that requires specialized processing to extract the target information. The next section examines this processing pipeline in more detail.

Processing HTML and PDF documents

HTML and PDF documents are popular and rich data sources with complex structures that need special processing. No production-grade system can skip the processing of HTML or PDF documents due to their dominant presence on most data sources. Your LLM system needs to effectively extract information from these documents to understand and generate natural responses.

The goal of this section is to understand the commonly used techniques for processing these documents. You will learn about the challenges and parsing methods and gain insights into the best practices for handling these formats.

HTML documents include nested elements and dynamic content. On the other hand, PDF parsing faces other challenges such as raw text extraction and identification of embedded images and tables. Various techniques are used to extract data from HTMLs and PDFs such as specialized libraries and tools, regular expressions, XPath queries, and OCR. You can list the challenges of HTML parsing and PDF parsing separately.

Challenges of parsing HTML documents

Here is a brief list of the key challenges associated with HTML document parsing during data collection. HTML documents use **highly variable structures** that make it difficult to identify relevant content structure such as headings, bullets and tables. Websites may use creative HTML tags, CSS styles, and scripts for formatting their pages. This variability requires flexible and adaptable parsing techniques.

Most HTML documents contain **noise and irrelevant data**, such as advertisements and navigation menus. Removing this noise is essential to extract high-quality data that can be used for training and evaluating LLMs.

Most HTML documents may contain **dynamic content** such as JavaScript-generated content, which is difficult to extract using traditional HTML parsing techniques. On top of this, **HTML documents vary in formatting styles**, such as different font sizes, colors, and layouts, which affects the accuracy of relevant data extraction. These challenges are driven by the flexible structure of webpages. PDFs introduce a different set of problems, especially around page layout, text extraction, and scanned content.

Challenges of parsing PDF documents

On the PDF document side, there are different challenges that are unique to PDF parsing. PDFs may have **complex layouts and structures**, which complicates raw text extraction and relationship understanding. Additionally, PDFs may contain multiple columns, tables, images, and other elements that need special parsing techniques.

Scanned PDFs require OCR to extract text. OCR can be challenging for documents with poor image quality and unusual fonts. Some PDFs use encryption with password protection or other decryption techniques that require special handling to access the content. Finally, PDFs may conform to different standards, such as PDF/A or PDF/UA which affects data extraction coverage and quality.

Now that you understand the challenges, you can discuss the processing techniques that address these challenges and harvest the value of incorporating these documents into LLM systems.

HTML and PDF processing techniques

There are many available techniques to extract the relevant content from HTML and PDF documents. The right technique depends on the content representation. Static HTML can usually be processed with an HTML parser, while dynamically generated HTML might require a browser-based approach that executes JavaScript. PDFs with selectable text can be processed directly with PDF parsing libraries, while scanned or image-based documents require **optical character recognition** (**OCR**) before text extraction. After the initial extraction, additional techniques can identify the relevant information and remove noise such as navigation menus, sidebars and footers. This section goes over these techniques, starting with HTML Parsing.

HTML parsing

Use HTML parsing libraries to extract relevant data from HTML documents. HTML parsing libraries, such as `BeautifulSoup`, `Scrapy`, and `Selenium` can handle the variability in HTML structures and extract data based on specific tags, attributes, and patterns.

This sample code retrieves and parses live HTML, converting it into JSON documents using `requests` and `BeautifulSoup` libraries.

```
import requests, json
from bs4 import BeautifulSoup
from urllib.parse import urljoin
import pandas as pd

URL = "https://quotes.toscrape.com/"
HEADERS = {
    "User-Agent": "Mozilla/5.0",
    "Accept-Language": "en-US,en;q=0.8",
}

resp = requests.get(URL, headers=HEADERS, timeout=20)
resp.raise_for_status()
soup = BeautifulSoup(resp.text, "lxml")
title = soup.title.get_text(strip=True) if soup.title else ""

quotes = []
for q in soup.select("div.quote"):
    text = q.select_one("span.text")
    author = q.select_one("small.author")
    tags = [t.get_text(strip=True) for t in q.select("div.tags a.tag")]
    quotes.append({
        "quote": text.get_text(strip=True) if text else "",
        "author": author.get_text(strip=True) if author else "",
        "tags": tags
    })

next_link = soup.select_one("li.next a")
next_page = urljoin(URL, next_link["href"]) if next_link and next_link.has_attr("href") else None

result = {
    "source_url": resp.url, "title": title,
    "count": len(quotes), "next_page": next_page
}
print(json.dumps(result, indent=2, ensure_ascii=False))
print(json.dumps(quotes[:3], indent=2, ensure_ascii=False))

df = pd.DataFrame(quotes)
print(df.head(5))

# Chapter_2_Notebook.ipynb: Real website HTML parsing
```

Now, you can examine the parsing process of PDF documents using different libraries.

PDF parsing

Use PDF parsing libraries to extract text, images, and other data from PDF documents. PDF parsing libraries such as `PyPDF`, `PDFMiner`, and `LlamaParse`

handle multiple layouts and structures. They also support extracting data based on specific patterns and rules.

This example code fetches and parses a PDF document, converting it into JSON using `requests` and `PDFMiner` libraries:

```
import io, json, re, requests
import pandas as pd
from pdfminer.high_level import extract_text, extract_pages

PDF_URL = "https://arxiv.org/pdf/1706.03762.pdf"

def fetch_pdf(url):
    r = requests.get(url, timeout=60)
    r.raise_for_status()
    return r.content, r.url

# Decode pdf metadata dicts (bytes -> str) into a clean {key: value} map
def decode_meta(info_list):
    ...

def extract_urls(text):
    return re.findall(r"https?://[^\s)>\]}]+", text or "")

pdf_bytes, final_url = fetch_pdf(PDF_URL)
page_count = sum(1 for _ in extract_pages(io.BytesIO(pdf_bytes)))
pages = list(range(min(page_count, 5))) if page_count else []
scanned_text = extract_text(
    io.BytesIO(pdf_bytes), page_numbers=pages) if pages else ""
urls = sorted(set(extract_urls(scanned_text)))[:200]

result = {
    "source_url": final_url, "pdf_meta": {"pages": page_count},
    "extracted": {"scanned_pages": len(pages)}, "links_found": urls
}
print(json.dumps(result, indent=2, ensure_ascii=False))
if urls:
    df = pd.DataFrame(urls, columns=["url"])
    # Display first 10 links in a table
    ...
else:
    print("No links found.")

# Chapter_2_Notebook.ipynb: PDF parsing demo
```

Beyond basic PDF parsing, we also need OCR. Let's examine it next.

Optical character recognition (OCR)

Apply OCR techniques such as `Tesseract`, `EasyOCR`, and `PaddleOCR` to extract text from scanned PDFs or images. OCR can recognize text in images and extract it into a machine-readable format. This code snippet applies English OCR on PDFs using the `Tesseract` library and converts it into JSON.

```
import io, json, requests, pymupdf, pytesseract, pandas as pd
from PIL import Image

PDF_URL = "https://www.irs.gov/pub/irs-pdf/fw9.pdf"
LANGS, CONF_THRESHOLD = "eng", 60
PAGES_TO_OCR, DPI = 2, 300

def fetch_pdf(url):
    r = requests.get(url, timeout=60)
    r.raise_for_status()
    return r.content

def ocr_page(image, langs=LANGS):
    ...  # run pytesseract to get per-word boxes
         #  with confidences and full-page text
    return [...], "..."

pdf_bytes = fetch_pdf(PDF_URL)
doc = pymupdf.open(stream=pdf_bytes, filetype="pdf")
pages, rows = [], []
for i in range(min(doc.page_count, PAGES_TO_OCR)):
    img = Image.open(io.BytesIO(doc[i].get_pixmap(
        dpi=DPI, alpha=False
    ).tobytes("png")))
    words, text = ocr_page(img, langs=LANGS)
    ...  # append page dict with index, word_count_kept, text_preview,
    ...  # words[:N] extend rows with word boxes for preview table

result = {
    "source_url": PDF_URL, "pages_total": doc.page_count,
    "pages_scanned": len(pages), "dpi": DPI,
    "lang": LANGS, "min_conf_kept": CONF_THRESHOLD, "pages": pages
}
print(json.dumps(result, indent=2, ensure_ascii=False))
df = pd.DataFrame(rows)
print(df.head(20)
      if not df.empty else "No words passed the confidence threshold.")

# Chapter_2_Notebook.ipynb: OCR a live PDF
```

So far, you have examined multiple ways to extract text out of HTML, PDF and images. You still need to focus on the relevant pieces of text from these documents with data extraction.

Data extraction

Use data extraction techniques to extract specific data from HTML and PDF documents based on particular patterns, rules, or templates. The specific patterns and rules are largely different between websites and document templates.

In this example code, you extract the key information from an example invoice PDF and store it in structured JSON:

```
PDF_PATH = "../data/invoice_sample.pdf"
DPI = 400

# Render the local PDF to pixels; do not read its embedded text.
with pymupdf.open(PDF_PATH) as document:
    page_image = document[0].get_pixmap(dpi=DPI, alpha=False)
    image = Image.open(io.BytesIO(page_image.tobytes("png")))

# Recognize words across the whole page, with layout information.
data = pytesseract.image_to_data(
    image, lang="eng", config="--oem 3 --psm 3",
    output_type=Output.DICT
)
...

# Group words into lines in reading order.
lines = {}
for index, word in enumerate(data["text"]):
    if word.strip():
        key = (data["block_num"][index],
               data["par_num"][index],
               data["line_num"][index])
        lines.setdefault(key, []).append(index)
...
full_text = ...

def find_field(pattern):
    match = re.search(pattern, full_text, re.IGNORECASE)
    if not match:
        raise ValueError(f"OCR missed invoice field: {pattern}")
    return match.group(1)

fields = {
    "invoice_number": find_field(r"Invoice\s+Number\s+([A-Z0-9-]+)"),
    "total_due": find_field(r"Total\s+Due\s+\$?([\d,]+\.\d{2})"),
}
... # The notebook also extracts dates and emails, estimates OCR
... # confidence, saves the structured result, and displays a preview.
```

```
# Chapter_2_Notebook.ipynb: OCR + Data Extraction from invoice PDF
```

You have examined the challenges of parsing HTML and PDF documents along with the primary parsing techniques that are necessary for overcoming these challenges. Next is a small brief of additional high-level best practices when working with HTML and PDF documents.

Best practices for processing HTML and PDF

HTML and PDF documents are rich sources of data. They are also error-prone data sources for any data pipeline. They have highly diverse and complex layout structures. Effective use of multiple techniques is essential for successful data extraction from these formats.

You should consider using **specialized libraries** for processing HTML and PDF documents because these formats represent layout and structure in special ways that simple parsers struggle with. Special tools are necessary to preserve the structure of information and the meaningful relationships between them during extraction. The problem is not just the complex representation. It is also high **variability of representations**. Even within a single data source, these documents differ in formatting and markup conventions. Effective data extractors assume inconsistency by default and rely on flexible parsing strategies that can handle structural diversity.

After extraction, you still need additional layer of **data cleaning**. The cleaning step includes noise removal, auto-correction of obvious extraction errors, and content normalization into standard representations for the downstream retrieval, training and evaluation tasks. This step is more critical for model quality than the choice of parser itself.

Finally, you need overall **quality control**. Automated document processing introduces silent failures that are easy to miss without explicit checks. Quality control includes validation of extracted data, monitoring output consistency over time, and tracking quality metrics. Quality control ensures that HTML and PDF sources remain valuable assets rather than liabilities in your data pipeline.

Application of these combined practices make HTML and PDF document processing feasible at large scale. They allow teams to extract value from these

complex formats without the structural noise. With the right mix of tooling and process, these sources can reliably support both generalization and domain grounding for LLM applications.

The next section shifts your focus to public datasets. Public datasets are often clean, well-curated and widely reused, which makes them a powerful data source for both broad high-quality training and specializing models in key domains.

Leveraging public datasets

Public datasets are pre-existing data collections made available for public use, often curated and maintained by organizations, research institutions, or individuals. These datasets are the backbone of developing and training foundation LLMs. They combine large-volume and high-quality data that can improve model performance, accuracy, and generalization abilities.

In this section, you will understand the key benefits of public datasets compared to other data sources. You will also be introduced to representative training and benchmarking public datasets and the key considerations for selecting and using them.

Benefits of public datasets

Compared with the other data sources you have examined, public datasets offer these key benefits:

- **Accessibility**: Public datasets are readily available, reducing the need for extensive data collection and curation efforts.
- **Diversity**: Public datasets often contain diverse and representative data, which can help improve the generalizability and robustness of LLMs.
- **Quality**: Experts curated and maintained many public datasets, ensuring high-quality data that can support model training and evaluation.
- **Community-driven**: Public datasets often benefit from community contributions, updates, and feedback, ensuring the data remains relevant and accurate.

When it comes to using public datasets for LLMs and LLM applications, I prefer to classify them into training and evaluation or benchmarking datasets.

Although some datasets might serve both training and benchmarking datasets, generally they have different characteristics. The next two sections cover them separately, starting with training datasets.

LLM training datasets

Pretraining corpora are the base layer of an LLM's capability. Before a model can be useful for any enterprise, domain-specific, or consumer application, it needs broad exposure to how language is used at scale: different writing styles, different levels of formality, long- and short-form structure, and the long tail of topics that show up in real user prompts.

It is tempting to treat training data as "a big pile of text," but in practice these corpora serve different purposes. Some maximize breadth and freshness, some reinforce long-form coherence, some teach structure and syntax through code, and some extend the model beyond English. That distinction matters because it determines what the model learns easily versus what it will struggle with later, even after fine-tuning.

Below are representative **foundational pretraining corpora**, grouped by the signal they primarily contribute.

Web-based corpora emphasize coverage and diversity. They are often the backbone of general-purpose language understanding, but they also tend to include more noise and duplication than other categories.

- **Common Crawl** is a massive, freely available archive of web crawl data maintained by the non-profit organization Common Crawl. Filtered Common Crawl data has been used as part of the training mixtures for models such as GPT-3 and LLaMA (Brown et al., 2020; Touvron et al., 2023). See the example code in thet `"Common Crawl mini-sampler"` cell of the chapter notebook.
- **C4 (Colossal Clean Crawled Corpus)** is a large-scale English text dataset derived from Common Crawl and used for pretraining T5 (Raffel et al., 2020). See the example code in thet `"C4 (Colossal Clean Crawled Corpus) mini-sampler"` cell of the chapter notebook.
- **RefinedWeb** is a high-quality dataset of English web text created by the Technology Innovation Institute. See the example code in thet `"RefinedWeb mini-sampler"` cell of the chapter notebook.

Books and academic corpora emphasize longer context and structured writing. These datasets are typically where models pick up cohesive storytelling, higher editorial consistency, and more stable language patterns:

- **BookCorpus** is a corpus of approximately 985 million words from self-published books, and was included in the pretraining data for models such as RoBERTa and XLNet (Liu et al., 2019; Yang et al., 2019). See the example code in thet `"BookCorpusOpen mini-sampler"` cell of the chapter notebook.
- **The Pile** is an approximately 800 GB corpus assembled from 22 source datasets and was used to train models, such as GPT-Neo. See the example code in thet `"The Pile mini-sampler"` cell of the chapter notebook.

Code corpora teach strict structure. Code behaves like a high-signal language: syntax is strict, patterns repeat, and correctness matters. These datasets tend to strengthen tool-like behavior and structured reasoning:

- **StarCoder Training Dataset** is a code-centric dataset containing 783 GB of code in 86 programming languages, 54 GB of GitHub issues, 13 GB of Jupyter notebooks, and 32 GB of GitHub commits. It was used to train StarCoder and StarCoderBase models. See the example code in thet `"StarCoder mini-sampler"` cell of the chapter notebook.

Multilingual corpora expand representation beyond English. They improve cross-lingual generalization and robustness to mixed-language inputs, which becomes increasingly important in global products.

- **Responsible Open-source Online Text Sources (ROOTS)** is a 1.6 TB multilingual corpus spanning 59 natural and programming languages and was used to train the BigScience Large Open-science Open-access Multilingual (BLOOM) model. Read example code at `"Multilingual mini-sampler"` cell of the chapter notebook.

When selecting public training datasets, the goal is not to include everything, but to choose corpora that match the intended scope of the model, including general-purpose breadth, domain depth, structured reasoning, or multilingual coverage. With that foundation in place, you can now turn to **benchmarking datasets**, which are designed not to teach models language, but to measure what they have learned.

LLM benchmarking datasets

Benchmarking datasets serve a fundamentally different role from pretraining corpora. Rather than teaching language, benchmarks are designed to measure what the model has learned and more importantly what it has not. They provide controlled, repeatable tasks. These tasks make it possible to compare models, measure progress over time, and reason about trade-offs between capabilities such as reasoning, factual knowledge, coding, and multilingual performance. However, benchmark scores depend on factors such as task design, data contamination, prompting strategy, and evaluation setup, so they should not be interpreted as absolute measures of model quality.

In practice, benchmarks act as lenses. Each one isolates a particular skill under simplified assumptions, which makes evaluation tractable but also incomplete. You need to understand exactly what a benchmark measures, because benchmark results influence model selection, release decisions, and perceptions of "state of the art."

When selecting benchmarking datasets, the goal is not to maximize leaderboard scores, but to assemble a measurement suite that reflects the behaviors that matter for the intended application. No single benchmark captures all of reasoning depth, robustness, safety, and quality of user experience. Additionally, improvements on one dataset can mask regressions elsewhere.

For that reason, benchmarks should be treated as diagnostic tools rather than final verdicts. They are most valuable when used in combination, alongside qualitative analysis and task-specific evaluations. With this context in mind, you can now look at the most used benchmarking datasets and the specific capabilities each one is designed to probe.

- **massive multitask language understanding** (**MMLU**) is a benchmark covering 57 subjects with varying difficulty levels, assessing LLMs' general knowledge and reasoning abilities. See the example code in thet `"MMLU sample"` cell of the chapter notebook.
- **HellaSwag** is a challenging dataset focusing on common sense reasoning, testing LLMs' ability to understand and complete sentences. See the example code in thet `"HellaSwag sample"` cell of the chapter notebook.

- **HumanEval** is a benchmark for evaluating code generation capabilities, assessing the functional correctness of LLM-generated code. See the example code in thet `"HumanEval sample"` cell of the chapter notebook.
- **BIG-Bench (Beyond the Imitation Game Benchmark)** is a collection of challenging tasks, pushing the boundaries of LLM reasoning abilities. See the example code in thet `"BIG-Bench sample"` cell of the chapter notebook.
- **ARC (AI2 Reasoning Challenge)** is a benchmark focusing on scientific reasoning, using grade-school science questions in a multiple-choice format. See the example code in thet `"ARC (AI2 Reasoning Challenge) sample"` cell of the chapter notebook.
- **GSM8K (Grade School Math 8K)** is a benchmark designed explicitly for evaluating LLMs' mathematical reasoning skills, focusing on elementary math problems. See the example code in thet `"GSM8K (Grade School Math 8K) sample"` cell of the chapter notebook.
- **TruthfulQA** is a dataset for evaluating LLMs' truthfulness and ability to avoid generating wrong or misleading information. See the example code in thet `"TruthfulQA sample"` cell of the chapter notebook.

By leveraging public benchmarking datasets, developers can evaluate how their applications perform relative to other models, identify opportunities for improvement, and update their systems accordingly. This iterative process helps produce AI assistants that are more accurate and useful in practice.

To apply these benchmarks effectively, developers should be intentional in data selection. The following section discusses the criteria for public datasets selection.

Public dataset selection

When selecting public datasets, target diversity and relevance to your use case. Effectiveness of public datasets depends on matching the distribution and goals of the target application. Use comprehensive and representative datasets to improve model generalization and robustness and reduce overfitting to narrow use cases or styles. The same selection criteria apply to both training and benchmarking data.

When selected thoughtfully, public datasets offer researchers and developers access to high-quality data that supports the development and evaluation of effective LLMs. You have one final category of data to consider: **licensed data**. The next section explores how licensed datasets complement public data and what is special about them.

Exploring licensed data

Licensed data refers to **external datasets acquired from third-party vendors** under commercial licensing agreements. These datasets are either pre-curated for a commonly known scenario or custom-built according to provided specifications. Because these datasets are expensive, organizations typically purchase them to address gaps not covered by public and internal data.

For this chapter, licensed data does not include data provided by customers or partners. Customer data is treated as domain data, with different assumptions around consent and usage constraints. What you mean by licensed data is the data that is sourced from external data vendors and governed by commercial and legal terms rather than customer relationships.

Licensed data is most valuable when **data precision is the bottleneck** rather than scale or diversity. It is best used to improve performance in new or restricted domains, build evaluation ground truth with trusted reference answers or labels against which you can assess model outputs, and provide high-quality labels that are more expensive to generate in-house. In the following sections, you examine the benefits and common types of licensed data, then discuss the contractual and operational considerations for using it properly.

Benefits of licensed data

Licensed datasets are typically acquired to address one or more gaps in the existing data. When evaluating these datasets, consider their data quality, domain coverage, annotation quality, and how well they have been maintained over time.

First is **data quality**. Data vendors curate the datasets to remove duplicates, low-quality content, and inconsistent formatting if desired. This improves training efficiency and reduces unintended learnings. Second is **coverage of restricted domains**. Certain domains are poorly represented in public data or

require specific content structure and terminology. Licensed data provides this coverage in a controlled and repeatable way.

For evaluation data, the main value is **professional annotations and labels**. You look for highly precise labels and robust rubrics produced by subject-matter experts. Lastly, the best licensed datasets should be **versioned and maintained over time**, making them ideal for regression testing, benchmarking, and release gating. These benefits come from different types of licensed data. Curated text data provides controlled domain coverage; annotated datasets provide precise labels and evaluation ground truth, while structured knowledge provides consistent and normalized information. Understanding these categories helps you identify the types of licensed data that address the gaps in your existing datasets.

Common types of licensed data

In practice, externally licensed datasets used for LLM development tend to fall into a small number of categories:

- **Curated text corpora:** Professionally assembled collections of documents aligned to a specific domain, format, or taxonomy.
- **Annotated fine-tuning or evaluation sets:** Datasets packaged explicitly for supervised learning or model assessment. For these datasets, precision and consistency are the primary differentiators.
- **Structured or semi-structured knowledge sources:** Data with explicit fields, hierarchies, or normalization that would be difficult to reconstruct from public sources.

Guidance for using licensed data

Vendor-provided data introduces constraints that must be handled for both training and evaluation workflows. The permitted use depends on the specific license and applicable jurisdiction. For example, a license may permit using a dataset for evaluation while prohibiting its use for training, derivative data generation, or redistribution. These constraints must be treated as hard system requirements, and they become more complex when you use datasets provided by multiple vendors.

Whenever you have scope limitations on portions of your data, data isolation and access controls become necessary. Your data storage system must

separate licensed data from other data sources and provide restricted access and auditing features. This affects storage design choices, feature generation and debugging workflows. Additionally, derived artifacts such as embeddings and fine-tuned model weights could be subject to the same contractual and compliance requirements as well.

Some licenses impose retention limits and deletion requirements. Data pipelines should account for retention of both raw data and derivative artifacts including cached features and evaluation outputs. A related concern is version management and reproducibility. Treat vendor data updates as explicit dataset versions. Silent data replacement makes it difficult to reproduce results and debug regressions.

So far, you have become familiar with the non-trivial acquisition and operational costs associated with licensed data. Thus, you cannot think of licensed data as "better" than public data. Measure the positive impact of licensed data on quality versus its cost and operational overhead and make your decisions accordingly.

To recap, use licensed data selectively to complement public data, synthetic data, and internal domain data. Apply it where it provides a clear advantage that is otherwise more costly to cover such as required domain expert annotations and up-to-date evaluation benchmarks.

In this chapter, we learned that each of public data, licensed data, and domain data offers distinct strengths and tradeoffs for quality, coverage, cost, and operational constraints. No single data source is universally sufficient for building and evaluating LLM applications. The next challenge is selecting the **right combination of data** for each task and stage of development. The next section focuses on how to select relevant data sources based on use-case requirements and practical constraints.

Selecting relevant data sources

The effectiveness of AI assistants relies heavily on the quality and relevance of the data sources used to train them. Different assistants require distinct data sources to achieve the best performance. Thus, you need to understand the specific data needs of each use case to build effective and efficient AI assistants.

In this section, you can find the recommended data sources to use for key LLM assistant use cases, then close with general best practices for selecting data sources. As you read this section, you should to imagine examples of different scenarios and user questions that will make use of each one of these data sources.

Data sources for assistant use cases

Considering the selected set of use cases from previous chapter, you can determine what data sources are best applicable for each one of the use cases. Here are my recommendations:

Personal assistant

Data sources for personal assistants may include:

- User-generated content (for example, emails, messages, calendar events)
- Web data (for example, news, weather, traffic)
- Public datasets (for example, knowledge graphs, entity recognition)
- Personal preferences and history (for example, music, movies, or book preferences)
- Location-based data (for example, the global positioning system, geolocation)

Domain-specific assistant

The key data sources for domain-specific assistants include:

- Domain-specific documents and knowledge bases (e.g., medical textbooks, financial reports)
- Industry reports and research papers (e.g., market trends, scientific studies)
- Specialized datasets and ontologies (e.g., medical terminology, financial concepts)
- Expert interviews and consultations
- Industry-specific regulations and guidelines

Enterprise assistant

Data sources for enterprise assistants may include:

- Internal documents and knowledge bases (e.g., company policies, employee handbooks)
- Enterprise-specific data and metadata (e.g., customer information, sales data)
- Logs of existing enterprise systems and tools (e.g., CRM, ERP)
- Employee feedback and performance data
- Industry benchmarks and best practices

Content moderator

For a content moderator, the key data sources could be:

- Labeled datasets of moderated content (e.g., hate speech, spam)
- Industry guidelines and policies (e.g., community standards, terms of service)
- User-generated reports and feedback (e.g., flagging, reporting)
- Contextual information (e.g., user demographics, content metadata)

Beyond these key applications, you can briefly review general best practices for selecting data sources.

Best practices when selecting data sources

The data source recommendations discussed in the previous sections can apply generally to other LLM applications by following a small set of practical best practices. These practices help ensure that data selection is intentional, aligned with the use case and manageable over time:

- **Define precise requirements:** Clearly articulate the target use case, expected behaviors, and success criteria for the model. Data sources are what you should select based on how well they support these requirements, rather than on availability or scale alone.
- **Measure data quality and robustness:** Evaluate data sources for accuracy, consistency, coverage, and noise. Low-quality data can degrade model performance and obscure true capability gaps during evaluation.
- **Combine complementary data sources:** Relying on a single data source is rarely sufficient. Combining public, licensed, and domain-specific

data can provide broader coverage, expose edge cases, and reduce blind spots in both training and evaluation.

- **Plan for maintenance and evolution:** Data relevance changes over time. Establish mechanisms to monitor data drift, refresh datasets, and retire sources that no longer reflect the target environment or task distribution.

By applying these best practices, practitioners can move from ad hoc data collection to a more systematic approach to data selection. This, in turn, enables the development of AI assistants that are more robust, efficient, and better aligned with user needs.

Summary

In this chapter, we examined how to identify, acquire, and extract relevant data from a range of sources used in training and fine-tuning large language models. You covered web data, domain-specific content, public datasets, licensed vendor data, and structured document formats such as HTML and PDF, highlighting the strengths, limitations, and practical considerations associated with each. Throughout the chapter, you covered concrete examples of how these data sources support real-world AI scenarios, including personal assistants, enterprise automation, domain-specific advisors, and content moderation systems.

The takeaway is that data quality, diversity, and relevance are primary drivers of model performance. The choice of data directly influences an LLM's accuracy, generalization, reliability, and ability to operate within technical, legal, and operational constraints. Being intentional about where data comes from, how it aligns with the target use case, and how it can be governed over time is the backbone of building trustworthy and effective LLM-powered applications.

With data sources in place, the next step is to make that data usable. The following chapter shifts from data collection to preparation, covering embeddings and vectorization, chunking, metadata enrichment, and indexing. These steps transform collected data into structured and searchable inputs for retrieval-augmented generation and fine-tuning, improving LLM application performance.

References

- Brown, T. B., et al. (2020). *Language models are few-shot learners*. Advances in Neural Information Processing Systems, 33, 1877–1901. https://arxiv.org/abs/2005.14165
- Dogan, S. (2025, April 3). *Must-Have Python Web Scraping Libraries & Tools in 2025*. AIMultiple. https://research.aimultiple.com/python-web-scraping-libraries/
- Gao, L., et al. (2020). *The Pile: An 800GB dataset of diverse text for language modeling*. https://arxiv.org/abs/2101.00027
- Kaubrė, V. (2024, September 27). *LLM training Data: The 8 main Public Data Sources*. Oxylabs. https://oxylabs.io/blog/llm-training-data
- Lee, K. (n.d.). *Open-Sourced Training Datasets for Large Language Models (LLMs)*. Kili. Retrieved June 14, 2025, from https://kili-technology.com/large-language-models-llms/9-open-sourced-datasets-for-training-large-language-models
- Leobeeson. (2024, July 13). *GitHub - leobeeson/llm_benchmarks: A collection of benchmarks and datasets for evaluating LLM*. https://github.com/leobeeson/llm_benchmarks
- Liu, Y., et al. (2019). *RoBERTa: A robustly optimized BERT pretraining approach*. https://arxiv.org/abs/1907.11692
- Li, R., et al. (2023). *StarCoder: May the source be with you!* https://arxiv.org/abs/2305.06161
- Raffel, C., et al. (2020). *Exploring the limits of transfer learning with a unified text-to-text transformer*. Journal of Machine Learning Research, 21(140), 1–67. https://www.jmlr.org/papers/v21/20-074.html
- Ryan, C. (2023, November 24). *Thorough comparison of 6 free and open source OCR Tools 2025*. Cisdem. https://www.cisdem.com/resource/open-source-ocr.html
- Saive, R., & Saive, R. (2024, November 15). *Top 7 Python OCR Libraries for Text Extraction From Images*. https://www.tecmint.com/python-text-extraction-from-images/
- Touvron, H., et al. (2023). *LLaMA: Open and efficient foundation language models*. https://arxiv.org/abs/2302.13971

- Vishe, M. (2024, June 1). *Popular PDF parsing tools for efficient document extraction*. *FutureSmart AI Blog*. https://blog.futuresmart.ai/popular-pdf-parsing-tools-for-efficient-document-extraction
- Yang, Z., et al. (2019). *XLNet: Generalized autoregressive pretraining for language understanding*. Advances in Neural Information Processing Systems, 32. https://papers.nips.cc/paper_files/paper/2019/hash/dc6a7e655d7e5840e66733e9ee67cc69-Abstract.html

3

Data Preparation: From Raw Data to Vector-Ready Knowledge

Chapter 1 guided us to a functional **large language model** (**LLM**) call. When we queried the model about an individual absent from its training data, it told us it had no information about that person. However, after providing a paragraph of relevant context, the model produced an accurate response, demonstrating in-context learning. *Chapter 2* examined valuable data sources to collect, such as PDFs, help articles, and internal documentation. The current focus is on the processes that occur between collecting these documents and achieving effective retrieval. There are numerous critical steps involved in this transition. At Clearwater Analytics, we developed a pipeline in which new **Retrieval-Augmented Generation** (**RAG**) sources became operational in under an hour. This rapid deployment was possible only because document preparation had been thoroughly addressed. The chunking process respected paragraph boundaries rather than relying on arbitrary character counts. Metadata was attached during ingestion, enabling retrieval to filter by document type and date in addition to vector similarity. In my experience, projects lacking this level of preparation often suffer; adding a new source can degrade retrieval quality rather than improve it.

That preparation work is what this chapter is about. Embeddings first, then chunking in five flavors, then metadata and graph relationships, and finally a pipeline that ties it all together in ChromaDB.

In this chapter, we will work on the following topics:

- Vectorization and embedding fundamentals: the language of AI
- Graph database integration: connecting the dots

- Vector database optimization: from prototype to production

../../EPUB/NOTE

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Vectorization and embedding fundamentals: the language of AI

How does a machine understand text? It does not, not directly. What we do is convert words and sentences into lists of numbers called vectors. A vector is just a list of numbers like [0.12, −0.03, 0.54, ...]. You can picture it as an arrow from its origin to a point in space. The direction of that arrow captures the meaning, and we measure similarity by comparing directions between arrows. When these vectors capture the meaning of the text, they are called embeddings. For example, consider these two sentences: "Apple iPhone is expensive" and "There is a new Apple iPhone." If you pass them through an embedding model, you will get two vectors that are close to each other in the space of possible embeddings, showing that they represent similar ideas, as shown in *Figure 3.1*.

We measure similarity with cosine. Cosine compares directions, not lengths. The two paraphrases "Apple iPhone is expensive" and "There is a new Apple iPhone" have a higher cosine than unrelated text like "Mango is a fruit." A smaller angle equals a higher cosine. Two sentences about the same topic yield closer numbers. They are clustered closely in a high-dimensional space, even if the words are completely different. That is the trick that makes semantic search work. The user does not need to type the exact same keywords as the document; the meaning is enough.

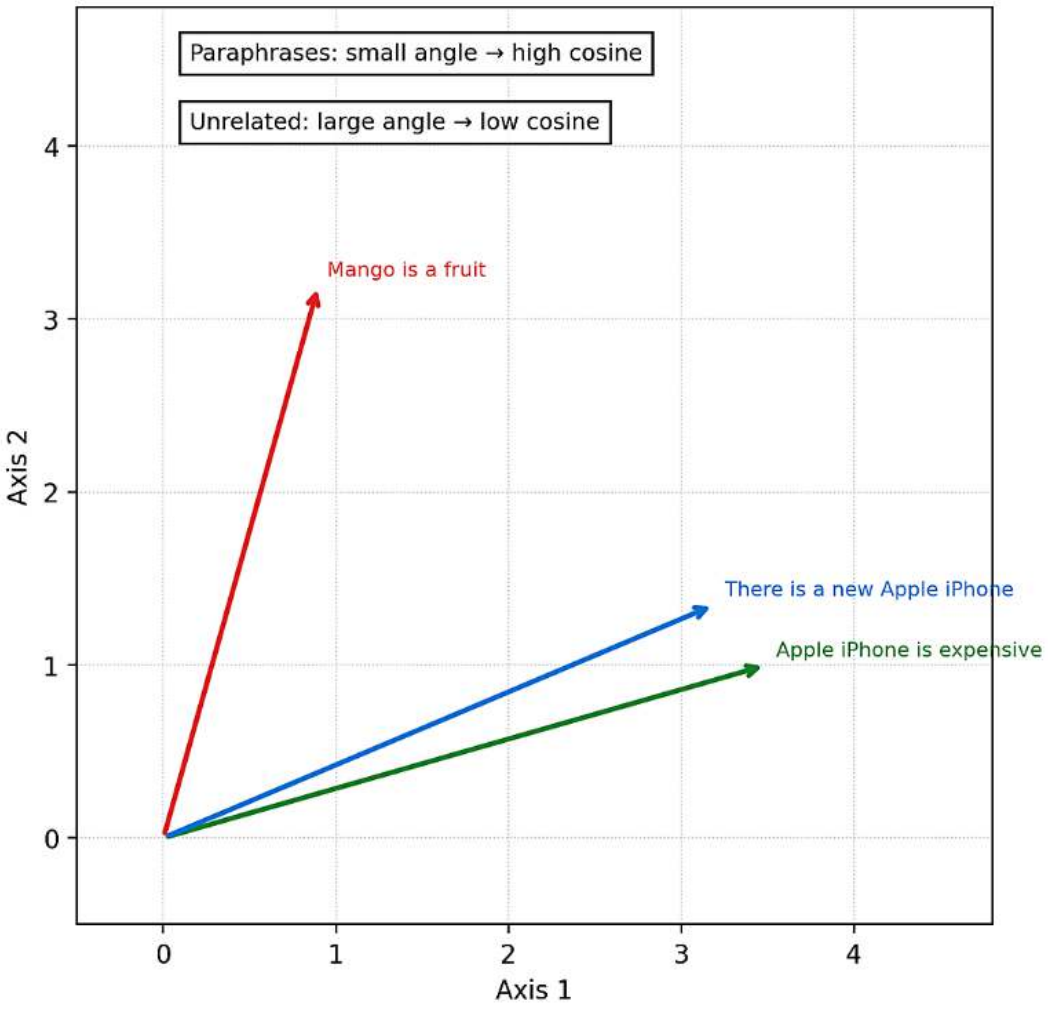


Figure 3.1: Cosine similarity measures how aligned two directions are (smaller angle -> higher cosine)

For example, someone searches for how to book a flight to London, and the system finds the international travel policy document, even though it never mentions the word 'flight'. A lawyer searches for intellectual property disputes and gets back a document about patent infringement litigation. The embedding model figured out that those phrases mean the same thing. Without embeddings, you are back to keyword matching, which is fine until your users do not use the right keywords, which is most of the time. This is what makes RAG actually useful.

RAG works through a series of steps. First, the user enters a query. This query is converted into an embedding (a vector of numbers). The system then compares this embedding to a database of embeddings created from all relevant documents. It finds the documents whose embeddings are closest to

the query in this multi-dimensional space. These relevant documents are retrieved, and their content can be provided to a language model to generate a helpful, accurate response for the user. We will understand this more as we go through the chapter.

To understand this process further, the following section walks through how these embeddings are actually created, starting with OpenAI's text-embedding-3 models because they are well-documented, easy to use, and good enough for most of what we need.

../../EPUB/NOTE

I highly recommend my articles *Evolution of Word Embeddings: A Journey Through NLP History* (`https://www.linkedin.com/pulse/evolution-word-embeddings-journey-through-nlp-history-rany-pzmjc/`), and for more details on embedding and vectorization, *Understanding Word Embedding in NLP using Sentence Transformers* (`https://www.linkedin.com/pulse/understanding-word-embedding-nlp-using-sentence-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-fcd2c/`).

Choosing your first embedding model: OpenAI's text-embedding-3 series

For this book, we will be using OpenAI's text-embedding-3-small and text-embedding-3-large. The reason I picked OpenAI is practical: the API is straightforward, the documentation is thorough, and most readers already have access to an OpenAI key from *Chapter 1*. Other providers like Cohere or Voyage AI work fine too; we cover those briefly later in the chapter.

I introduce both models because the trade-off between them comes up on every project. The small model, text-embedding-3-small, produces 1536-dimensional vectors. Fast, cheap, good enough for most text. The large model, text-embedding-3-large, doubles that to 3072 dimensions and catches finer distinctions in meaning, which matters when you are working with legal contracts or dense technical documentation where one word changes the interpretation. The large model also lets you truncate the vector down to 256 dimensions if you need to save storage, which is a useful trick in production. For this book, we use the small model most of the time because it keeps costs low while we experiment.

Your first embedding: A practical example

Enough theory. The best way to understand embeddings is to create one. Here is a simple, self-contained Python snippet that you can run in a Jupyter notebook or a Python script. All you need is the OpenAI Python library installed (`pip install openai`) and a valid API key.

../../EPUB/NOTE

The associated notebook is organized by sections, so that you can skip to the section you are interested in. The full code is available in the `chapter_03` folder at `chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb`. For local setup, follow the environment instructions in *Chapter 1*. However, the easiest way is to use Google Colab to run this notebook and avoid local setup. You need to click on the **Run all** button at the top of the Notebook to run all the cells before it. You can open the notebook directly from GitHub as we explained in *Chapter 1*.

Here is the simplest code to embed one sentence (The full code is in the notebook). This example will take a single sentence and convert it into its numerical vector representation. Run the following code in the companion notebook (`chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb`), search for `Your First Embedding` to jump to the right cell.

We're making a direct embeddings call to turn text into a numeric vector that a vector database can store and search efficiently. I like starting with a single sentence so you can see the full request-response shape before batching. Have a look at the following code:

```
import os
from openai import OpenAI

client = OpenAI()

text_to_embed = "Data strategy is the key to powerful AI."

EMBEDDING_MODEL = "text-embedding-3-small"

response = client.embeddings.create(
    input=text_to_embed,
    model=EMBEDDING_MODEL
)
```

```
embedding = response.data[0].embedding

print(f"Original Text: {text_to_embed}")
print(f"Embedding Vector (first 5 dimensions): {embedding[:5]}...")
print(f"Total dimensions: {len(embedding)}")
```

The code sends one sentence to the OpenAI embedding API using the small model and gets back a response object. The vector itself sits at `response.data[0].embedding`, a list of 1536 floating-point numbers for text-embedding-3-small. We pull it out, print the first five numbers so you can see what an embedding actually looks like, and print the total length. You should see 1536, which is the dimensionality your vector database will expect when you configure it with the same model. Any single number does not mean much on its own; the meaning only shows up when you compare the full vector against others. In production, you would typically pass a list of strings rather than a single sentence at a time; batching helps keep costs down, as shown here:

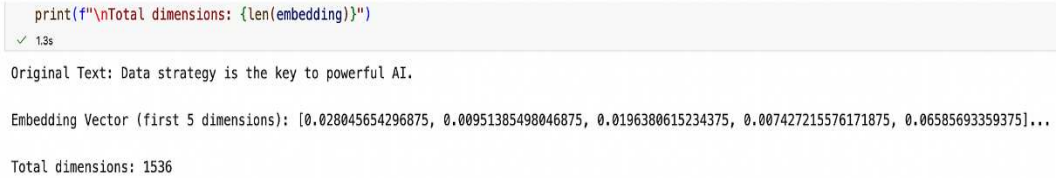


Figure 3.2: The Output of the Code with the first 5 Dimensions of the Embedding Vector

Figure 3.2 shows the printout. First five numbers of the vector: [0.028077619, 0.009524666, 0.019656669, 0.007422348, 0.065825894]. That is five out of 1536 total. Those 1536 floating-point numbers are what text-embedding-3-small gives you for a single sentence. Individually they mean nothing; the model learned what to put where during training, and there is no way to say dimension 47 represents sentiment or dimension 900 represents topic. The useful part is comparison. When your RAG system needs to find relevant text, it embeds the query with the same model and computes cosine similarity or dot product between the query vector and every stored vector. Closer means more similar in meaning. I want to stress the same model part because I have seen people index with one model and query with another; the dimensions do not even line up and every result comes back garbage.

The following figure shows how the full embedding process works from raw text to similarity search:

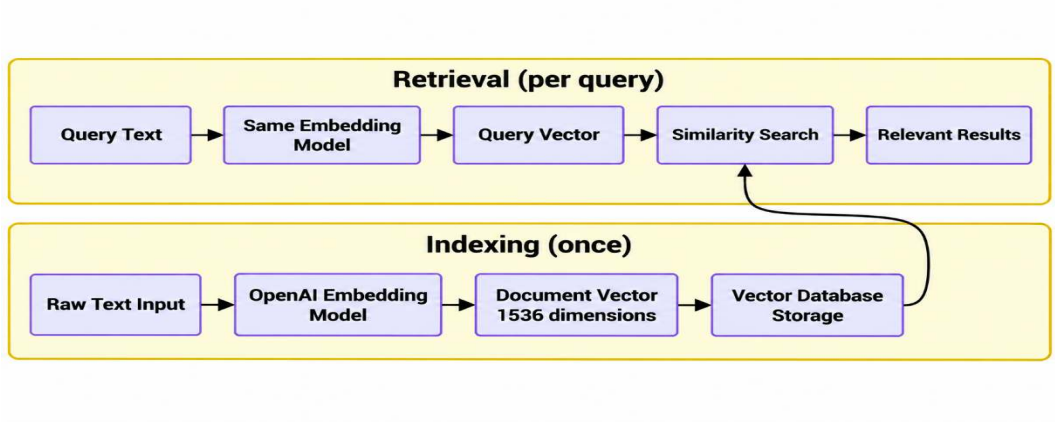


Figure 3.3: Embedding Process Flow: From Text Input to Similarity Search

This diagram shows how raw text gets transformed into vectors, stored in a database, and then used for similarity search when processing queries. The same embedding model processes both your source documents and user queries, ensuring they exist in the same semantic space for accurate matching. To learn more about word embeddings, I highly recommend my article Understanding Word Embedding in NLP using Sentence Transformers (https://www.linkedin.com/pulse/understanding-word-embedding-nlp-using-sentence-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-fcd2c/).

Similarity numbers: A quick check

We can verify the example in *Figure 3.1* with real numbers. We embed the same two iPhone paraphrases and the unrelated sentence "*Mango is a fruit*," then compute cosine similarities.

You can run this code in the companion notebook (*chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb*) under the section titled **Similarity numbers: a quick check**. Have a look at the following code:

```
import numpy as np
from openai import OpenAI

client = OpenAI()

# Reuse the same model defined earlier in the chapter/notebook
EMBEDDING_MODEL = "text-embedding-3-small"

sents = [
    "Apple iPhone is expensive",
    "There is a new Apple iPhone",
    "Mango is a fruit",
```

```
]

resp = client.embeddings.create(input=sents, model=EMBEDDING_MODEL)
vecs = [d.embedding for d in resp.data]

def cosine(a, b):
    a, b = np.array(a), np.array(b)
    return float(a @ b) / (np.linalg.norm(a) * np.linalg.norm(b))

print("cos(Apple iPhone is expensive, There is a new Apple iPhone):",
      round(cosine(vecs[0], vecs[1]), 3))
print("cos(Apple iPhone is expensive, Mango is a fruit): ",
      round(cosine(vecs[0], vecs[2]), 3))
print("cos(There is a new Apple iPhone, Mango is a fruit): ",
      round(cosine(vecs[1], vecs[2]), 3))
```

This is the output you get:

```
]  ✓ 1.2s

cos(Apple iPhone is expensive, There is a new Apple iPhone): 0.618
cos(Apple iPhone is expensive, Mango is a fruit):  0.194
cos(There is a new Apple iPhone, Mango is a fruit):  0.211
```

Figure 3.4: Code output shows Cosine Similarity between sentences

In *Figure 3.4*, you should see a higher cosine for the two iPhone paraphrases (0.618), and a noticeably lower cosine for pairs that include the unrelated "Mango is a fruit" sentence (0.194 and 0.211). This connects directly to the RAG pipeline we are building. We embed both knowledge chunks and user queries into the same space, then retrieve the closest chunks by similarity. The better the embeddings, the more likely we find the right chunk, even if the words differ.

In the sections that follow, we prepare documents so that each chunk represents a single idea. Combined with embeddings, this yields high-precision retrieval, which we then pass to the model for generation.

../../EPUB/NOTE

Mini-glossary (quick definitions)

Embedding space: the "map" where each word/sentence becomes a point.

Vector: the list of numbers that places a point on that map. Cosine similarity compares the directions of these vectors to measure meaning.

> **Cosine similarity**: how aligned two directions are (closer to 1 = more similar).

A quick word on other embedding models

While we use OpenAI's embeddings as a practical baseline, the same concepts apply across providers. If you prefer open-source, models such as BGE-M3, E5, or Nomic Embed run locally and give you strong quality with privacy and cost control. If you need a hosted API with broad multilingual coverage, Cohere's embed-v4 handles text and images in a single call across over a hundred languages.

When choosing an embedding model for production, look beyond raw quality:

- **Latency & throughput**: Can it keep up with your ingestion and query rates?
- **Cost**: Per-request pricing vs. local GPU/CPU costs.
- **Multilingual coverage**: Is it needed for your audience and data?
- **Licensing & governance**: Especially important in enterprise settings.

The good news: your RAG architecture stays the same. You'll still create vectors, store them in a vector database, and compare them with cosine (or another metric). Only the client library and model name change.

Everything we have covered so far assumes your data is text. But what happens when your documents have screenshots, architecture diagrams, or charts embedded in the PDF? Text embeddings simply skip those. Starting with CLIP from OpenAI in 2021, a family of multimodal embedding models now maps both images and text into the same vector space. You feed it a photograph and a sentence, and both come back as vectors you can compare with cosine, same as before. I am not covering CLIP implementation in this book because the pipeline changes are significant: you need an image extraction step, a separate model call, and usually a larger index. But if your enterprise data is heavy on visuals, know that the option exists.

The indexing pipeline: building our knowledge catalog

Now we get to the fun part: writing the code for our Indexing Pipeline. This is the process where we take our raw documents and prepare them for our AI to

use. It's a one-time, offline process that makes the real-time querying fast and efficient. Our goal is to create a simple, reusable pipeline that can store those embeddings and the original text in our ChromaDB vector database.

Before we dive into full documents, we'll begin with the smallest possible indexing example, a single sentence, to verify the end-to-end path and persistence. Later in this chapter, we'll upgrade to chunked ingestion, where we split longer pages into coherent pieces before embedding and indexing. Chunking is an independent improvement, so seeing the one-sentence path first keeps things clear.

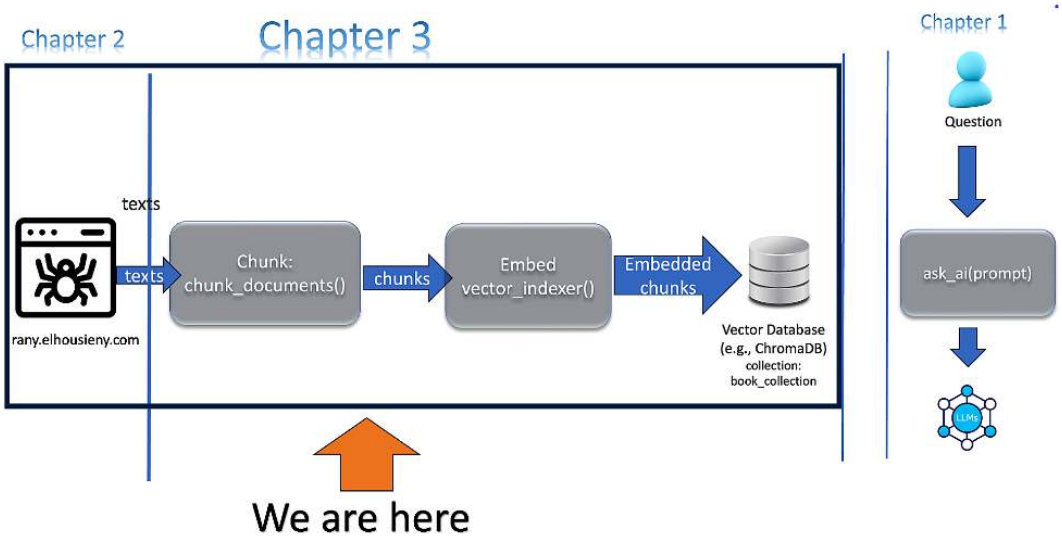


Figure 3.5: The Complete Pipeline: From Crawling (Chapter 2) Through Indexing (Chapter 3)

As shown in *Figure 3.5*, we start with data collection from multiple sources (*Chapter 2*). We then chunk that content and generate embeddings (*Chapter 3*). *Chapter 4* will connect *Chapter 1* with *Chapter 3*. In *Chapter 4*, we will build the retriever that pulls the related context and augments the prompt before sending it to the AI. In this chapter, we store those embeddings and metadata in a vector database and build a simple indexing pipeline. In the next section, we'll implement a minimal indexing pipeline so you can see the basics end-to-end.

The code: Step-by-Step implementation

The full code is available in the `chapter_03` folder at `chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb`. The easiest way is to use Google Colab to run this notebook and avoid local setup. You will find this

code under the subsection *The Indexing Pipeline: Building Our Knowledge Catalog*. For this example, we're going to build everything in a single, clear Jupyter notebook. I've chosen to use ChromaDB because it's incredibly easy to get started with. We'll set it up to save our database to disk. In the notebook, search for `Create ChromaDB client` to find the cell for the following code snippet:

> **../../EPUB/NOTE**
>
> The complete, runnable code is in the companion notebook. Below are short snippets that highlight the key steps.

```
chroma_client = chromadb.PersistentClient(path="DB PATH HERE")
```

Here we're initializing a persistent ChromaDB client that will store our vector database in a local directory of your choice. In the notebook we used a shared DB storage on the root directory in the path `data/chroma_db`. I chose persistent storage because it means you can close your notebook, come back later, and all your data will still be there. No need to re-run everything from scratch.

> **../../EPUB/NOTE**
>
> Note if you are using Google Colab, you will need to store it in a Google Drive folder to avoid creating it every time the environment is reset.

I am creating a shared database path for ChromaDB at the root of the repo in the following code snippet (search for `Shared database path for ChromaDB`):

```
SHARED_DB = repo_root / 'data' / 'chroma_db'
SHARED_DB.mkdir(parents=True, exist_ok=True)
```

When you execute the following code snippet (you can search for it in the Notebook with the following query `Create ChromaDB client`), it will create a folder at the root of the repo:

```
chroma_client = chromadb.PersistentClient(path=str(SHARED_DB))
```

This is depicted in the following screenshot:

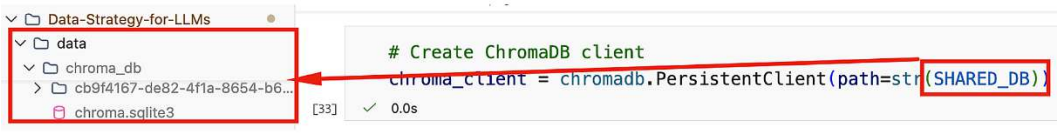


Figure 3.6: ChromaDB Folder

To help you understand the concept, I will show you how to create a data folder on a custom path. You can try the command on cell 82 (to find the cell, you can search for `chromadb.PersistentClient(path="data/rany")`). If you ran the entire Notebook at the beginning, you will find the folder created as shown in *Figure 3.7*:



Figure 3.7: Creating Chroma Client Creates a New Folder

As you can see in *Figure 3.7*, it created the folder in the same folder as the notebook. Feel free to change the path from `data/rany` to anything else and run the cell again, then watch the created directory.

We persist the database at the root of the repo. You can change that by changing the SHARED_DB variable. The reason this persistence matters is practical: as we build out our RAG system, we'll be running different parts of the code in different cells. Without persistence, you'd have to re-index all your documents every single time you restart your notebook, which gets tedious fast. This way, once we add documents to our collection, they stay there until we explicitly remove them. I want to be clear about something: Using local file storage like this is ok for learning and experimentation, but in a real production system, you'd want to use a managed vector database with proper lifecycle management, observability, and access controls. But for understanding how RAG works, this approach gives us exactly what we need without the complexity of setting up external services.

Now, we need to specify the embedding model, as we covered earlier. Let's use `embedding-3-small`. We're going to set up the embedding function that turns text into vectors so the retriever can index and search efficiently. I prefer to define this once, early, so everything downstream (collection creation, add/upsert) uses a single, consistent embedding configuration. To find the cell

in the Notebook containing the following code snippet, search for "`Set up embedding function.`":

```
from chromadb.utils import embedding_functions

ef = embedding_functions.OpenAIEmbeddingFunction(
    api_key=os.getenv("OPENAI_API_KEY"),
    model_name="text-embedding-3-small",
)
```

The previous code snippet initializes `embedding_functions.OpenAIEmbeddingFunction` and assigns it to `ef`. It pulls the API key from the environment via `os.getenv("OPENAI_API_KEY")`, so there are no hardcoded secrets, and specifies `model_name="text-embedding-3-small"` as a cost-efficient default. Practically, `ef` is just a callable that accepts a list of strings and returns their embeddings; vector stores (e.g., when you call `create_collection(…, embedding_function=ef)`) will invoke it automatically during add/upsert.

Now, let's create a collection inside this DB:

```
collection = chroma_client.get_or_create_collection(
    name="book_collection",
    embedding_function=ef
)
```

This line opens or creates a Chroma collection. Think of it as a vector 'table' that will hold embeddings along with any documents and metadata we choose to store. The `get_or_create_collection(...)` call is idempotent: if the collection already exists, it returns a handle; otherwise, it creates one. You can run this cell repeatedly without recreating the index. Note that the original text is only stored when you add it later via `collection.add(documents=[...], metadatas=[...], ids=[...])`, and changing embedding models on an existing collection will require a new collection and a full re-index.

After running the above code, you will have a folder called `chroma_db` in the root directory of the repo, inside the `data` folder as shown in *Figure 3.6*. This folder contains the ChromaDB database. Inside this database will be the `book_collection` collection. You can verify the results by running the following code (Search for `Verify collection was created` in the Notebook):

```
print(f"Database directory exists: {SHARED_DB.exists()}")
print(f"Collection count: {collection.count()}")
print(f"Collection name: {collection.name}")
```

When you run the code in the notebook, you will get the following output as shown in *Figure 3.8*:



Figure 3.8: The output of the verification of the Collection

When you run this verification code, you'll see three key pieces of information that tell us everything is working correctly. The first result, `Database directory exists: True`, confirms that ChromaDB successfully created the `chroma_db` directory on your local file system. This is important because it means your vector embeddings and metadata will persist between notebook sessions and kernel restarts. You won't lose your work when you close and reopen the notebook. You can verify your current folder and you will find a folder named `data/chroma_db` as shown in *Figure 3.6*.

The second result, `Collection count: 0`, shows that our `book_collection` is empty, which is exactly what we expect immediately after creation. This zero count confirms the collection exists and is ready to receive documents, but we haven't populated it with any data yet. Think of it like creating an empty folder. It's there and ready to use, but there's nothing in it yet.

Finally, **"Collection name: book_collection"** validates that the collection was created with the intended name. This ensures proper organization and identification within your RAG system. This name will be used throughout the chapter for document storage and retrieval operations.

Setup confirmed. Next, we add sample documents. ChromaDB embeds them automatically, and we run similarity queries to see if the right chunks come back. Eventually that plugs into `ask_ai` from `Chapter 1`. I am spending time on the verification because I have been bitten before; collection looked fine,

turned out the embedding function was not wired and every query returned garbage. Five minutes of sanity checking here saves hours of debugging later.

Now, we will be adding a single sentence to the collection to verify the indexing path end-to-end. I like doing this quick sanity check before batch ingestion so we know the collection accepts new items and the embedding function is wired correctly:

```
collection.add(documents=["This is a quick test sentence to index."], ids=["manual_1"])
```

Calling `collection.add(documents=["This is a quick test sentence to index."], ids=["manual_1"])` inserts one record with a unique ID. Because we passed `embedding_function=ef` when creating the collection, Chroma embeds the text automatically (unless you supply embeddings= yourself). Keep IDs unique and stable, and ensure the collection's embedding dimensionality matches the model you're using.

Upgrading to chunked ingestion

That single sentence indexed fine, so the plumbing works. Real pages need more work though. I split each page into chunks, a few sentences each, with some overlap at the edges so the model does not lose context mid-thought. The notebook has a helper I wrote, `chunk_documents`, that handles the splitting. It tags every chunk with an index number and a source ID. Handy when retrieval pulls the wrong chunk and you need to trace it back.

I am simplifying the indexer here to explain the main concept. In production, the indexer does a lot more than this. In this example, I want you to learn the core loop: text goes in, embeddings come out, ChromaDB stores them.

Now that we have explained the main concept of embedding and storing the chunks, we will be using a helper function I created in the notebook, `def vector_indexer`, to do this work for us. The following code snippet is the core code:

```
collection.add(
    documents=pending_docs,
    ids=pending_ids,
    metadatas=pending_metas if metadatas else None,
)
```

In the Notebook, the full function `vector_indexer` wraps this code with content-hash IDs for idempotency (same text always gets the same ID) and makes sure there are no duplicates, even when you re-run this function multiple times. After you run this function cell, you can call the `vector_indexer` function as shown in the following example, where I add two simple sentences, `First line` and `Second Line`, to the collection:

```
ids = vector_indexer(collection, ["First line", "Second line"], id_prefix="demo")
print(f"Indexed {len(ids)} items")
```

Now that the database, collection, and the helper function are in place, we're going to load a tiny dataset and index it properly. I'm keeping it simple on purpose: a few short documents, each with a stable ID, so you can see exactly how ChromaDB stores and reuses your data. The code adds documents only if they aren't already present (I've learned the hard way that idempotency matters), lets ChromaDB handle embeddings for us, and then verifies the final count so you know the index is ready for retrieval. You can run this code in the companion notebook (`chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb`), search for "`Our Raw Data`" to find the cell. After adding the documents to the documents list, we call the `vector_indexer` function and pass the entire list as shown in the following code snippet (Search for *3. The Indexing Process Example* in the Notebook):

```
written_ids = vector_indexer(
    collection=collection,
    texts=documents,
    id_prefix="doc"
)

print(f"Indexed {len(written_ids)} new items: {written_ids}")
```

What `vector_indexer` does is pretty mechanical. It hashes each document with a prefix to make a stable ID, something like doc_a3f29b. Then it checks whether that ID already sits in the collection. If it does, skip. If not, call `collection.add` and let ChromaDB run the embedding automatically. At the end you get back `written_ids`, just the ones that were new this run. I added that return value after a debugging session where I could not figure out why

counts were not going up; it turned out the documents were already there from a previous run.

You should see an output saying that it indexed 4 new items and their ids:

```
Indexed 4 new items: ['doc_e703d12f46044bda', 'doc_3881aea93a79aa81', 'doc_8a91f8af1bf32f5e',
'doc_7da6c838a91b28c5']
```

Now, let's verify how many items are in the index:

```
count = collection.count()
print(f"\nThe collection now contains {count} items.")
```

At this point in the notebook, you should see 7 items because we also indexed three sentences earlier: the manual test sentence and two demo sentences:

```
The collection now contains 7 items.
```

I built `vector_indexer` to save myself from re-indexing headaches. It takes a SHA-1 hash of each document, sticks a prefix on it, and that becomes the ID. Same text, same hash, every time. Before adding anything, it checks the collection first, and if the ID is already there, it moves on. Saves you from the duplicate nightmare I ran into during my first week with ChromaDB.

Now, let's peek at what's actually stored in our collection. You can run this code in the companion notebook (`chapter_03/Jupyter_Notebooks/Chapter_3_Notebook.ipynb`), search for `peek at what's actually stored`.

```
all_data = collection.get()
print(f"Document IDs: {all_data['ids']}")
print(f"A document preview: {all_data['documents'][3][:50]}...")
```

You should get something close to the following but with different IDs. In the Notebook, search for `Add Same Documents with Fixed IDs` to find the cell where I added the same four documents with `ids ['id_0', 'id_1', 'id_2', 'id_3']`. This is what I will show in the diagram later:

```
Document IDs: ['manual_1', 'demo_ce405c63b181a6ab', 'demo_b456a020d1eecda5', 'doc_e703d12f46044bda',
'doc_3881aea93a79aa81', 'doc_8a91f8af1bf32f5e', 'doc_7da6c838a91b28c5', 'id_0', 'id_1', 'id_2',
'id_3']
A document preview: The company's new AI policy, effective June 1st, r...
```

Perfect! Now let's visualize what we've built so far.

Here's *Figure 3.9* showing our ChromaDB structure with the indexed documents:

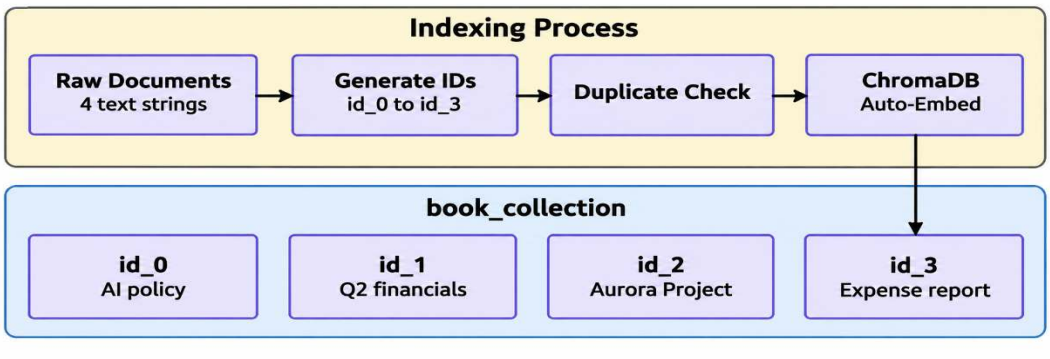


Figure 3.9: ChromaDB structure showing our indexed documents with their IDs and embeddings

And that's it. We've just built a simple indexing pipeline. We now have a persistent vector database containing the vectorized knowledge of our documents, ready and waiting for a user's query. In the next chapter, we will explain the full RAG pipeline, starting with the query pipeline.

We can index a sentence now. That works. But nobody hands you one sentence in production; they hand you a 200-page PDF. We need to chunk those documents into smaller pieces before embedding, and how you chop matters more than you would think.

Chunking

I have seen chunking cause more retrieval bugs than any other part of the pipeline. At Clearwater Analytics, Dan Siddall wrote a public article on question generation for LLM fine-tuning that describes five chunking methods: fixed-size, sentence-based, paragraph-based, semantic, and hierarchical. His point was that the method you pick changes the quality of downstream question generation and retrieval in ways that are hard to predict without testing. On one of my own projects, I tried three different embedding models to fix bad retrieval before someone looked at the chunks themselves. The actual problem was 2000-character chunks with no overlap splitting paragraphs mid-sentence. Once I fixed the boundaries, the original embedding model worked fine.

Chunks that are too big drown the answer in noise, chunks that are too small lose the surrounding story. I spent a while thinking this was a prompt

problem before realizing it was a chunking problem. Small-to-big retrieval is the pattern that fixed it for me. You embed something small, a couple hundred tokens, so the cosine match is sharp. Next to it in metadata you store the parent paragraph or section. Retriever grabs the small piece, system pulls the parent from metadata, LLM sees both. You evaluate whether any of this is working in *Chapter 9*.

What is metadata and why it matters

In practice, metadata is just helpful labels attached to each chunk so we know where it came from and how to use it. It enables filtering, grounding, parent-child navigation, analytics, and listing references for results.

Tiny example:

```
{
  "text": "Employees must complete security training annually.",
  "metadata": {
    "doc_id": "policy_2024_v3",
    "title": "Employee Security Policy",
    "chapter": "Compliance",
    "section": "Training Requirements",
    "page": 12,
    "chunk_id": "policy_2024_v3:12:3",
    "parent_id": "policy_2024_v3:section:training",
    "order": 3,
    "timestamp": "2024-05-01",
    "tags": ["security", "training", "annual"]
  }
}
```

With those fields in place, you can filter results to a single document or chapter, ground answers with title and page citations, expand context by loading the parent chunk through `parent_id`, and track which chunks get retrieved so you know where to improve your chunking.

The naive approach: fixed-size chunking

The simplest approach is to chop the document into pieces of a fixed character count, say 500 characters each. It is fast and easy to implement. But think about what happens when that 500-character boundary lands in the middle of a sentence. You end up with one chunk that says the key to successful AI implementation is a well-defined data strategy and the next chunk starts with which ensures that the model has access to clean structured information.

Neither chunk is useful on its own, and the retriever will probably miss both when someone asks about data strategy.

../../EPUB/NOTE

I am counting characters here because they are easy to see, though in practice most production chunkers count tokens instead, which we switch to later when we tune sizes.

You can run the code in the companion notebook: `Chapter_3_Notebook.ipynb`. Look for the subsection titled *The naive approach: fixed-size chunking* under the section titled *From a Sentence to a Document: The Need for Chunking*.

Let's see this in action with a simple code example:

```
def fixed_size_chunker(text: str, chunk_size: int, overlap: int):
    assert overlap < chunk_size, "overlap must be smaller than chunk_size"
    chunks = []
    for i in range(0, len(text), chunk_size - overlap):
        chunks.append(text[i:i + chunk_size])
    return chunks

long_text = (
    "Data strategy is fundamental to building powerful and
     reliable AI systems. Without a clear strategy, models may
     produce inaccurate or irrelevant results, leading to a poor
     user experience and a lack of trust in the system."
)

# Let's use a small chunk size to see the problem clearly
chunks = fixed_size_chunker(long_text, chunk_size=100, overlap=20)

print("Generated Chunks:")

for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: '{chunk}'")
```

This is the output we get:

```
Generated Chunks:
Chunk 1: 'Data strategy is fundamental to building powerful and reliable AI systems. Without a clear strategy,'
Chunk 2: 'ut a clear strategy, models may produce inaccurate or irrelevant results, leading to a poor user exp'
Chunk 3: 'g to a poor user experience and a lack of trust in the system.'
```

Figure 3.10: Fixed-Size Chunking Example Output

When you run this, you'll see exactly the problem I'm talking about, as shown in *Figure 3.10*. The chunks will likely end and begin in awkward, unnatural

places, breaking sentences and ideas apart. While this approach is better than nothing, it's far from optimal. It's a brute-force method that ignores the most important thing about our data: its meaning.

This is why we need to move beyond simple, arbitrary rules. First, we'll upgrade to a sentence-based approach with a small overlap to avoid mid-sentence splits. Then we'll move into **semantic chunking**, where we use the meaning of the text itself to guide how we break it down.

Sentence-Based chunking (practical bridge)

Before we bring in semantic methods, I like to adopt a simple, dependable bridge: split by sentence boundaries with a light overlap so ideas don't get cut off. It's fast, has zero external dependencies, and immediately improves retrieval quality.

Key idea: pack sentences up to a character budget, then start a new chunk with a one-sentence overlap:

```
import re

SENT_SPLIT = re.compile(r"(?<=[.!?])\s+")

def sentence_split(text: str):
    text = (text or "").strip()
    if not text:
        return []
    parts = [s.strip() for s in SENT_SPLIT.split(text) if s.strip()]
    return parts if parts else [text]

def chunk_sentences(text: str, max_chars: int = 1200, overlap_sentences: int = 1):
    chunks, buf, buf_len = [], [], 0
    for s in sentence_split(text):
        L = len(s) + 1
        if buf_len + L <= max_chars:
            buf.append(s); buf_len += L
        else:
            if buf: chunks.append(" ".join(buf).strip())
            overlap = (
                buf[-overlap_sentences:]
                if (overlap_sentences > 0 and buf)
                else []
            )
            buf = overlap + [s]
            buf_len = sum(len(x) + 1 for x in buf)
    if buf: chunks.append(" ".join(buf).strip())
    return chunks
```

```
# Example
chunk_texts = chunk_sentences(long_text, max_chars=1200, overlap_sentences=1)
```

What goes to the indexer now is `chunk_texts` along with per-chunk metadata (we'll add `chunk_index`, `chunk_count`, and `source_id` in the notebook helper). Tune `max_chars` to your model's context window; the default here works well for most demos.

Practical bridge: End-to-End mini pipeline (crawl to chunk to index)

Before we dive into advanced chunking, I want to connect the dots with a tiny, working pipeline, exactly what we run in the notebook. Search for "`crawl_site(BASE,`". We'll crawl a few pages from *rany.elhousieny.com* using `crawl_site`, convert them to sentence-based chunks using `chunk_documents`, and index those embedded chunks into `book_collection` (ChromaDB). This gives you the complete flow before we optimize chunking strategy. This is an advancement from *Chapter 1*, when we added the context directly to the prompt about `Rany ElHousieny` to improve the response from the LLM. Now, we will crawl the latest online data about `Rany ElHousieny` and go through the pipeline to store it in the ChromaDB for the next chapter to retrieve and add it to the prompt to improve the response from the LLM. *Figure 3.11* shows the flow of the pipeline.

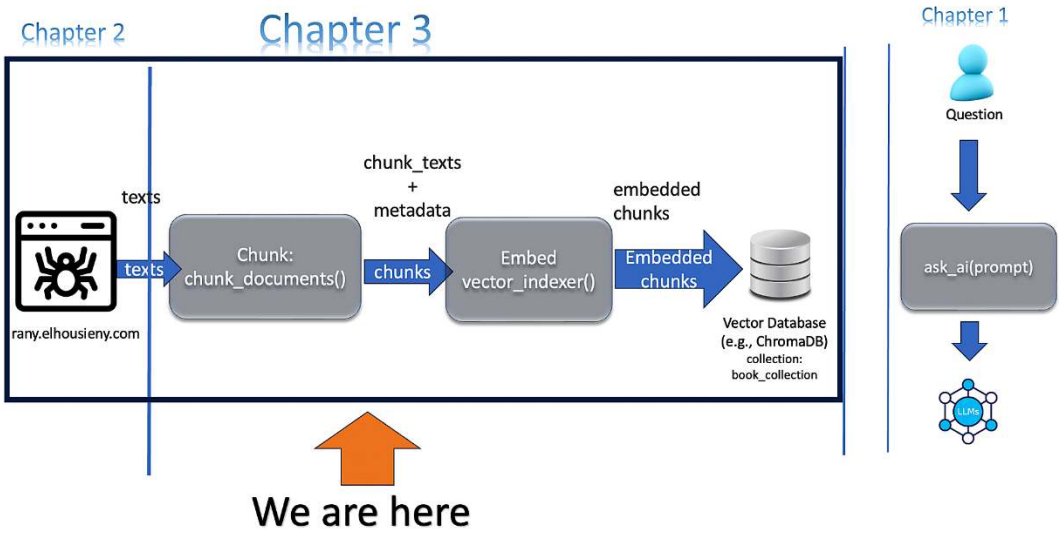


Figure 3.11: End-to-End Mini Pipeline (Crawl to Chunk to Index)

This figure shows the practical bridge we're building in *Chapter 3*. We crawl a few pages from `rany.elhousieny.com`, split each page into sentence-based chunks with light overlap via `chunk_documents()`, and index those embedded chunks into the ChromaDB collection `book_collection` using `vector_indexer()`. This prepares the knowledge base we'll retrieve from in the next chapter, while keeping the *Chapter 1* `ask_ai(prompt)` path separate. The three calls that run the full pipeline are in the companion notebook. Search for `crawl_site(BASE` to find the cell.

There is no magic inside crawl_site. It starts from the base URL and works through a queue of pages, remembering what it has already visited so it never fetches the same page twice. Each page gets fetched with a ten-second timeout, and any page that fails is skipped with a printed warning instead of killing the whole run. The HTML then goes through BeautifulSoup, which drops the script, style, and navigation tags before extracting the text. Pages with barely any text left, under about 100 characters, are treated as noise and dropped. For every page it keeps, the function records the URL, the page title, and the crawl date as metadata, then follows links that stay on the same domain until it reaches max_pages. That metadata is the part I care about most; it is what lets retrieval filter by source and date instead of relying on vector similarity alone.

Here are the steps for it:

1. Crawl pages from `rany.elhousieny.com`:

```
BASE = "https://rany.elhousieny.com"
texts, metadatas = crawl_site(BASE, max_pages=25)
print(f"Crawled {len(texts)} pages from {BASE}")
```

2. Chunk the crawled text into sentence-based chunks:

```
chunk_texts, chunk_metas, chunk_ids =
chunk_documents(
    texts, metadatas,
    max_chars=1200,
    overlap_sentences=1,
    id_prefix="rany"
)
```

3. Index the chunks into ChromaDB:

```
written_ids = vector_indexer(
    collection=collection,
    texts=chunk_texts,
    metadatas=chunk_metas,
    id_prefix="rany"
)
```

Optional quick peek. You can peek into what we stored from the crawled database as shown in the Notebook; when you search for `Quick peek`, you will see something similar to *Figure 3.12*:

```
First 3 crawled items:
  rany_80aeb4c2b084e4f2: Rany ElHousieny Rany ElHousieny , Ph.D (ABD) Senior Generati... |
  rany_199f1863752a27d1: Passionate about building scalable AI solutions and collabor... |
  rany_1523884f057fdc82: Key Achievements CTO-endorsed agentic AI methodology adopted... |
```

Figure 3.12: Quick Peek at what we stored from the crawl

Now, have a look at the following code:

```
data = collection.get(ids=written_ids[:3], include=["documents", "metadatas"])
    print(f"\nFirst 3 crawled items:")
    for doc_id, doc_text, meta in zip(
        data['ids'], data['documents'], data['metadatas']
    ):
        source = meta.get('source_url', 'N/A') if meta else 'N/A'
        print(f"  {doc_id}: {doc_text[:60]}... | source: {source}")
```

The previous snippet gets the first three chunks from the `written_ids` list, `written_ids[:3]`, that we created in step 3, then prints them. Those are just small code snippets to show you the flow of the pipeline. You can run them in the companion notebook: [`Chapter_3_Notebook.ipynb`]. This is the pipeline we will use in the next chapter to retrieve the knowledge base we prepared here and add it to the prompt to improve the response from the LLM.

Optional: You can swap our simple sentence splitter for the widely used LangChain splitter. It gives you token-aware sizes, regex separators, and start indices while keeping the same pipeline shape. Here is the code:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Token-aware variant sized in tokens (good for prompt budgets)
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=800,           # tokens
    chunk_overlap=100,
    separators=["\n\n", "\n", ". ", " ", ""],
```

```
    keep_separator=True,
    add_start_index=True
)

chunk_texts = text_splitter.split_text(long_text)
```

With that end-to-end path working, it's the right moment to improve the quality of what we store. The sentence-based bridge avoids mid-sentence cuts, but it still ignores meaning. In the next section, we'll move to semantic chunking so our boundaries follow ideas, not just character counts, leading to cleaner retrieval and better answers.

A smarter approach: semantic chunking

You take each sentence, embed it with the same model from earlier in the chapter, and check cosine similarity between neighbors. High similarity means same topic, keep them together. If similarity drops, that is your cut point. The approach is called semantic chunking, basically the same math we already have but pointed at chunking instead of retrieval.

I ran into the limits of sentence-based chunking pretty quickly when I started building real pipelines. Two paragraphs about completely different topics would land in the same chunk just because neither one hits the character limit. That is what made me look at using embeddings for the split decisions.

Let's look at a conceptual code example next. For this, we'll use the sentence-transformers library, which is fantastic for this kind of sentence-level analysis. You can run the code in the companion notebook. Look for the subsection titled *A Smarter Approach: Semantic Chunking* under the section titled *From a Sentence to a Document: The Need for Chunking*. Note: the model will download on first run; this may take a minute.

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("all-MiniLM-L6-v2")

long_text = (
    "Retrieval augmented generation feeds external documents "
    "into the LLM context window. "
    "The retriever finds the most relevant chunks "
    "and passes them to the model as context. "
    "My favorite pizza topping is mushrooms with extra cheese. "
    "I also enjoy trying new pizza places whenever I travel to a new city. "
    "Chunking splits long documents into smaller pieces for the vector store. "
```

```
    "Good chunk boundaries preserve the meaning of each passage."
)

# 1) Sentence split
sentences = nltk.sent_tokenize(long_text)

# 2) Embeddings
embeddings = model.encode(sentences)

# 3) Adjacent similarities
print("Sentences:")
for i, s in enumerate(sentences, 1):
    print(f"  {i}. {s}")

print("\nSimilarity between adjacent sentences:")
for i in range(len(sentences) - 1):
    score = util.cos_sim(embeddings[i], embeddings[i + 1]).item()
    marker = " <-- topic shift!" if score < 0.15 else ""
    print(f"  Sentence {i+1} -> {i+2}: {score:.4f}{marker}")
```

When you run this code snippet, you'll see a similarity score between each pair of sentences, as shown in *Figure 3.13*. Notice how the sentences that discuss the same core idea have higher similarity scores. This is the information we can use to build semantically coherent chunks. The result is a knowledge base that provides our RAG system with clean, contextually complete information, which is the key to getting high-quality answers. As you can see, Sentences 1-2: RAG retrieval (should score high). Sentences 3-4: Pizza (should score high with each other, LOW with sentence 2). Sentences 5-6: Chunking (should score high with each other, LOW with sentence 4).

```
Sentences:
  1. Retrieval augmented generation feeds external documents into the LLM context window.
  2. The retriever finds the most relevant chunks and passes them to the model as context.
  3. My favorite pizza topping is mushrooms with extra cheese.
  4. I also enjoy trying new pizza places whenever I travel to a new city.
  5. Chunking splits long documents into smaller pieces for the vector store.
  6. Good chunk boundaries preserve the meaning of each passage.

Similarity between adjacent sentences:
  Sentence 1 -> 2: 0.4524
  Sentence 2 -> 3: 0.0410 <-- topic shift!
  Sentence 3 -> 4: 0.5550
  Sentence 4 -> 5: -0.0198 <-- topic shift!
  Sentence 5 -> 6: 0.4760
```

Figure 3.13: Output of Similarity Test

That covers the embedding-based approach to chunking. It works well when the text has clear topic boundaries, but not every document cooperates.

LLM-driven chunking: the agentic approach

Semantic chunking with similarity thresholds works well for most content, but we can push it further. An emerging approach, increasingly used in production pilots, comes in: **Agentic Chunking**.

This is actually a shift in thinking. Instead of just using an embedding model to measure the distance between sentences, I use a powerful Large Language Model (LLM) (the same kind of model that powers our AI assistant) to actually decide where the most logical breakpoints in the text are.

The idea is simple: hand the text to an LLM and let it decide where the breaks should go. You give it a prompt like: read this document and mark where the topic changes. It costs more per chunk and the output is less deterministic than a threshold, but for messy documents where the structure is all over the place, it finds boundaries I would not have thought of myself.

If the output is unsuitable, I reject it and fall back to the cosine splitter. I also pin the model version because a quiet provider update once changed every chunk boundary I had in production.

I keep guardrails tight when I use it: max and min chunk sizes, a small overlap, and temperature set to zero so the output stays consistent across runs.

Works best on the messy stuff, policy documents, long FAQs, anything where the format keeps changing and a fixed threshold just guesses.

Example prompt I use:

```
Read this text. Insert the delimiter "|||" at natural topic shifts. Keep chunks under ~900 tokens,
allow ~20% overlap, and avoid splitting mid-sentence.
```

All four chunking methods we just covered share the same blind spot, and it matters no matter which one you pick.

The final layer: hierarchical and context-aware chunking

Semantic chunking handles meaning, but it treats every chunk the same. A paragraph from the introduction and a paragraph from the technical appendix get the same flat representation in the vector store. For structured documents like manuals or contracts, that matters, or even this book, because where something was said changes what it means.

Think about it. A sentence's meaning can change dramatically based on its context. A statement in the 'Introduction' section might be a high-level summary, while the exact same statement in the 'Technical Deep Dive' section has a much more specific, detailed meaning. If we just create a flat list of semantic chunks, we lose all of that valuable structural context.

This is where **hierarchical chunking** comes in. It's less about a new way to cut up text and more about enriching our existing semantic chunks with **metadata**. For every chunk we create, we also store information about where it came from.

For example, a chunk's metadata might look like this:

```
{
  "text": "The core of the system uses a text-embedding-3-small model...",
  "chunk_id": "ai_arch_v2_ch4_s2_1_001",
  "parent_id": "ai_arch_v2_ch4_s2_1",
  "source_document": "ai_system_architecture_v2.pdf",
  "chapter": "Chapter 4: Core Components",
  "section": "4.2.1 - Embedding Model Selection",
  "page_number": 42
}
```

Once you have that metadata attached, two things get easier: filtering at retrieval time and citing sources in the response.

For our **Enterprise Knowledge Assistant**, when an employee asks about company policy, the system can prioritize answers from chunks with metadata indicating they came from the official 'HR Policy Manual' over chunks from an informal team wiki.

For our **Domain-Specific Assistant**, a lawyer can ask for information and the system can weigh evidence from a 'Majority Opinion' section of a court document more heavily than from the 'Dissenting Opinion'.

For our **Personal Productivity Assistant**, when you ask about a specific project, it can surface relevant chunks from meeting notes, emails, and documents while maintaining the chronological context through timestamps and parent relationships.

Metadata captures structure inside a document, but it stops at the document boundary. What happens when a person shows up in one file and the project they lead shows up in another? The metadata has no way to link them. For

those cross-document connections you need something built for relationships, and that is where graph databases come in.

Graph database integration: connecting the dots

So far, we've done a fantastic job of breaking our documents down into meaningful, context-aware chunks. Our vector database is great at finding the most semantically relevant chunks for a given query. But I've learned from building complex systems that sometimes, the most important piece of information isn't inside a chunk of text. It's the relationship between chunks.

Graph databases store things differently. Instead of vectors in a space you get nodes and edges on a whiteboard. As shown in *Figure 3.14*, Rany EL is a node. Aurora Project is a node. The LEADS relationship between them is an edge, typed and directional. When someone asks who leads Aurora you traverse one edge, done. I started using Neo4j on a project where the retriever kept returning the right documents but could not connect them, engineer name in one page, project assignment in another, both in the top five results with no link between them. Adding maybe forty lines of Cypher queries to the pipeline fixed questions that no amount of embedding tuning was going to fix.

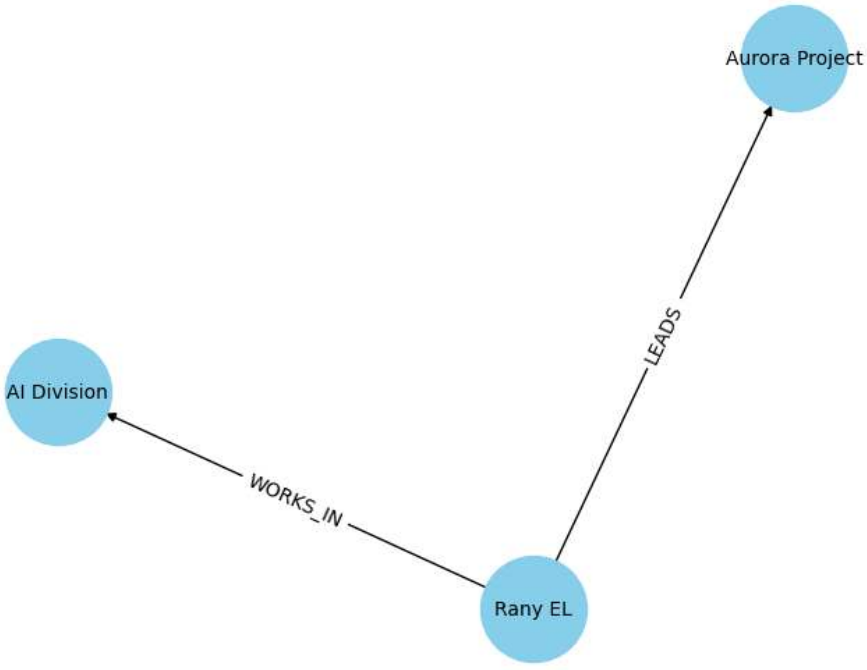


Figure 3.14: Nodes and directed edges in a simple graph

What is a graph database?

Simplest way to think about it: sticky notes on a whiteboard with labeled string between them. Each sticky note is a node. Could be a person, a project, a Confluence page, or one of our semantic chunks. The string is a relationship, and you write a label on it like `WORKS_FOR` or `MENTIONS` or `IS_PART_OF`, with an arrow showing which direction it goes. That is the whole data model: nodes and labeled directional edges.

The reason this matters for our AI assistant is that you get two things at query time instead of one. You still have the content inside each chunk, that is the what. But now you also have edges telling you how chunks relate, that is the how. An assistant that knows both can answer questions that pure vector similarity cannot even attempt.

To visualize how these relationships work in practice, let's look at a concrete example in *Figure 3.15*:

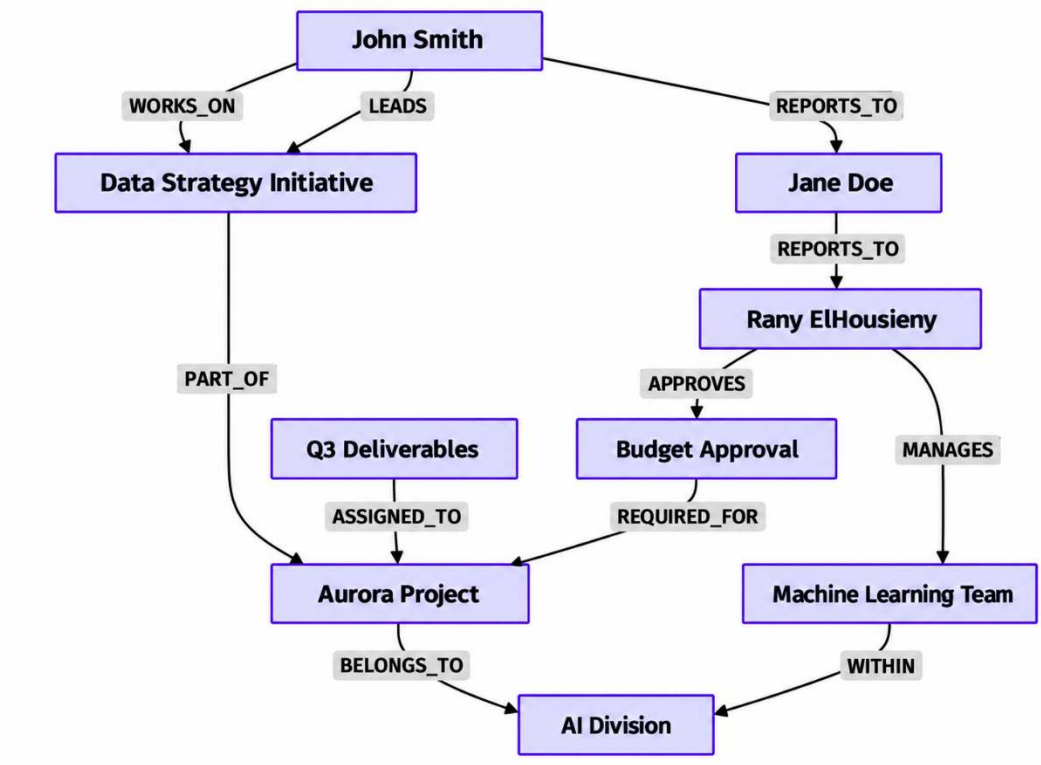


Figure 3.15: Graph Database Entity Relationships: Connecting Documents, People, and Projects

Figure 3.15 is worth studying for a minute. John Smith connects to Aurora Project through LEADS and PART_OF, by way of the Data Strategy Initiative,

and both Aurora Project and Machine Learning Team feed into AI Division through BELONGS_TO and WITHIN. The query I keep coming back to is find all documents related to projects that John Smith worked on. Pure vector search cannot do that because it has no idea what worked on means structurally. The graph just follows the edges.

For our **Content Moderation Assistant**, this is crucial. It can go beyond just flagging a toxic comment (a chunk). It can trace the relationships to see if this user has a history of toxic behavior, if they belong to a coordinated group, or if they are targeting another specific user. This is a level of contextual understanding that a simple vector search can't provide.

Let's take two chunks from *Figure 3.15*. Chunk 1: Rany ElHousieny manages the Machine Learning Team. Chunk 2: The Machine Learning Team is within the AI Division. In a simple vector search, these two chunks are related by the phrase Machine Learning Team. But in a graph database, we model it explicitly: a Person node for Rany ElHousieny, an Organization node for Machine Learning Team, a Division node for AI Division, and two typed edges connecting them. Rany ElHousieny manages the Machine Learning Team, which is within the AI Division. Two hops, and the graph has the answer.

I have never shipped a production RAG system that used only one of these. Every single project ended up with vector search handling the what-does-this-document-say questions and a graph handling the who-is-connected-to-what questions. The code for merging their results is maybe twenty lines, and it is the twenty lines that made the biggest difference. Setting up and querying a production graph database like Neo4j is an implementation topic and outside the scope of this book. Our focus is the data strategy side: how to shape your data with chunking, metadata, and relationships so that when you do connect a graph database, the data is ready for it.

Wiring the two together into a single retrieval path comes in the next chapter. First, though, both the vector index and the graph have to survive a jump in scale, which is where we go next.

Vector database optimization: from prototype to production

At this point you have enough to build a working RAG prototype. Most teams do, and most teams get stuck right here. A system that handles a dozen documents falls apart at a million, and the fixes are not obvious until you understand what is happening under the hood.

Something that confused me early on is why two vector databases with the same data give different recall numbers. Recall here means out of the true closest matches, how many did the search actually return. The answer lives in the search algorithm. Exact search checks every vector, which is fine for demos but falls apart past a few hundred thousand records. Production databases use Approximate Nearest Neighbor search instead, ANN for short, and the algorithm you will run into most often is HNSW, Hierarchical Navigable Small World. Qdrant uses it, Weaviate uses it as its default. Pinecone does not; it has never used HNSW. It runs its own proprietary algorithms and picks one for each slab of data automatically, based on slab size. HNSW builds a multi-layer graph of neighbors and hops through it at query time, fast but memory-hungry because the whole graph lives in RAM.

That trade-off is the real story behind database selection, worth understanding before you pick a vendor off a benchmark blog post. HNSW when you can afford the memory, PQ when you cannot.

When the index gets too big for RAM, a different algorithm kicks in. Product Quantization, PQ for short, compresses each vector into a compact code. Milvus offers it through its PQ index variants, IVF_PQ for example; there is no single default there, and its AUTOINDEX option picks an index to match your data. You trade memory for recall, roughly an order of magnitude less RAM but the compressed codes lose precision on close matches.

Next up is the tuning that separates a demo from something you can actually deploy: chunk sizes, overlap settings, metadata, and a pipeline to tie it all together.

The Goldilocks problem: optimizing chunk size and overlap

One of the first and most impactful levers you can pull is the size of your chunks. It's a classic Goldilocks problem: if your chunks are too small, you lose the surrounding context, and the AI might not have enough information to provide a good answer. If your chunks are too large, you introduce too much

noise, and the specific answer gets diluted. You're looking for that 'just right' size.

There is no magic number. When I went looking for one, I found that the teams who actually ran the experiments disagreed with each other. LlamaIndex swept chunk sizes from 128 to 2,048 tokens and watched faithfulness and relevancy; both peaked at 1,024, but the whole experiment ran on a single SEC filing, so I would not generalize from it. Pinecone and Milvus land lower, around 256 to 512, and Chroma found that even 200 tokens held up on precision, which honestly surprised me. A lot of that clustering near 512 is mechanical; the encoder models this got built on stopped reading after 512 tokens, and that hard ceiling is what the newer long-context embeddings are just now starting to push past.

What I keep coming back to in my own projects is the 256 to 512 band. Big enough to carry one full idea, tight enough that the idea stays its own thing. Short factual lookups want the low end. Anything where the reader needs the surrounding passage to follow along wants the high end.

But we can't just set a size and forget it. We also need to think about **overlap**. This is where we intentionally include a small amount of text from the end of the previous chunk at the beginning of the next one. Why? It helps ensure that a single, coherent idea that happens to fall across a chunk boundary isn't completely lost. It provides a little bit of semantic glue between your chunks.

A good starting point for overlap is **around 20% of your chunk size**. For a 500-token chunk, that would be a 100-token overlap. The notebook has a working version of this chunker you can run and modify. Search for `optimized_chunker` or `Goldilocks` in the *Chapter 3* notebook.

It's important to remember that these numbers are a starting point, not a universal rule. For our **Domain-Specific Assistant** working with dense legal documents, we might find that larger chunks (800-1000 tokens) perform better. For technical documentation, smaller, more focused chunks might be superior. The key is to start with a proven baseline and then test and measure what works best for your specific data and use case. This iterative process of tuning is a core part of building a high-performance RAG system.

Beyond the text: metadata enrichment

Here is something that bit me. I had two chunks saying different things about PTO days, 20 versus 25, and the system treated them as equals. It had no way to know one came from a draft from 2021, and the other was the published HR handbook. That is the problem metadata solves: you tag each chunk with source, status, and date so the retriever can tell the difference.

Look at the two chunks below.

Chunk A: "Employees are entitled to 20 days of paid time off per year."

Chunk B: "Employees are entitled to 25 days of paid time off per year."

Without metadata, the system has no way to know which is correct. But what if we add metadata?

Chunk A Metadata: `{ "source": "draft_policy_ideas_2021.docx", "status": "draft" }`

Chunk B Metadata: `{ "source": "Official_HR_Handbook_v4_2025.pdf", "status": "published", "effective_date": "2025-01-01" }`

Now, the choice is obvious. We can configure our retrieval system to heavily favor chunks from official, recent, published documents. This is a simple example, but it illustrates a profound point: metadata gives our system the context it needs to reason about the quality and relevance of information.

For our **Enterprise Knowledge Assistant**, this is non-negotiable. It needs to be able to distinguish between a final, approved document and an early draft. For our **Personal Productivity Assistant**, it could use metadata like `creation_date` to surface more recent notes first.

Data ingestion and indexing pipeline: getting it all together

Here is what we have done so far. As shown in *Figure 3.16*, we start with data collection from multiple sources (*Chapter 2*). In this chapter, we then chunk that content, generate embeddings, and store those embeddings and metadata in a vector database. In production, this is typically handled by an ingestion/indexing service that runs on a schedule or as a background job. If you prefer a managed option, services like Amazon Bedrock Knowledge Bases provide connectors, chunking, embeddings, and synchronization (often backed by OpenSearch Serverless for vector search).

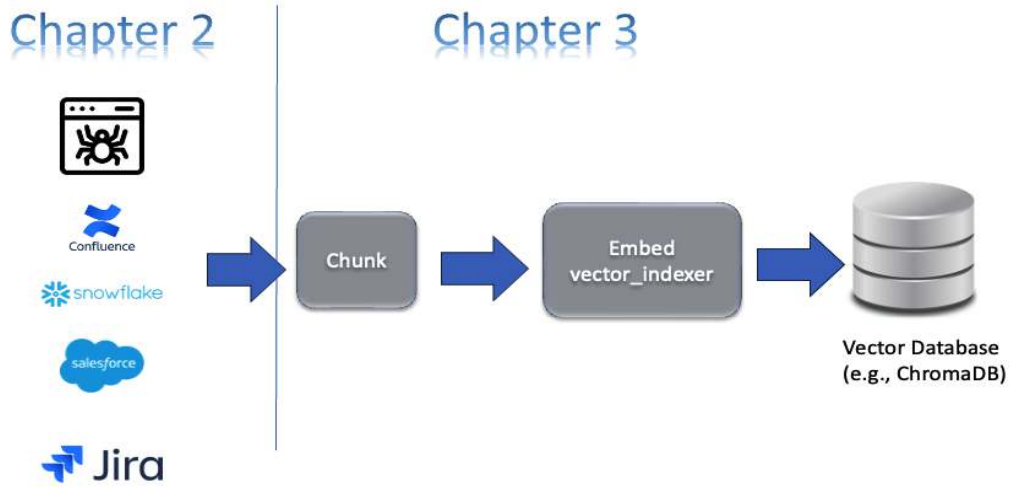


Figure 3.16: The Data Ingestion and Indexing Pipeline

Putting theory into practice

I keep a comparison table at *Table 3.1* for this: four assistants, four rows, and the columns are embedding model, chunking method, key metadata fields, and anything unusual.

Assistant	Embedding Model	Chunking	Key Metadata	Notes
Personal Productivity	text-embedding-3-small	Semantic (low threshold)	`source_type`, `source_app`, `creation_date`, `participants`	Cost and speed matter more than nuance for personal notes
Domain-Specific (Legal/Science)	text-embedding-3-large	Hierarchical (by document structure)	`document_id`, `section_id`, `section_title`, `document_type`	Precision is non-negotiable; subtle wording changes meaning

Assistant	Embedding Model	Chunking	Key Metadata	Notes
Enterprise Knowledge	text-embedding-3-small (hybrid: large for high-value docs)	Automated pipeline by doc type	`access_control_list`, `source_system`, `space_or_site`, `sensitivity_level`	Filter by ACL first, search second
Content Moderation	text-embedding-3-small	Per-post (each comment is a chunk)	`user_id`, `timestamp`, `flagged_status`	Graph layer: user and post nodes with POSTED/MENTIONS edges

Table 3.1: Data preparation strategy for each of the four AI assistants

The one row that looks different from the rest is Content Moderation. It is the only assistant that needs the graph layer because the real question is not what this post says, it is who keeps posting it and who are they targeting. Without the graph you catch individual comments but miss the pattern across accounts.

Summary

This chapter covered a lot of ground, so here is what to take away. Embeddings turn text into numbers, and the small model is good enough for most use cases. Chunking matters more than people think; the five methods we walked through go from naive to agentic and each one fits a different kind of document.

Metadata is the part most teams skip and then regret. Tag each chunk with source, date, and access level or you will end up serving answers from old drafts. Graph relationships handle the rest, the cases where the answer is not in any single chunk but in the connection between two things.

I have rewritten ingestion pipelines three times on the same project and each time the retrieval got better without touching the model or the prompt. That pattern keeps showing up. The preparation is where the leverage is.

In the next chapter we take this prepared data and build the retrieval layer on top of it. That is where the user's question meets the index and you find out whether your preparation was good enough.

Further reading and references

These are the articles and papers I kept going back to while writing this chapter:

- **Mastering the Art of Natural Language Processing: A Key to AI's Future**: To learn more about NLP, I highly recommend my article "*Mastering the Art of Natural Language Processing: A Key to AI's Future*". https://levelup.gitconnected.com/mastering-the-art-of-natural-language-processing-a-key-to-ais-future-43e2d6434a82
- **Understanding Word Embedding in NLP using Sentence Transformers**: To learn more about word embeddings, I highly recommend my article Understanding Word Embedding in NLP using Sentence Transformers https://www.linkedin.com/pulse/understanding-word-embedding-nlp-using-sentence-elhousieny-phd
- **Evolution of Word Embeddings: A Journey Through NLP History**: To learn more about word embeddings, I highly recommend my article Evolution of Word Embeddings: A Journey Through NLP History https://www.linkedin.com/pulse/evolution-word-embeddings-journey-through-nlp-history-rany-pzmjc/
- **The Evolution of AI: From Symbolic to Neural**: For foundational AI history that motivates neural-based data preparation, I recommend my article The Evolution of AI: From Symbolic to Neural https://levelup.gitconnected.com/the-evolution-of-ai-from-symbolic-to-neural-3fbe384ccaf0
- **The Rise of Transformers: A Revolution in Natural Language Processing (NLP) and AI**: To understand the transformer architecture as the foundation of modern embeddings, I recommend my article The Rise of Transformers https://www.linkedin.com/pulse/rise-transformers-revolution-natural-language-nlp-ai-rany-t01wc/
- **Unveiling the Linguistic Pillars of Natural Language Processing**: A Beginner's Guide: For the linguistic structure that informs chunking decisions, I recommend my article Unveiling the Linguistic Pillars of

Natural Language Processing https://levelup.gitconnected.com/unveiling-the-linguistic-pillars-of-natural-language-processing-a-beginners-guide-de4c1fa55c8b

- **Pioneering AI Innovation with Transformers and Beyond**: For Clearwater Analytics' AI direction with transformer-based foundations, see Pioneering AI Innovation with Transformers and Beyond https://medium.com/cwan-engineering/pioneering-ai-innovation-with-transformers-and-beyond-5f4cfe8a829f
- **Document Similarity Examples in Python**: To learn more about document similarity, I highly recommend my article Document Similarity Examples in Python https://www.linkedin.com/pulse/document-similarity-examples-python-rany
- **Retrieval-Augmented Generation (RAG) with LangChain**: To learn more about RAG, I highly recommend my article **Retrieval-Augmented Generation** (**RAG**) with LangChain https://www.linkedin.com/pulse/retrieval-augmented-generation-rag-langchain-refining-rany-adgmc
- **Customizing Generative AI Solutions with RAG**: At Clearwater Analytics, we developed a pipeline in which new RAG sources became operational in under an hour, as Cherry (2024) explained in his article. https://medium.com/cwan-engineering/customizing-generative-ai-solutions-with-retrieval-augmented-generation-86121761b8e3
- **Generating Diverse Questions for LLM Instruction Fine-Tuning**: Siddall (2025) compared five chunking methods at Clearwater Analytics, finding that the choice of method shapes retrieval quality more than chunk size alone. https://medium.com/cwan-engineering/generating-diverse-questions-for-llm-instruction-fine-tuning-strategies-and-considerations-73270f2b0adb
- **On Chunking Strategies**: For a comprehensive overview of modern chunking techniques, including detailed code examples, the article "*8 Types of Chunking for RAG Systems*" by Analytics Vidhya is an excellent starting point. https://www.analyticsvidhya.com/blog/2025/02/types-of-chunking-for-rag-systems/

- **On Vector Databases**: DataCamp provides a great, up-to-date comparison of the top vector databases in their article, "*7 Best Vector Databases for AI in 2026*". https://www.datacamp.com/blog/the-top-5-vector-databases
- **On Graph Integration**: For the most current best practices on combining vector search with graph databases, the official Neo4j Vector Search Developer Guide is the authoritative resource. https://neo4j.com/developer/genai-ecosystem/vector-search/
- **On Chunk Size**: LlamaIndex evaluated chunk sizes from 128 to 2,048 tokens and found 1,024 hit the best balance of faithfulness and relevancy. https://www.llamaindex.ai/blog/evaluating-the-ideal-chunk-size-for-a-rag-system-using-llamaindex-6207e5d3fec5
- **On Chunk Size**: Milvus recommends starting at 256 to 512 tokens for most RAG applications. https://milvus.io/ai-quick-reference/what-is-the-optimal-chunk-size-for-rag-applications
- On Chunk Size: **Chroma's technical report Evaluating Chunking Strategies for Retrieval** (2024) found that 200-token chunks delivered the highest precision in their retrieval benchmarks. https://www.trychroma.com/research/evaluating-chunking
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). **Efficient Estimation of Word Representations in Vector Space**. arXiv:1301.3781.
- OpenAI. **Vector Embeddings Guide**. https://platform.openai.com/docs/guides/embeddings.
- OpenAI. **Embeddings API Reference**. https://platform.openai.com/docs/api-reference/embeddings.

Part 2

Delivering Knowledge at Inference Time

This part of the book puts your prepared data to work. You will build a retrieval pipeline that finds relevant information and supplies it to the model as context. You will then examine the failures that emerge as sources grow and change, including outdated documents outranking current guidance and answers requiring information from multiple systems. Through these examples, you will learn how metadata and source characteristics influence retrieval design. By the end of this part, you will be able to build a working retrieval layer and select improvements based on the data problems your application encounters.

This part of the book includes the following chapters:

- *Chapter 4*, *Building Your First RAG System: From Theory to Implementation*
- *Chapter 5*, *Data Strategy for Advanced RAG: When Your Data Tells You What You Need*

4

Building Your First RAG System: From Theory to Implementation

Let's review what we have done so far. In *Chapter 1*, we built a basic AI assistant, `ask_ai(prompt)`, that takes a prompt query and sends it as is to the LLM. This is the right part of *Figure 4.1*. At the end of *Chapter 1*, we learned about in-context learning, and we gave the prompt a small boost with context. We explained that this is the foundation of our RAG system, adding context to the prompt. In *Chapter 2*, we showed how to collect different types of raw data that can be used later as context, as shown in the left part of *Figure 4.1*. In the previous chapter, we did the essential, foundational work of preparing our raw data. We took raw text, transformed it into a structured vectorized knowledge base, and stored it in a vector database. We created the `vector_indexer()` function to vectorize and store our data in a vector database, as you can see in the second section of *Figure 4.1*. Now, it's time to put that data to work. This is the chapter where we build the engine. We're going to complete our first end-to-end **Retrieval-Augmented Generation** (**RAG**) system by building the retrieval and generation components. We will start by building the retrieval function from raw Python, line by line, and test it with queries that show both successful retrieval and honest failure when the answer is not in the data. From there we cover the pitfalls I see in first-time builds, when it makes sense to graduate to a framework like LangChain, and how the managed RAG options and vector databases compare when you would rather not wire everything yourself.

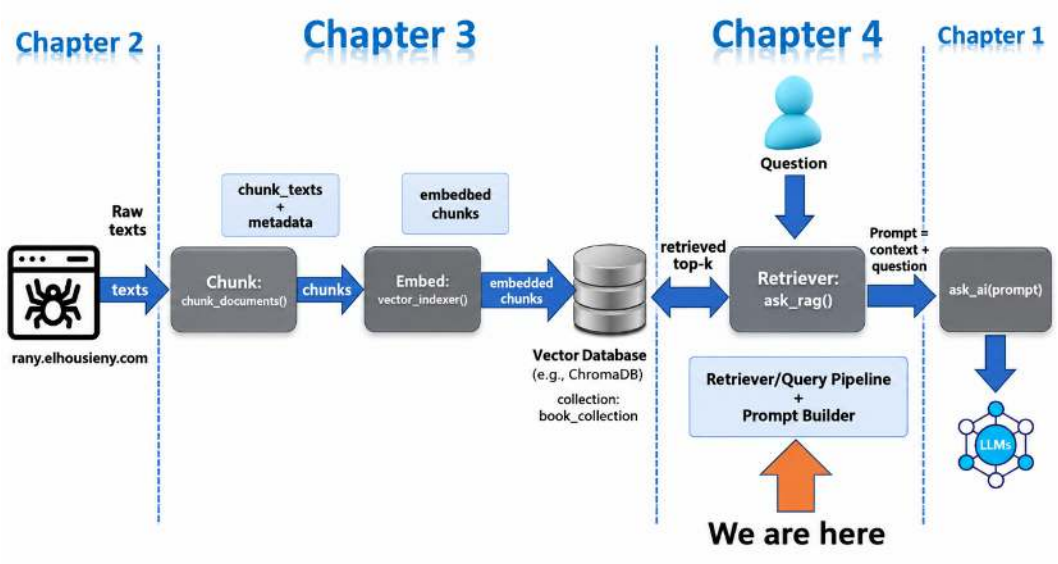


Figure 4.1: The Full RAG Pipeline

By the end of this chapter, you will have built a basic functional RAG pipeline. You'll understand every single component, from the moment a user asks a question to the moment our AI generates a context-aware answer. The reason we're going to build it from scratch, using Python and the core libraries directly, is because I've learned that you can't truly master a system until you've built it yourself. Frameworks like LangChain are incredibly powerful, but they hide a lot of the magic. I want you to see the magic, understand it, and be able to control it. We will build the basic RAG system from scratch in the `ask_rag()` function, as shown in the right part of *Figure 4.1*. This basic RAG system is sometimes called the "naive" or "vanilla" RAG system, as it doesn't use any advanced techniques to improve retrieval or generation. It's a simple but effective way to get started with RAG. In *Chapter 5*, we will examine what breaks when real data hits this system, and the data strategies that fix each failure.

In this chapter, we cover the following topics:

- The two halves of a RAG system: indexing and retrieval
- The retrieval pipeline: finding and using knowledge
- Common pitfalls in first-time RAG builds
- Graduating to a framework
- Managed RAG and enterprise search options

- Choosing a vector database (Top 5)
- Applying RAG to the four case studies

The two halves of a RAG system: indexing and retrieval

I think of RAG as two halves that barely meet. One half runs offline and builds the knowledge base. The other runs online and answers questions as they come in. They touch at exactly one place, the vector index, where the first half writes and the second half reads. Each half runs as its own pipeline:

- **The Indexing Pipeline**: This is the offline process where we prepare our knowledge. This is what we have done in *Chapter 3*. This process involves loading our documents, chunking them, creating vector embeddings for each chunk, and storing them in our vector database.
- **The Retrieval Pipeline (The Research Assistant)**: This is the online, real-time process that happens when a user asks a question. Our AI acts as a brilliant research assistant. When a query comes in, the assistant first goes to the card catalog (our vector index) to find the most relevant books and pages (our document chunks). Then, it takes those pages, reads them, and uses that information, and only that information, to write a clear, concise answer to the user's question.

I have found that splitting is the whole point. You index once, then retrieve against that index a million times, which is why the heavy preparation in *Chapter 3* only had to happen the one time. From here on, I will stay on the Retrieval side and walk it piece by piece, picking up where *Chapter 3* left off.

The retrieval Pipeline: finding and using knowledge

What I like about Retrieval is how short the code is. A few function calls take a question, pull the closest chunks out of the vector index, glue them onto the prompt, and send it all to the LLM. The rest of this section walks through that path, line by line.

Coding our RAG function

The full `ask_rag` function is shown ahead. It is short, so rather than explain it twice, I put a comment on each step inside the code. Read it top to bottom, and the comments tell you which part of the pipeline each block belongs to.

```
def ask_rag(
    query: str, db_path="data/chroma_db",
    collection_name="chapter4_collection"
):p
       and generate a grounded answer."""
    # 1. Connect to the Chroma DB built in Chapter 3
    # (reuse the client if it exists)
    if 'chroma_client' not in globals():
        chroma_client = chromadb.PersistentClient(path=db_path)
    else:
        chroma_client = globals()['chroma_client']
    collection = chroma_client.get_collection(name=collection_name)

    # 2. Vector-search the index for the 2 closest chunks to the user's query
    results = collection.query(query_texts=[query], n_results=2)
    retrieved_documents = results['documents'][0]
    context = "\n\n".join(retrieved_documents)
    # Debug: see what the LLM will receive:
    print(f"DEBUG: Retrieved context:\n{context}\n")

    # 3. Build the prompt: role, retrieved context wrapped in tags,
    #    and the user's question
    prompt = f"""
    You are an expert assistant.
      Use the following retrieved context to answer the user's question.
    If the answer is not in the context,
      state that you cannot find the information.
    Do not use any other information.

    <context>
    {context}
    </context>

    <question>
    {query}
    </question>

    Answer:
    """
    # 4. Hand the prompt to Chapter 1's ask_ai() and return the grounded answer
    return ask_ai(prompt)
```

Reading `ask_rag` from top to bottom, here is what happens:

1. It opens the Chroma database we built back in *Chapter 3*.

2. A vector search runs for whatever the user asked.
3. Only the two closest chunks are kept, nothing more.
4. Those two get glued into a single context block.
5. The prompt then sets the model's role and wraps the context and the question in tags.
6. The whole thing goes to `ask_ai`, and the model answers.

The retrieved-context print in the middle is just a debug aid, so you can see what the model receives before it answers.

Now we're testing our complete RAG system with two strategic questions. The first query asks about the AI policy, which should find the relevant document and provide a direct answer. The second asks about Q1 financial results, which should demonstrate the system's ability to recognize when it doesn't have the information - our database only contains Q2 results. This is shown here:

> **../../EPUB/NOTE**
>
> To use the *Chapter 3* mini-pipeline index, call: `ask_rag(..., db_path="data/chroma_db", collection_name="book_collection")`. Start with a small number of retrieved chunks, setting `n_results` to 2-5. Later, add thresholding, reranking, or diversification before building the prompt.

```
QUERY: 'What is the new AI policy?'

Results found: 2 documents (similarity: 0.58, 0.52)

GENERATED ANSWER:

The new AI policy requires all employees to complete a mandatory training course and is effective
June 1st.
```

A quick word about that similarity number, because it will save you a debugging session later. Chroma's default distance metric is L2, and 1 minus an L2 distance is not a similarity of any kind. The notebook creates this demo collection with `metadata={"hnsw:space": "cosine"}`, and in cosine space the math means what the label says: similarity is 1 minus distance. When you query a collection that kept the default, print the raw distance instead and remember lower means closer, which is exactly what we will do with the *Chapter 3* index later in this chapter.

That first query landed cleanly. The second one is the more interesting test, because the answer it asks for was never put in the index:

```
QUERY: 'What were the Q1 financial results?'

Results found: 2 documents (closest match: the Q2 financial results document)

GENERATED ANSWER:

I cannot find the information about the Q1 financial results in the provided context.
```

The first query lands on the AI policy doc and returns a direct answer. The second is more interesting. We ask about Q1 financials, but only Q2 data is indexed. The system has to admit it doesn't know rather than make something up, and it does, exactly because the prompt told it to.

Here's a visual representation of how both queries flow through our RAG system:

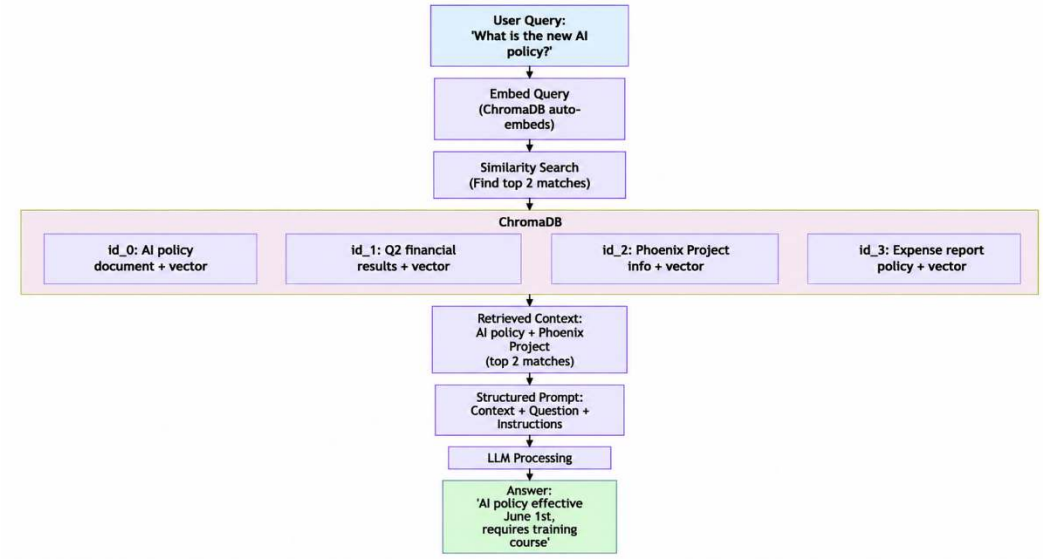


Figure 4.2: RAG query process showing how user questions flow through embedding, similarity search, context retrieval, and LLM generation

Understanding the Q1/Q2 Example

The second query is the one worth paying attention to. We ask for Q1 financial results, and ChromaDB comes back with the Q2 financial document. That looks like a bug until you think about what similarity search actually does. The words Q1 financial results and Q2 financial results are almost identical in embedding space, so of course the Q2 document scores highest. The

interesting part is what happens next. The LLM gets the Q2 data in its context window, reads the grounding instruction we wrote that says only answer from the provided context, and tells the user it cannot find Q1 information. No hallucinated numbers, no guessing, just an honest I do not know. That is exactly the behavior you want from a production system.

Let's look at the second query again:

```
Query: "What were the Q1 financial results?"
↓
ChromaDB thinks: "Financial results? The Q2 document is most similar"
↓
Retrieved: Q2 results + expense policy (top 2 matches)
↓
LLM thinks: "I have Q2 data, but they asked for Q1. I was told to only use provided context and admit
when I don't know"
↓
Answer: "I cannot find Q1 information"
```

This demonstrates two crucial RAG principles: semantic similarity will find related content even when it's not exactly what you want, and proper prompt engineering ensures the LLM responds honestly when the retrieved context doesn't contain the precise answer.

Here is one more query, and unlike the Q1 example, this one should land. Back in *Chapter 1*, I answered a question about myself by pasting a bio into the prompt by hand. No pasting this time. I asked who Rany ElHousieny is, pointed `ask_rag` at *book_collection*, the index we built from the *rany.elhousieny.com* crawl in *Chapter 3*, and the retriever pulled my profile page as the top chunk. The model read it and answered from those crawled pages, not from training data. If you want to see it yourself, run the *Chapter 3* notebook first to build the index, then run this cell in the companion notebook and watch the scores and the answer print (search for " *Closing the Loop: Ask About Rany*" in the Notebook to find the cell).

```
Query: Who is Rany ElHousieny?

# STEP 1 - RETRIEVE: embed the question, search book_collection (top 2)
RESULTS: Found 2 documents:
doc_0 | Distance (l2, lower is closer): 0.539 | Rany ElHousieny, Ph.D (ABD), Senior Generative AI
Engineering Manager ...
doc_1 | Distance (l2, lower is closer): 0.746 | Key Accomplishments ...
```

```
# STEP 2 - GROUND: both chunks become the only context in the prompt
CONTEXT LENGTH: 6089 characters

# STEP 3 - GENERATE: the model answers from that context
Answer: Rany ElHousieny is a Senior Generative AI Engineering Manager with over 30 years of
experience, including pioneering NLP and AI advancements at Microsoft and driving Generative AI
innovation at Clearwater Analytics ...
```

As you can see in the previous example, we retrieved the content about Rany ElHousieny that we indexed in *Chapter 3* from *rany.elhousieny.com*. The retriever used similarity to retrieve the correct content from ChromaDB and added it as the context with the prompt, as shown in *Figure 4.1*. Here is the prompt I sent to the LLM:

```
You are an expert assistant. Use the following retrieved context to answer the user's question.
If the answer is not in the context, state that you cannot find the information.
Do not use any other information.

<context>
Rany ElHousieny Rany ElHousieny Senior Generative AI Engineering Manager Ex-Microsoft | 30+ Years
Experience | AI Solutions Architect 30+ Years of Experience... [doc_0 + doc_1: my full profile page,
6,089 characters of retrieved context]
</context>

<question>
Who is Rany ElHousieny?
</question>

Answer:
```

../../EPUB/NOTE

The first time I built a production system like this, I just threw everything ChromaDB returned into the prompt and wondered why the answers were so scattered. The fix was obvious once I saw it: you fetch more candidates than you need, maybe 20, but you only keep the ones above a relevance threshold, rerank those, and send maybe three or five chunks to the LLM. Duplicates get dropped. The model's token budget is the ceiling and evidence quality is the floor, and anything between those two lines is what the LLM actually sees.

With this, we have a complete, working RAG system. We have successfully built and validated the Query pipeline against the index created in *Chapter 3*. In the Graduating to a Framework section, we'll discuss when and why you

might want to move from this manual approach to a framework like LangChain.

Common pitfalls in first-time RAG builds

Three mistakes come up more than anything else in first-time RAG builds. Each one runs clean in a notebook and only shows up once real data and real users arrive, which is the expensive place to find them. The first one burned me personally: I indexed documents with one embedding model and queried with a different one, and the results were garbage because the two models produce vectors in completely different spaces. Use the same model for both, or the cosine scores are meaningless. The second is context window overflow: you set `n_results` to `20`, concatenate all the chunks, and the prompt exceeds the model's token limit. The API just fails, with no graceful degradation, just an error. The third is forgetting the grounding instruction. If you do not explicitly tell the LLM to answer only from the provided context, it falls back on its own training data, and you get confident-sounding answers that have nothing to do with your documents. With those three warnings in mind, let us talk about when it is time to graduate from a manual implementation.

Graduating to a framework

I resisted using LangChain for about six months after it came out. Every tutorial jumped straight to the framework without explaining what it was actually doing, and I did not trust code I could not trace line by line. The RAG function you just wrote is the kind of thing that changed my mind, not because LangChain does it better (it literally calls the same retriever and prompt steps we just coded), but because once I needed a second retrieval step feeding into the first one and an LLM deciding whether to call a SQL database or the vector store, the raw Python version was three hundred lines of if-else spaghetti that nobody wanted to maintain.

Why use a Framework?

Think of it like this: you can build a simple website using just HTML and CSS. But for a large, complex web application, you'd use a framework like React or Angular. The same principle applies here. You should consider moving to a framework like LangChain when:

- **You need to build complex chains**: What if you want to retrieve documents, then have an LLM decide if they are relevant, then perform another search, and then finally generate an answer? Chaining these steps together manually can get complicated. LangChain is designed specifically for this, allowing you to build complex 'chains' of logic with ease.
- **You need to integrate multiple tools**: What if your AI needs to search your vector database, but also query a SQL database or call an external API? LangChain provides a standardized way to give your AI 'tools' that it can decide to use to answer a question.
- **You are building agents**: This is the most powerful concept. An 'agent' is an AI that doesn't just follow a fixed chain of logic, but uses an LLM to decide *which tools to use and in what order* to accomplish a goal. This is the next evolution of the systems we're building, and LangChain provides the core building blocks for it.
- **You need production-ready features**: LangChain, along with its companion tool LangSmith, provides built-in support for logging, monitoring, and debugging your LLM applications, which are critical for production systems.

We won't do a full implementation here, but I want to show you what our query pipeline might look like using LangChain, so you can see the difference in abstraction. (Search for *Graduating to a Framework: LangChain* in the chapter Notebook):

```
from langchain.chat_models import init_chat_model
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# 1. The Retriever (what we built with collection.query)
vectorstore = Chroma(
    persist_directory="data/chroma_db",
    collection_name="book_collection",
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-small")
)
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})

# 2. The Prompt Template (what we built with an f-string)
```

```
template = """Answer the question based only on the following context.
If the answer is not in the context, say you cannot find the information.
{context}
Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)

# 3. The Model
model = init_chat_model(assistant.selected_model, model_provider="openai")

# 4. The Chain (pipes retriever -> prompt -> model -> text)
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)

# 5. Run it
chain.invoke("What is the new AI policy?")
```

As you can see, the core concepts are identical: retriever, prompt, model. But LangChain provides a standardized 'chain' to link them all together. The reason we built it manually first is so that you now look at this code and know *exactly* what the `retriever` is doing and *exactly* what the `prompt` is for.

My advice is this: master the fundamentals, as you have in this chapter. Then, when you're ready to build your first serious, multi-step AI application, adopt a framework like LangChain. Your foundational knowledge will make you a far more effective developer.

If you want to learn more about LangChain, I recommend my article *Mastering LangChain with Examples: The Future of AI Development*, https://levelup.gitconnected.com/mastering-langchain-with-examples-the-future-of-ai-development-180bea2a6d27.

That covers building RAG yourself, with or without a framework. The next question is whether you should be running any of it yourself at all.

Managed RAG and enterprise search options

We built everything so far by hand. That was on purpose; you cannot trust what you cannot see. But I rarely want to own all that plumbing in

production, and you probably will not either. This is where the managed options come in. The cloud providers and a couple of specialized vendors will run retrieval for you and scale it, and *Table 4.1* is my shortlist of the ones worth knowing.

Service	Best For	Key Strength
Amazon Bedrock Knowledge Bases (https://aws.amazon.com/bedrock/knowledge-bases/)	End-to-end RAG on AWS without managing vector infra	Built-in chunking, citations, guardrails, model selection
Amazon Kendra (https://aws.amazon.com/kendra/)	Enterprise search with large connector catalog and GenAI Index for RAG	Hybrid search, FAQs, auth-aware results, GenAI augmentation
Azure AI Search + "On Your Data" (https://learn.microsoft.com/azure/search/search-what-is-azure-search)	Grounding Azure OpenAI with enterprise security	Hybrid retrieval, semantic ranking, RBAC, vector search
Vertex AI Search and Grounding (https://cloud.google.com/vertex-ai/generative-ai/docs/grounding/overview)	RAG on GCP with Google-native grounding	Citations, safety guardrails, Vertex AI integration
Glean (https://www.glean.com/)	Turnkey enterprise search across SaaS tools	Minimal infra, broad connectors, identity/permissions sync

Table 4.1: Managed RAG Services Comparison

The table tells you what exists. It does not tell you which one to pick, and that is the question I get asked most.

How to choose (practical guidance)

There is no single right answer here, so the quickest way to decide is to start from where you already are. Match your situation to one of the cases ahead, and you will have a strong default to work from.

- **Already on a cloud**: On AWS, prefer Bedrock Knowledge Bases for Bedrock-native apps, and add Kendra when you need rich connectors and ACLs. On Azure, Azure AI Search with "On Your Data" is the default path to ground Azure OpenAI. On GCP, Vertex AI Search keeps everything inside Vertex AI with citations and controls.
- **Want turnkey**: Glean gives the fastest end-user value across many SaaS tools with minimal engineering.
- **Need heavy compliance or ACLs**: Favor Kendra, Azure AI Search, or Glean for their mature identity and permissions models.
- **Need maximum control**: Roll your own (as we did here) or use a framework like LangChain on a managed vector store, and lean on the clouds' managed options when ops burden and connector breadth matter more than low-level control.

Cost and ops trade-offs (at a glance)

The real question is where you want the cost to sit, on an invoice or on your own team. The three notes below lay that out: managed versus rolling your own, and the path most teams take between them.

Managed platforms shift cost from engineering time to service fees but reduce maintenance, upgrades, scaling, and monitoring overhead.

DIY offers control and potentially lower unit cost at scale, but you own ingestion, connectors, quality (chunking, rerank, filters), observability, and SLAs.

A common pattern: start with a managed option to validate value quickly; migrate hot paths to custom retrieval when you need deeper control, latency tuning, or lower TCO.

At Clearwater Analytics, we started with LangChain because it gave us the right building blocks to move fast. Before LangGraph matured, we built our own multi-agent orchestration (I walk through the design and lessons in these two articles: *Multi-Agent Systems and CWIC Flow* (https://medium.com/cwan-engineering/multi-agent-systems-and-cwic-flow-pioneering-ai-orchestration-5867ac9c0a34) and *Building Multi-Agent Systems with LangGraph* (https://medium.com/cwan-engineering/building-multi-agent-systems-with-langgraph-04f90f312b8e). However, as the big providers accelerated and their managed offerings improved reliability, connectors, and guardrails, we began shifting selectively to those services where they produced better outcomes. When Amazon came with Amazon Bedrock AgentCore, we were one of the first companies to implement it https://aws.amazon.com/blogs/machine-learning/amazon-bedrock-agentcore-is-now-generally-available/. The pattern I've seen repeatedly: build core understanding and control first, then adopt managed services where they clearly reduce operational load and improve quality so your team can focus on business logic.

Choosing a vector database (Top 5)

Choosing a vector database is rarely about finding one option that is best for everyone. The right choice depends on your existing technology stack, expected scale, operational capacity, and the level of control you need over deployment and retrieval. A team already working with PostgreSQL, for example, will have very different priorities from one building a large-scale, fully managed AI application. The following table compares five popular options and highlights where each one fits best:

Service	Best For	Key Strength
Pinecone (https://www.pinecone.io/)	Zero-ops vector search at production scale	Fully managed, serverless, hybrid/metadata filtering, strong SLAs
Weaviate (https://weaviate.io/)	OSS flexibility with optional managed cloud	Hybrid retrieval, filters, schema flexibility, good MLOps integrations

Service	Best For	Key Strength
Qdrant (https://qdrant.tech/)	High-performance, cost-effective vector search	Fast approximate search, strong payload filtering, simple API
Milvus / Zilliz Cloud (https://milvus.io/)	Serious scale with OSS or managed option	Distributed partitioning/replication, strong ANN engines, 40K+ GitHub stars; Zilliz Cloud for managed
PostgreSQL + pgvector	Teams already on Postgres who want fewer moving parts	Transactional guarantees, joins with structured data; best at moderate scale

Table 4.2: Vector database comparison

How I choose in practice: if I need zero-ops at scale, I go with Pinecone. For OSS flexibility with a cloud option, Weaviate or Qdrant work well. For the largest enterprise deployments, Milvus with Zilliz Cloud handles serious scale. And if I am already on Postgres and my workload is moderate, `pgvector` keeps things simple with fewer moving parts.

> **../../EPUB/NOTE**
>
> Capabilities evolve quickly. For the latest details, review vendor docs and recent benchmarks before finalizing your choice.

Cloud-Native Vector options (AWS, Elastic, MongoDB)

If you're already on one of these platforms, its built-in vector capabilities are often the most operationally convenient, with tight IAM, networking, monitoring, and compliance:

Service	Best For	Key Strength

Service	Best For	Key Strength
Amazon OpenSearch Serverless (https://aws.amazon.com/opensearch-service/serverless-vector-database/)	Vector search on AWS without cluster management	Serverless scaling, k-NN + keyword hybrid search, GPU-accelerated indexing
Elasticsearch (https://www.elastic.co/elasticsearch)	Teams already on Elastic who want to add vector search	Mature BM25 + kNN hybrid search, large ecosystem, self-hosted or Elastic Cloud
MongoDB Atlas Vector Search (https://www.mongodb.com/products/platform/atlas-vector-search)	Teams already on MongoDB who want vector search alongside documents	Native integration with document model, ANN and ENN search, Atlas managed

Table 4.3: Cloud-Native Vector Services Comparison

Applying RAG to the four case studies

The demo in this chapter used four company documents in one ChromaDB collection, and if you are building something close to that shape, the domain-specific assistant, the code runs almost unchanged; just add version tags to the metadata so you never serve a retired spec. For the enterprise assistant, reread the pitfalls section before anything else, because a missing grounding instruction in a permissioned system is not a hallucination problem; it is a data leak, and the managed RAG section covers platforms that handle access control natively.

Chapter 5 will discuss enterprise RAG in more detail. Most of this chapter does not apply to a personal assistant directly because email and calendar data come from APIs, not a document store, but the Q1/Q2 honest-failure behavior is exactly what you want when the user asks about a meeting that was never indexed. Content moderation is where the embedding mismatch pitfall from

the pitfalls section turns into an operations incident: the policy index gets re-embedded every time rules change, and if the query encoder falls behind, you are matching against the wrong vector space.

Summary

In this chapter, we built our first complete RAG system from the ground up, focusing on the query pipeline. We started with the fundamental understanding that RAG systems consist of two distinct pipelines: the indexing pipeline and the query pipeline (our research assistant). This separation allows for efficient, scalable systems where heavy processing happens once during indexing, and real-time queries remain fast and responsive.

We implemented a hands-on RAG system using ChromaDB and OpenAI, walking through every component of the Query pipeline from retrieval to answer generation, using the index that was created in *Chapter 3*.

On the technical side, we wired the query pipeline to the *Chapter 3* index and grounded the model in what it retrieved. We also covered the production pitfalls that bite first: embedding model consistency, context window limits, and the missing grounding instruction. And we saw how an honest RAG system behaves when the answer is not in the retrieved documents, returning "I don't know" rather than hallucinating.

You now understand every component of a Naive RAG pipeline and can make informed decisions about scaling, optimization, and technology choices as your requirements grow.

Most importantly, you've gained the foundational knowledge that will make you effective whether you choose to build custom solutions, adopt frameworks like LangChain, or leverage managed services. This deep understanding of the fundamentals is what separates hobbyists from professional AI engineers.

Now that you have a working RAG system, the next chapter flips the question from *"how do I build it*" to "*what breaks when real data hits it.*" *Chapter 5* walks through the failures I ran into once production data started flowing: temporal drift where yesterday's answer is wrong today, keyword queries that embeddings miss entirely, multi-hop questions that span more than one

system, and the architectural decisions around chunking, index design, access control, and cost that determine whether a RAG system holds up or falls apart at scale.

5

Data Strategy for Advanced RAG: When Your Data Tells You What You Need

A failure pattern I have seen repeatedly in production RAG: a user asks for "section 12.3" of a specific document, and the system returns section 14.1 of a different document, with high confidence. Vector similarity is great at understanding meaning. It is basically blind to exact numbers. Bolting keyword search onto the side of the vector index is the first fix I always reach for, and it usually turns out to be the first of a long series of patches.

On most production RAG systems I have worked on, the same pattern kept repeating. Every few weeks, something else would break, and we would patch it. Every time, the fix itself would teach me something about the data that I really should have figured out before indexing anything. Old documents kept winning over newer ones because they happened to be better written. Structured tickets did not play well in the same index as long-form wiki pages; their shapes as documents are just too different. Users kept asking questions whose answers lived half in one source and half in another, which single-source retrieval had no way to handle. The lesson was always the same one. Your data decides what retrieval strategy you need, not the other way around.

What follows is the arc I have seen play out repeatedly across production RAG systems, in roughly the order the problems tend to surface. Real failure patterns from production, and what each one teaches you about what your data wants from you at ingestion time. What *Chapter 4* built is what the RAG literature calls a **naive RAG**: vector retrieval over a single source, no metadata filtering, no hybrid search. This chapter starts there and walks through the

problems that surface as you add complexity. By the end, you should have a much better feel for what to plan for before you index the first document, and which metadata fields you really cannot skip. At Clearwater Analytics, we have run enterprise RAG successfully in production for years across our AI assistant systems. I have learned a lot from the scale of this system.

The RAG landscape has so many named patterns, from Standard and Conversational through Corrective, Adaptive, Self-RAG, Fusion, HyDE, Agentic, and GraphRAG. This chapter does not walk through them as a taxonomy, because the differences that matter for a data strategy book are not the names of the architectures. They are the data decisions each architecture forces you to make. Hybrid search is about the index. GraphRAG is about what you embed and how you link it. Agentic search is about whether you embed at all. The rest of the chapter treats the data decisions, with the relevant architectures showing up as examples where they matter.

The first one shows up before you even add a second source, and it is almost always the same story: your best new documentation gets buried under older pages.

In this chapter, we cover the following topics:

- Detecting and correcting temporal data drift
- Adding keyword exact match retrieval
- Adding a second source exposes volume and schema mismatches
- Handling multi-hop failures when answers span multiple systems
- Building your data architecture

Technical requirements

This chapter builds on the RAG pipeline we set up in *Chapter 4*, so you already have most of what you need. You need Python 3, the OpenAI library (`pip install OpenAI`), and ChromaDB (`pip install chromadb`) for the vector store, plus a valid API key. If you have not set up a local environment yet, follow the instructions in *Chapter 1*. However, the easiest way is to use Google Colab and skip local setup.

The full code is in the `chapter_05` folder at `chapter_05/Jupyter_Notebooks/Chapter_5_Notebook.ipynb`. You can open it

directly from GitHub the way we did in earlier chapters. The setup cells at the top reuse the *Chapter 4* environment, so run those first before you jump to any section.

You do not need to bring your own data, since the notebook ships with the sample data each section uses. For temporal drift, there are wiki pages with old and superseded versions. The keyword exact match example uses a handful of runbooks for common error codes like 401, 403, 429, 500, 502, and 503. And for multi-hop, I included Jira and Confluence style documents carrying the entity-link fields we get to later. Each section tells you which cell to jump to, so you can read and run the code side by side.

Detecting and correcting temporal data drift

A Naive RAG over a single Confluence space will work fine until the documentation starts spreading across multiple spaces for the same subject. The pattern is almost always the same shape. The old document was written years ago, well organized, and detailed. The new version lives in a different space, maintained by a different team, shorter because the emphasis shifted. Internal coding standards are a good example: a company has Java conventions written five years ago, comprehensive, and polished. When another team starts adopting Python and writes new standards, those go into a new space, and the Java section gets thinner because the emphasis is now on Python. A new hire asks the chat system, powered by Naive RAG, 'What are the Java coding standards?' and the system returns the old page. The richer content scores higher in semantic similarity. The new hire follows it. The lesson lands a few weeks later when they have to redo the work. Every team that creates new documentation without retiring old versions creates the same failure. The solution is to score documents on recency, not just relevance.

Similarity search does not understand that the newer version replaces the older one. It has no concept of time. The problem was not the search algorithm; it was that relevance and recency are independent signals, and pure vector search only knows about one of them.

Running the query 'What are the code conventions for naming classes?' against a production wiki index of around a hundred thousand chunks

confirms the problem:

Rank	Score	Title	Content Size	Current?
1	0.62	Code conventions	~1,800 chars	No, marked OLD in the title
2	0.60	Java Best Practices	~2,000 chars	No, legacy language standards
3	0.47	[OLD] Code conventions (chunk 2)	~1,200 chars	No, same old page
4	0.43	Code Conventions (Draft Replacement)	~1,400 chars	Yes, the replacement

Table 5.1: Relevance Scores for the Query

As you can see from the results, the most relevant document is marked as [OLD]. The newer document is ranked fourth because it is shorter and still being filled in. The system cannot read "[OLD]" in the title and understand that it means *do not use this anymore*.

The fix for this issue has two parts. The first part is to capture time information before indexing the documents into the RAG system. We need to store three things in the metadata: when the document was created, when it was last updated, and which document it replaces. This goes into the metadata stored with each document, not into the document itself. Naive RAG returns older, higher-similarity documents regardless of version. The second part is temporal filtering on the query side, which removes superseded versions before the search runs and boosts newer documents in the ranking. We discussed the ingestion part in *Chapter 3*, and the retrieval part in *Chapter 4*. All we need to do is add the metadata before chunking and apply temporal filtering after retrieval, as shown in *Figure 5.1*:

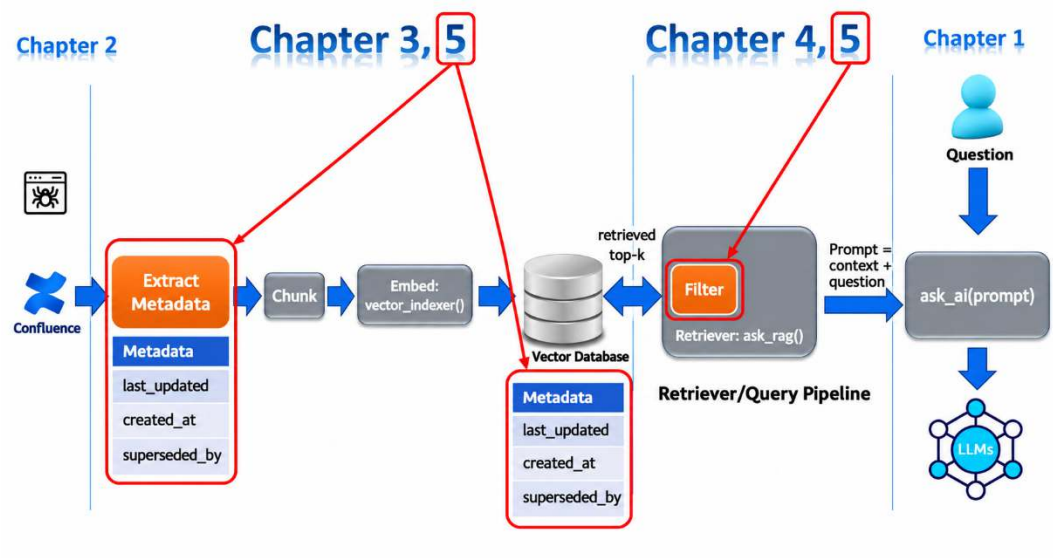


Figure 5.1: Adding Metadata and Filter

After fixing the temporal issue, a RAG system will run fine and return the current documents until someone adds a new source type. Runbooks are a common second source. Support engineers need them during on-call shifts, and the runbooks go into the same index as the wiki pages. The escalations start almost immediately: tickets take longer to investigate because similarity search alone cannot find the correct runbook for the exact issue.

Adding keyword exact match retrieval

When on-call engineers start escalating, you take a couple of the escalated cases to investigate the issue. It is clear from the first one that the issue is in the vector search. When the engineer searches for a specific error, for example, error 502, the similarity search will look for a similar meaning, not the exact number. RAG returns the runbooks for different errors, such as 503. This is the failure mode that keyword exact-match retrieval addresses. The embedding model sees "server error" in both, but not the exact value. For the model, both 502 and 503 are the same concept. In the Notebook, search for `error code runbook`. This is where we add the data for this scenario. We added six runbook documents for six errors: 401, 403, 500, 429, 502, and 503, as shown in *Figure 5.2*:



Figure 5.2: Six simulated runbook documents inserted into the database

To locate the first issue scenario in the notebook, search for the comment "PROBLEM: Vector search cannot distinguish error codes". First, the on-call engineer sees the error "502 Bad Gateway" and searches for: "server returns 502". Since the wiki has separate runbooks for six different errors, the vector search will return all of them with different similarity scores, as shown in this code:

This simulates the support engineer searching for "server returns 502"

```
results = ch5.query(query_texts=['server returns 502'], n_results=4,
                    include=['documents', 'distances', 'metadatas'])
```

When we list the results by cosine distance, the runbook for error 503 comes first because it has the shortest distance. The runbook for 503 will address a different issue regarding "service unavailable," which is a different problem than 502 "Bad Gateway." When the engineer follows the instructions in the 503 runbook by checking the CPU utilization and auto-scaling configuration,

they will not find the issue. As a result, the engineer will spend time on the wrong diagnosis. The runbook for 502 will guide the engineer to check the proxy and backend instead.

```
=== Vector search for "server returns 502" ===
  1. (dist: 0.3546) [WRONG] error=503 | Troubleshooting HTTP 503 Service Unavailable: This error means the ser...
  2. (dist: 0.4113) [CORRECT] error=502 | Troubleshooting HTTP 502 Bad Gateway: This error means the upstream se...
  3. (dist: 0.5689) [WRONG] error=429 | Troubleshooting HTTP 429 Too Many Requests: This error means the clien...
  4. (dist: 0.5697) [WRONG] error=500 | Troubleshooting HTTP 500 Internal Server Error: This error means the s...

The 503 runbook (Service Unavailable) outranks the 502 runbook (Bad Gateway).
The fixes are completely different: 503 = check CPU/scaling, 502 = check proxy/backend.
```

Figure 5.3: Search results and their cosine distances

From the results in *Figure 5.3*, you can see that the cosine distance for the 502 runbook document is 0.4113, while it is 0.3546 for the **503 runbook** document. The correct runbook is farther than the wrong one. The embedding model sees both as the same "server error" concept and treats the specific code number as noise.

> **../../EPUB/NOTE**
>
> In *Chapter 3*, we learned that cosine similarity measures how close two embeddings are, with 1.0 meaning they are identical. ChromaDB measures cosine distance, not cosine similarity. The notebook creates the collection with the cosine space setting, since ChromaDB defaults to L2 distance. The cosine distance measures how far apart the two embeddings are, with 0.0 indicating they are identical.

The fix is to use a hybrid search with BM25 and vector, then merge the results and re-rank. In production, you do not need additional code. This is always taken care of by search engines like Elasticsearch and Azure AI Search, as shown in *Figure 5.4*:

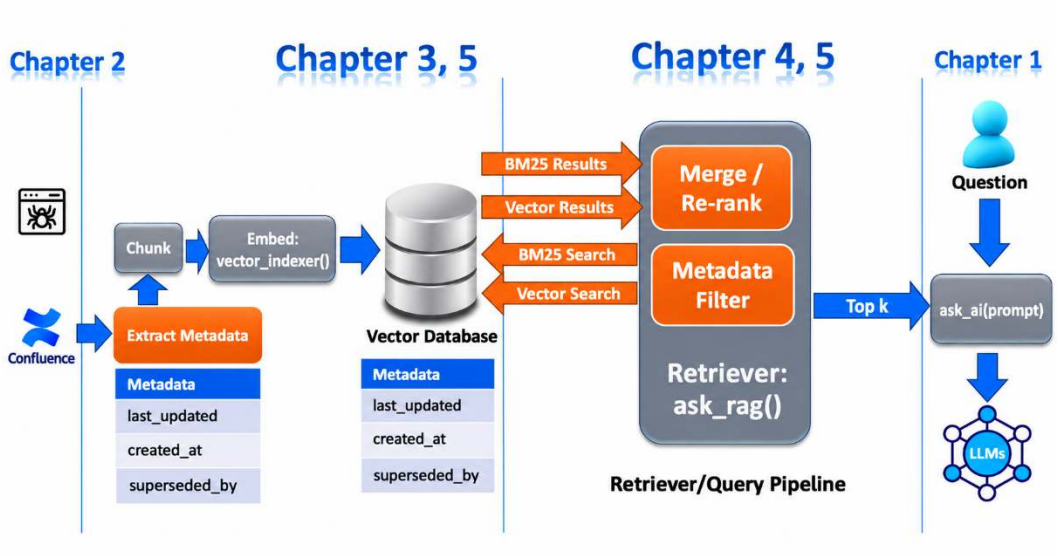


Figure 5.4: The hybrid search pipeline adds BM25 and vector search in parallel inside the retriever service

Once hybrid search is in place, the system returns both semantically similar documents and exact matches. However, support engineers want to reference the bug numbers for issues and look for similar bugs. Adding another source becomes the obvious move: Jira. Indexing Jira without any changes to the system solves one issue and creates three more.

Adding a second source exposes volume and schema mismatches

Adding a second source changes everything. To help support engineers search for similar issues in Jira tickets, most teams index Jira next. The first thing you notice is the volume difference. On the systems I have worked on, the Jira index had thirty to forty times as many chunks and roughly the same ratio in storage compared to Confluence. The Jira side was in the millions of chunks occupying tens of gigabytes, while Confluence had maybe a hundred thousand chunks in under a gigabyte. The size difference comes from attachments, comments, discussions, and links in each Jira ticket. Every support ticket, bug, feature request, or design gets its own Jira ticket. Confluence pages, on the other hand, are handcrafted by engineers. With that volume difference, retrieval will favor Jira chunks over Confluence chunks.

A support engineer asked the chat system, 'Who fixed the OAuth bug and when was it deployed?' To answer the question, the system must get data from multiple sources. It needs the assignee and ticket number from Jira, and the deployment date with the release notes from Confluence.

The schema difference between Jira and Confluence made it worse. Confluence pages are mostly prose. Jira, however, is a mix of structured content for the metadata and prose in the description and comments, as the following example shows:

```
Title: TKT-8421 - Null pointer exception in account loader
Status: Closed
Priority: High
Assignee: john.doe@company.com
Description: When account ID is null, the loader throws NPE at line 47...
Comments: [debug thread, 8 exchanges, stack traces, code snippets]
```

Every field is structured, and every comment is prose. The chunker treats both the same. As a result, the system returned unrelated bug tickets and four chunks from a deployment runbook for a different issue. Adding Jira surfaced three pressures on the system: a volume mismatch, a schema mismatch, and a new question type that single-source retrieval cannot handle. The volume and schema differences shape the data architecture choices the second half of this chapter walks through. The new question type is the more immediate failure, the one the next section is about.

Handling multi-hop failures when answers span multiple systems

With Jira and Confluence both indexed, the system handles single-source questions well. Everything works for a while. Then someone asks a question the system cannot handle with a single search. 'What caused the OAuth outage last quarter and what changed to prevent it from happening again?' Jira has the ticket: fix description, assignee, deploy date, the whole debugging thread. Closed cleanly. But that only covers half the question. The root-cause analysis lives in the post-mortem, which is a Confluence page, not a Jira ticket. The preventive actions live there too. Nobody linked the two in the index. The answer is a chain: Jira has one piece, Confluence has the other, and the retrieval layer only pulls one link at a time. This is the multi-hop problem.

Reviewing a typical Jira schema confirms it. A ticket has many structured fields: title, assignee, status, deploy date, fix version, resolution date, comments, and so on. None of them points to the Confluence post-mortem page that documents the institutional learning from that incident. In the real world, the Jira ticket and the post-mortem are obviously connected; someone on the team wrote both about the same incident. But in the search indexes, those connections disappear. Every source becomes an isolated silo.

The problem stays invisible with one source. With only Confluence, every answer lives in the wiki. You might pull two different pages, but both are in the same index. As soon as Jira is added, the system has two indexes. Any question that needs both hits the wall right away.

The fix has two parts. The first part runs at ingestion. When we index a Jira ticket, we scan the body and the comments looking for references to the post-mortem page. We pull that link out and store it as metadata on the Jira document. A simple field: `linked_postmortem`. We do the same for Confluence post-mortem pages. A post-mortem almost always links back to the Jira incident ticket it describes, so we parse that link when we ingest the page. The two sources stay in their own indexes, but now the link between them is a real field, and the retrieval layer can follow it.

This is a lightweight version of what the **GraphRAG** community calls **entity-relationship extraction**. The full version, which Microsoft Research published in 2024, sends every document through an LLM at ingestion to pull out named entities and the relationships between them, then stores the result in a graph database. Rich and flexible, but expensive at LLM rates as of 2026: three to five times the baseline cost in LLM calls, plus the operational weight of running a graph DB alongside the vector index. That said, the gap is closing. Graph databases like Neo4j now ship native vector indexes, so you can store the graph and the embeddings in the same engine instead of running two separate systems. What I am describing is the same idea narrowed to one specific signal: URLs that appear in the document body point to other documents in the corpus. Parse those out, store them as flat metadata on the document, and skip both the LLM extraction and the graph database.

The second part runs at query time. Instead of one search, we chain them. The first query finds the Jira ticket for the OAuth incident. Before we return

anything, we read the `linked_postmortem` field on the ticket and fetch the post-mortem page. Both documents go to the LLM together. The LLM has everything: the fix description and timeline from Jira, plus the root cause and preventive actions from the post-mortem. It can answer the question. *Figure 5.5* shows where the new entity link field and the multi-hop step sit in the overall architecture.

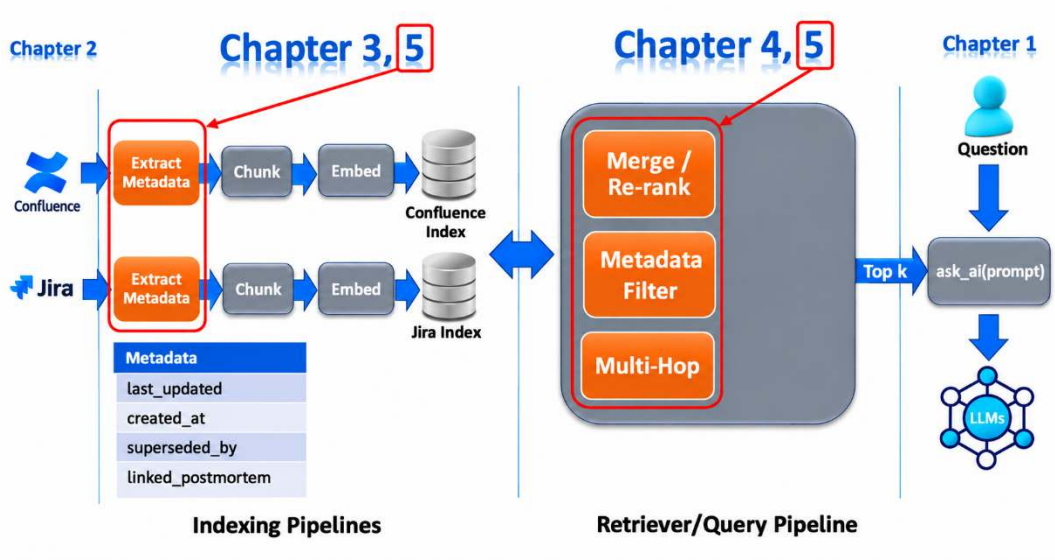


Figure 5.5: RAG architecture with Chapter 5 additions: entity link fields at ingestion, and a multi-hop step in the retriever that chains across indexes

There is another approach for when the entity links are messy. We can use the LLM to break the question into sub-queries before we do any retrieval. "What caused the OAuth outage and what did we change to prevent it?" becomes two questions. "Who owned the OAuth fix and when was it closed?" goes to Jira. "What was the root cause of the OAuth outage?" goes to Confluence post-mortems. Each one is a normal single-hop search. The LLM combines the two results at the end. This is called **query decomposition**. It adds an extra LLM call on every complex question, but it works when the links in the source data aren't clean enough to extract at ingestion.

Neither fix is free. Extracting entity links during ingestion adds work to the pipeline. Query decomposition incurs an additional LLM call for every complex query. But the alternative is worse. The alternative is an assistant that quietly returns half-answers, and once that happens a few times, the support engineers stop trusting it. In practice, entity extraction is usually the right primary fix. Cross-source links in production data tend to follow the

same patterns. Engineers paste the post-mortem link into the ticket comment when they update the incident, and they reference related ticket IDs the same way, mostly out of habit. A parser catches most of these. For the ones it misses, query decomposition is the fallback I have learned to keep ready.

The pattern shows outside support engineering, too. Take a clinical coordinator who wants to know if a recent formulary change affects her patients. She has to start with the formulary update, then jump to the electronic health record (EHR) to pull the list of patients on that drug class, and then check the clinical guidelines for the right dosing adjustment. Or take a trust and safety analyst getting a new report on a user account. She needs the current flagged content, any past violations on the account, and the specific policy rule under which the report was filed. All three are in different stores. Different data shapes, same underlying problem. The answer to the user's question is spread across a chain, and a single search can only retrieve one piece of it. All of these problems share the same root: the data was not architected properly. The rest of this chapter is about fixing that.

Building your data architecture

By the time multi-hop is fixed, a production RAG system will have accumulated a dozen patches in the retrieval layer. Hybrid search goes in first, once you figure out that error codes score nearly identically in the embedding space. Decay scoring follows a few weeks later, after users start complaining that old documents keep winning. Entity link fields are always the last batch, and they only happen because someone asks why the system cannot tell them which pull request fixed their bug. Reasonable question. The honest answer is usually that the data layer has no way to know. Every patch makes things better. But the list keeps getting longer, and every new feature needs a schema change, and every schema change means a full reindex, and reindexes cost the better part of the day while finance starts asking about the cloud bill.

The real problem is building the retrieval layer on top of a data layer designed for something simpler. A document store that treats every source the same way—Confluence runbooks indexed with the same chunk settings as long Jira tickets with twenty comments, same embedding model, everything landing in one index. That works when you have one source. Once you start adding more, things you did not plan for at ingestion keep showing up as retrieval

failures at query time. Temporal decay requires a *last_updated* field that is not there, and adding it means a full reindex of the corpus. Identifier routing is worse because you need either a separate index or a completely different query architecture for the exact-match cases. Every new feature runs into schema decisions made early on, when nobody was thinking about where the system was going.

This section is about building that foundation right from the start. Not a perfect system on the first try (I have never seen that work in production), but one designed to grow without requiring a full reindex every time you need a new feature.

Deciding what goes in the schema

The first decision, and the one that pays back the most over time, is what metadata you store on every chunk. This sounds like paperwork, but it is the most consequential architectural choice you will make.

A typical first version indexes documents with something like seven fields: `id`, `page_id`, `url`, `title`, `category`, `content`, and `source`. No date fields. None of the version or supersession flags that temporal decay requires. Nothing for access control, nothing to link a chunk back to related documents in other indexes. You discover each missing field the hard way, which means discovering it when a user needs something you cannot give them without a full reindex.

Every chunk in a well-designed system needs a minimum set of base fields regardless of where it came from. The six I would never skip are `source_system`, `doc_id`, `chunk_id`, `created_at`, `last_updated`, and `content_type`. None of them are exciting, but skipping any one of them tends to force a reindex later when you find out you needed it. `source_system` tells the router which index a chunk belongs to, which is necessary the moment you go multi-index. Date fields are what make temporal decay work without a full reindex. `content_type` is used for source-aware chunking and query routing decisions. I've added all of these late in projects when they weren't there from the start, and it's painful every time.

On top of the base fields, add source-specific fields by problem type. For documents that have version histories, store version and a superseded Boolean. When a new version is published, flip the flag on all previous

versions and add a `superseded_by` field pointing to the replacement. This is the fix for temporal drift at the data layer, not the query layer. A binary filter that eliminates stale documents before scoring even runs. You do not need a decay function for documents that have a clear successor. You need one only for documents that age continuously without ever being replaced, like architecture decisions that slowly drift away from the system they describe.

Handling access control

Access-controlled sources take a bit more work. Some sources have access restrictions, some documents are open to everyone, and others are restricted to specific roles. I started checking permissions at query time against the live source. That is always correct, but it is also slow. The other approach is to store the permission as metadata at ingestion and just filter on it, which is fast, but as soon as someone's role changes, the permission is stale.

What I do in practice is a hybrid, because neither approach on its own is safe. At ingestion time, I store the coarse-grained permission as metadata on every chunk, and the search index applies that as a filter on every query. On top of that, at query time, I also make a live call to the actual permission source to confirm the current user really has the role the chunk requires, and I cache that check for a minute or two so you are not paying its latency on every single request. Both checks run on every query. The indexed filter keeps the search fast, while the live check ensures nobody gets content they lost access to since the last reindex ran. Cutting either side would be a real security problem, and for any source with real access controls, I would not ship it that way.

Storing entity links for multi-hop retrieval

For multi-hop queries, store entity links on every chunk that mentions them. When you index a Jira ticket, scan the body and comments looking for anything that references other systems: PR numbers that show up in comments, version strings, related ticket IDs. Those go into `linked_pr`, `linked_ticket`, and `linked_release` fields on the chunk. Release notes get the same treatment, pulling in whatever PRs and Jira tickets are mentioned. These fields are what the multi-step retrieval query actually follows. Without them, once you have retrieved the first hop, there is nothing in the chunk that tells you where the second hop should go.

IDs are all you need in those link fields. A `linked_pr` field that stores the entire PR description will bloat the index by three times or more and slow ingestion to the point where it starts falling behind. The full object gets fetched in the second hop, which also means it is always current when you retrieve it, rather than a snapshot from ingestion time.

Choosing a chunking strategy

The default in most frameworks is fixed-size chunks with the same setting for every data source. Most teams leave it because the indexing pipeline runs without errors. A few million chunks later, the cost shows up as retrieval failures, and I have learned to think about the chunking strategy before the first ingestion run.

Fixed-size chunking works reasonably well for homogeneous prose. A Confluence runbook is mostly paragraphs, and a chunk sized to fit a typical paragraph or two will capture a complete thought most of the time. Tickets are different. A Jira ticket has a title, a description, a status field, an assignee, comments from multiple engineers, and a resolution. Chunk them by character count, and you get fragments that start in the middle of one engineer's comment and end partway through the next, which is incoherent for retrieval. The chunker does not know the title is one semantic unit and the comment thread is another.

Structure-aware chunking

Structure-aware chunking is what actually worked for me. For Confluence, I settled on recursive character splitting, slightly smaller than the framework default and with an overlap that keeps paragraphs together across chunk boundaries. Jira tickets are a completely different animal, basically a structured object rather than prose, so I split them at field boundaries instead of by character count. Title becomes a short chunk of its own. Descriptions are chunked separately based on length, which varies enormously between tickets. For comments, individual chunks per comment, sized to the comment, and each chunk, regardless of which field it came from, carries the parent ticket metadata. So when an engineer searches for 'authentication timeout' now, they get back the exact comment where someone mentioned it, not a fixed-size blob that started halfway through the assignee field.

Contextual chunking

One technique worth evaluating is **contextual chunking**, in which you prepend a summary of the parent document to each chunk before embedding. A chunk pulled from page 28 of a 60-page architecture document provides almost no context on its own. Adding a two-sentence summary of the parent document gives the embedding model enough signal to score the chunk correctly against a query that uses vocabulary from the introduction. Anthropic's September 2024 contextual retrieval research showed that prepending chunk-level context before embedding reduced retrieval failure rates by 35% on its own, by 49% once contextual BM25 was added, and by 67% with reranking layered on top (Anthropic, "Introducing Contextual Retrieval," September 19, 2024, https://www.anthropic.com/news/contextual-retrieval). Results vary significantly by domain. The right approach is to test it on a sample of your actual queries before committing, since contextual chunking runs about one dollar per million document tokens with prompt caching turned on, and higher without it. A 2024 alternative worth evaluating alongside it is late chunking, introduced by Jina AI, which embeds the full document first and pools chunk boundaries afterward, cheaper than Anthropic's approach and particularly helpful for pronouns and anaphoric references (Günther et al., "Late Chunking: Contextual Chunk Embeddings Using Long-Context Embedding Models," arXiv 2409.04701, September 2024, https://arxiv.org/abs/2409.04701). I applied this at work to a production Confluence corpus, which is mostly multi-section architecture documents. It helped exactly where the research said it would, on the vocabulary mismatch cases, and I rolled it to the top traffic sources first before reindexing the rest.

Chunking code

Code is a different case entirely. If you do need to chunk it, the one rule is to cut at function and class boundaries, not at character count, because a 300-line function split at 2,000 characters leaves you with fragments that have no retrievable meaning on their own. But the bigger question for code in 2026 is whether you should be chunking it at all, which we cover in the Retrieving code section.

That covers what each chunk looks like on its own. The next question is how they all get organized once you have a few million of them, and the decisions

from there shape the retrieval system almost as much as chunking does.

Designing index architecture

The question of one index versus many comes up in almost every production system I have worked with. We started with one index because it was simpler to operate. We moved to separate indexes by source type because routing became necessary. I am still not certain the second decision was right in all cases.

The argument for separate indexes is routing clarity. If a query asks about a specific Jira ticket, routing it directly to the Jira index avoids loading up the query layer with 100,000 irrelevant Confluence chunks that it would then have to filter down anyway. Faster and cleaner results. The argument against separate indexes is that multi-hop queries get complicated fast. When the answer is partly in Jira and partly in Confluence, you need two queries to two indexes, a merge step, and a combined ranking step. Adding GitLab and Salesforce makes the orchestration layer the most complex part of the system. In my own production system, I ran more than a dozen separate source-type indexes, and I will say the orchestration complexity was worse than I estimated when I made the split. I am still not certain that a single unified index with a strong source-type filter would not have been simpler to operate while maintaining similar query-time performance.

What we ended up doing was to use one index per major source type, with each index sharing the same metadata schema. Jira data has its own index. Confluence has another. Anything else that grows to significant volume eventually gets its own, too. Salesforce was the next source we had to plan for. Customer-facing questions like "which accounts hit the OAuth issue before we shipped the fix" need the case history from Salesforce in addition to the engineering trail we already have in Jira. That is a three-hop query: case in Salesforce, bug ticket in Jira, and post-mortem in Confluence. Getting it right requires the same shared metadata discipline as the two-hop case, just with more link fields in the schema (`linked_case` alongside `linked_postmortem` and `linked_pr`). What matters is that regardless of where a chunk is physically stored, it always carries the same base fields along with it. Same entity links. Same access control metadata. Same everything the query layer needs. When a question is obviously about a single source, the query router can send it

directly to that one index. When an answer is likely to span multiple sources, we can query all of the indexes together and merge the results at the end.

Inside each index, hybrid search is almost always the right algorithm for enterprise data. Pure vector retrieval will find you semantically similar documents, which works well for natural language queries, but is basically useless when someone types a ticket number and wants that specific ticket back; no amount of semantic similarity helps you there. Keyword retrieval handles the exact-match cases. Running both in parallel and then re-ranking the merged results covers most of what enterprise users actually ask for on any given day. The storage cost is roughly 2x compared to keyword-only, with near-zero latency overhead at query time when the two searches run in parallel. That trade-off is almost always worth making.

For embedding model selection, the Massive Text Embedding Benchmark (MTEB) leaderboard is a reasonable starting point. MTEB benchmarks models on retrieval, classification, and clustering tasks, and a handful of semantic similarity tests across public corpora. The problem is that public benchmarks don't always tell you much about your specific domain. A model that performs well on legal contracts can underperform significantly on software engineering tickets, mostly because the vocabulary and query patterns are different. The practical approach is to pull a sample of your actual queries, run two or three candidate models against them, and see which one your data actually responds to. Don't commit to an embedding model based on leaderboard position alone. **Benchmarking Information Retrieval** (**BEIR**) is another benchmark worth looking at for retrieval in particular, since it covers web, biomedical, legal, and scientific datasets and often tells you something the MTEB averages do not. If the best pre-trained model still falls short on your corpus, fine-tuning usually gives another ten to thirty percent on specialized domains like legal, medical, or code. That is almost always worth doing before accepting what off-the-shelf gives you.

Every one of those architectural decisions has a running cost. Most teams estimate those reasonably well. What catches people off guard is what happens when one of the decisions has to change later, which is what the next section is about.

Understanding what it costs

A RAG system has a few different cost buckets, and they behave differently. The one that surprises most teams is reindexing, so I will start there. Almost no one has this cost conversation before deployment. When you change the chunking strategy, you reindex. When you add a metadata field, you reindex. When you switch embedding models, you reindex. A full reindex of a corpus with a few million chunks takes the better part of a day, depending on load. Not a catastrophic number by itself. The first year can easily bring half a dozen full reindexes. The surprise is never the direct cost. It is everything that surrounds the rebuild. Storage doubles while the old and new indexes are both live. Writes that land mid-reindex are at risk of dropping on the floor unless you have dual-write or a write-ahead queue set up. Blue-green swaps handle this cleanly once you build the plumbing, and most teams don't get that plumbing right on the first try.

Reducing reindexing frequency comes down to one thing: designing the schema before you start indexing. Every field you add later costs you a reindex. Any change to the chunking strategy costs you a reindex. Upgrading the embedding model costs you yet another reindex. If you spend a week planning the schema up front, including the fields you think you'll want six months out, you might avoid three reindexes in the first year. At most, a day each, that adds up to several days of reindex windows you did not have to absorb.

The patterns in this chapter have quite different cost profiles, and it's worth considering them before you commit to anything. Hybrid search is mostly a storage cost. You're looking at roughly 2x compared to keyword-only since you keep both the keyword index and the vector representations side by side, but almost nothing gets added at query time. The metadata filters, temporal decay, and access control list (ACL) pre-filtering all fall in this bucket, and cost essentially nothing either way. They run before retrieval starts, at near-zero cost. Query expansion costs a bit more, around $0.001 per query at rates as of early 2026, since the LLM has to reformulate the question before retrieval runs. Trivial at most volumes, but worth watching at scale. Multi-hop retrieval is the expensive one. Each hop is a full retrieval call plus an LLM inference for the decomposition and reasoning on top of it, which means three hops end up costing roughly three times the retrieval spend and three to four times the LLM inference spend of a single retrieval run.

The practical rule is to reach for the cheap patterns first. Hybrid search and temporal decay together address most common retrieval failures at negligible marginal cost. Multi-hop is powerful but worth deferring until the simpler patterns are in place and you have measured how often the complex case actually occurs in your system.

Establishing governance

Before you go to production, check that each source's access controls survive ingestion and appear as filters at query time. The search index doesn't know who's asking. A manager-only HR document and a general HR policy are indistinguishable to the retriever unless you tagged them differently when you indexed them and then filter on that tag at query time.

For the HR case and anything like it, I tag each chunk with whatever authorization metadata the source already has. Group IDs, classification labels, owner user IDs, whatever the source system actually uses. At query time, I apply those tags as a pre-filter on the index using the identity of whoever is asking, and I never run the filter after retrieval. Running it after means the retriever was willing to pull the forbidden document into the candidate set, and only a last-mile check kept it out of the response. That is not access control. It is hope.

One wrinkle with indexed authorization metadata is that it goes stale the moment the source system changes a permission. Someone gets removed from a group today, and the index still thinks they have access until the next reindex, which might be a day or a week later. I hit this problem directly while running RAG at scale for an internal enterprise knowledge base. In that deployment, I layered a live authorization call on top of the indexed filter. The filter does the cheap pre-screen. For anything that passes, a short-lived check against the source system's authorization API confirms the user still has access, cached for a minute or so, so the latency cost is negligible on average. Most of the time, the cache answers. When it does not, the one extra API call is still cheaper than shipping a document to a user who lost access twenty minutes ago.

For multi-tenant systems, the best practice for 2026 has converged on two shapes. Strong isolation per tenant with a separate index each, for cases where the cost of cross-tenant leakage is high. Shared index with per-query

tenant filters, for cases where it is not. Both show up in production, and both are defensible. For permission models that depend on relationships rather than labels, like a manager being able to read their reports' records but not peer records, people have been reaching for Zanzibar-inspired systems like SpiceDB and OpenFGA. A static metadata tag cannot express "user X has this access if and only if user Y reports to them," and that gap is what the relationship-based authorization systems solve.

Beyond access control, a couple of other governance concerns tend to surface right before teams go live. Retention is one of them. Documents have different useful lifespans, and a security advisory about a vulnerability patched three years ago is actively harmful to leave retrievable today because someone might follow it, while an architecture decision from the same period could still be accurate. The way to handle it without manually reviewing every document is to put an `expires_at` field on content types with known lifespans at ingestion time, and rely on the `superseded` flag for content that ages out when its replacement is indexed. Source deletion needs its own path on top of retention. A document deleted at the source has to leave the index within the same SLA as a retention expiry, or the index will keep serving content the source has already removed.

The second is personally identifiable information (PII) detection at ingestion. The chunking pipeline runs before anyone reviews the stored content. If a wiki page contains a customer email address, or a ticket comment mentions personally identifiable information, that content ends up in the index and in every retrieval result that surfaces it. PII detection is harder to add after the fact than access control because you cannot retroactively scrub embedded chunks without a full reindex. Build it into the ingestion pipeline before you go to production.

I was slow to add audit logging in my own setup. Every retrieval the system performs, and every LLM response, needs to be loggable back to the user who asked, the query they sent, the chunks that came back, and the time it happened. That is non-negotiable in regulated industries. It is also the only way to answer "which users saw which documents" after something goes wrong. Get it in at ingestion and query time. Adding it after an incident is somewhere between painful and impossible, which is usually the moment you realize you should have had it.

Routing and self-correcting retrieval

Routing is basically the question of which retrieval strategy to use for a given query, and the honest answer is it depends on what the query is actually asking for.

Queries that contain identifiers should go to keyword search first. Ticket numbers, error codes, version strings. BM25 finds the exact thing with near-perfect precision, and vector search just finds semantically similar tickets, which isn't useful when the user already knows which specific ticket they want. For questions about concepts, processes, or what happened in the past, hybrid retrieval works better with vector weighted higher. And for questions that need information from multiple documents or systems at once, that's when multi-hop with query decomposition makes sense.

The routing heuristic I keep coming back to is simple. Before retrieval runs at all, check whether the query contains something that looks like an identifier. Version-formatted strings are the most obvious case. Numeric codes with known prefixes (error codes, ticket numbers, that sort of thing) also count, along with alphanumeric IDs that match a recognizable pattern. Anything that matches one of those routes straight to BM25. Everything else goes to hybrid. I implemented the whole thing as a simple regex check, which cleared out most of the identifier retrieval failures right away without requiring any changes to the underlying index.

For self-correcting patterns, the most practical option for production systems is **CRAG**, **Corrective RAG**. After retrieval, score the relevance of the retrieved documents before passing them to the LLM. If the relevance score falls below a threshold, fall back to a broader search or, in some cases, a web search rather than asking the LLM to synthesize from low-quality context. I use CRAG for queries about external technologies and third-party APIs that the internal knowledge base does not cover well. A question about a vendor API change is better answered by a current web search than by a stale internal document written two years before the API was updated.

Two other research patterns are worth knowing about, though I'd consider both experimental relative to CRAG. **Adaptive RAG**, published in 2024, adds a routing layer that classifies query complexity before retrieval: simple factual questions go to single-step retrieval while multi-part questions get routed to

a pipeline with intermediate reasoning steps in between. **Self-RAG** is structured quite differently: rather than routing, it trains the model itself to judge whether retrieval is even necessary and to score what it gets back before generating. Benchmark results on both look good, but as of early 2026, neither has anywhere near the production track record that CRAG has, which matters a lot when operational reliability is a requirement.

RAPTOR addresses a different problem entirely: retrieval from long documents where the answer requires understanding the whole document, not just a specific chunk. It builds a recursive summary tree of the corpus and retrieves at multiple levels of abstraction. Useful for legal and research corpora. For the enterprise use case that most of this chapter covers, CRAG and query routing are the right starting points.

In practice, the sequence that worked for me was pretty unglamorous. I started with hybrid search and let it bake for a while. Nothing else got added until the hybrid implementation was stable in production, because everything I wanted to build on top either depended on it or just assumed it was already there. Once I had some confidence, I added temporal decay. That was a low-risk change since decay is just a metadata filter, no index rebuild required, nothing retrained on the embedding side. Identifier routing came in after that, and if I had to pick a single change that gave me the most return relative to the effort, identifier routing would probably be the one. The regex classifier only took about a day to build and deploy, and it cleaned up most of the identifier-retrieval complaints in the first week. CRAG was the last of the patterns that went into the production system. It catches the long tail of queries where the internal knowledge base doesn't have a good answer, and falls back to a broader search. That combination covers most of what breaks in a production retrieval system. Multi-hop retrieval and the more experimental patterns I mentioned earlier are worth revisiting once you have data on where your baseline is still falling short.

The failures in this chapter share a common origin, even though they look different on the surface. All of them surface at query time, but the real causes stem from decisions made during ingestion. Sometimes it was a schema that was missing date fields I needed later. Other times, it was a chunker running the same settings on structured Jira tickets as on narrative Confluence prose, which never should have been the same job. And in one case, it was just an

index with no place to store an entity link connecting it to the next document in the chain. Whatever the specific failure was, the decision that caused it was made before the first query ever ran, and it came back as a retrieval failure months later. Changing retrieval logic is usually straightforward. Changing the data layer underneath is a much bigger lift, and that's the real reason to spend the extra week on schema design up front. Reindexing the corpus six months later because you skipped that planning step is almost always the more expensive option in the end.

Retrieving code

Code is yet another kind of content, and character-based splitting is particularly bad for it. You really want to chunk at function and class boundaries instead. Here's why: a 300-line function chunked at 2,000 characters breaks mid-logic, and the fragments that come out of that split have no real meaning without the surrounding context. Treating the same function as a single retrievable unit, even when it runs long, gives the LLM concrete context to work with when it has to reason about the code. This is part of why agentic code search outperforms indexed code retrieval for anything complex. The unit of meaning in code is a function or a class, not some arbitrary character count. None of this makes vector search wrong. Cursor still ships an embedding index alongside its agent mode, and most of the time it helps. Honest read in 2026 is that agentic search is the backbone for code, and embeddings show up where the agent needs a fast semantic shortcut. Numbers back this up loosely. On the original SWE-bench, a simple RAG baseline landed near 2 percent, the first agentic system hit roughly 12 percent on the same benchmark, and Claude Fable 5 was reported at 95 percent on SWE-bench Verified in June 2026, with no vector DB in the loop. Best data strategy for code retrieval in 2026 might be almost no vector retrieval infrastructure at all, which is a legitimate option the book should flag before anyone spends a quarter building one. The retrieval pattern that does work for code is the code knowledge graph in a graph database, which is the evolution of the markdown knowledge graph I started a while back.

The thing that surprised a lot of teams adopting Claude Code in 2025 was that it did not index anything. People expected at least an embedded copy of the repo sitting in some local vector DB. There wasn't one. Boris Cherny, the

engineer who built Claude Code at Anthropic, has written up why publicly more than once: the team tried RAG with a local vector DB early on and abandoned it; agentic search worked better in their testing and came with fewer of the headaches the RAG stack was creating around staleness and privacy. Retrieval is grep and file reads inside an agent loop; the loop keeps querying the filesystem until the model has enough to answer. At Clearwater Analytics, we started by treating AI as a junior developer paired with explicit collaboration rules in markdown files, but the next jump came from growing agents from junior to senior using knowledge graphs, which the rest of this section walks through.

The reason this matters for a data strategy book is that it changes what you ought to be building in the first place. Code has structure that embeddings tend to blur. If you search for where `authenticate()` is defined, you want the exact function back, not three semantically similar chunks that happen to mention the word. Grep handles that kind of question exactly and cheaply, and it's always fresh because there is no index to go stale. An embedding index is a snapshot from the last time the index reindexed, so asking it about code that changed this morning gets you yesterday's version, with no signal that anything moved.

Using GitLab APIs is common in enterprises, which works well. However, creating the code graph is the way to go. Very few enterprises apply this concept. I chose to build a knowledge graph for the entire codebase, which made designing new features easy. During the indexing phase, the crawling agents crawl all the repositories and create the graph. The mechanics are simpler than they sound. A code graph is just an abstract syntax tree, or AST, extended into a graph of relationships: each function, class, and file becomes a node, and every import, call, and inheritance link becomes an edge. Query the graph for every caller of a function, or every file that depends on a given type, and you get an exact list back. That kind of answer does not come from vector similarity over file-level embeddings, where the retriever can only hand you things that look lexically or semantically similar to the query.

This was a niche idea a year or two ago. By 2026, it is showing up as a real pattern in the research. Chinthareddy's January 2026 paper on Java codebases (arXiv 2601.08773, `https://arxiv.org/abs/2601.08773`) compared vector-only RAG against a deterministic AST-derived approach using tree-sitter, and the

tree-sitter approach built the graph in seconds while staying cheap, and it produced the highest correctness on architectural queries. The LLM-generated knowledge graph approach tested in the same paper was much slower, cost more, and missed 377 files on the Shopizer test corpus. Vector-only posted the worst correctness with the highest hallucination risk on those same queries. GitNexus, an open-source code graph tool, hit number one on GitHub trending on April 10, 2026, with more than a thousand stars in a single day. The pattern is moving from niche to mainstream, with multiple commercial and research efforts in 2026 publishing or open-sourcing code knowledge graph implementations. The code knowledge graph is the natural next step for the markdown knowledge graph approach I introduced publicly in January 2026, extending the same graph that links documents to architectural decisions to meetings into code itself. At Clearwater Analytics, we are moving in that direction, and the data-architecture decisions that go into building one (which entities, which edges, how to refresh) are the same questions this chapter has been asking about other content types. Barnett (2026) describes a parallel-agent retrieval system at Clearwater Analytics that analyzes microservice codebases automatically and generates architecture diagrams at scale.

Building a knowledge graph that mixes both code and documentation has been the advancement I have made to the system and published publicly. It gives the AI agents more knowledge to act more like senior engineers. I added all the knowledge about each project in its own knowledge graph and made the project agent aware of all decisions; even meetings were added to the knowledge graph to be cited as evidence. This started as a markdown knowledge base, just `.md` files with explicit navigation and a tier system telling the AI which documents to trust. I wrote the earlier version up on *Medium*. Since then, the setup has moved to proper Graph RAG running on a graph database. Graph RAG walks the edges of a knowledge graph when it retrieves instead of pulling top-k similar chunks from a vector index, so the LLM gets the architecture decision record (ADR), the meeting where the decision was made, and the code that implements it as one connected neighborhood. I implemented this in production, and I have a public write-up on the engineering blog on Medium at `https://medium.com/cwan-engineering/building-an-ai-powered-markdown-knowledge-base-system-for-`

your-engineering-team-4bccea3cdbfe. When a coding agent asks why the team picked a given pattern, it can walk the graph all the way from the code back to the meeting note, which is what a senior engineer carries in their head.

An AI agent is a junior until its knowledge is senior. Rolling Claude Code out with shared `CLAUDE.md` files and runbooks got the model to behave like someone in their first month on the team, which was useful but not enough. The real jump came from a dedicated agent per project, each with its own knowledge graph and a few narrow sub-agents underneath for review and research. The graph carried more than the code. It picked up the meeting where the design was argued out, whatever ADR came out of the discussion, and the commit that shipped the thing, usually with links back to both so the chain was obvious. If a regression turned up in production two weeks later, the postmortem ended up in the same graph and hung off the same thread. A new engineer or a coding agent asking the AI agent why the code looks the way it looks and gets the full chain back the way a senior teammate would pull it from memory. Lilian Weng framed the shape of an AI agent back in 2023 as LLM plus memory plus planning plus tool use, and 2026 surveys expanded the list to include knowledge. Of those four pieces, only knowledge still needs real data work. Everything this chapter has been about (the indexes and the graph and the retention rules and the access controls) is the piece that turns the junior into a senior.

Comparing the four use cases

The examples in this chapter all came from one enterprise assistant, support engineers searching across roughly a hundred thousand Confluence chunks and a few million Jira chunks, so for that kind of system the chunking, index architecture, and hybrid search sections matter most, and the architecture in the second half is close to a direct fit. If you are building for a regulated domain like legal or healthcare, the access control and governance sections are the ones to reread, because the boundary stops being team membership and becomes attorney-client privilege or the Health Insurance Portability and Accountability Act (HIPAA), and a superseded regulation surfacing as current is a compliance problem rather than just an inconvenience. For a personal assistant, most of the indexing and chunking work in this chapter does not apply because the data lives behind APIs rather than in a search index, but the

governance and retention rules still hold. Content moderation maps most directly to the temporal drift and keyword exact match sections, since policy versioning is temporal drift at higher velocity and adversarial evasion is the extreme version of identifiers that embeddings cannot distinguish.

Summary

In this chapter, I walked through the production failure patterns that show up once teams move past *Chapter 4*'s Naive RAG, and the architecture decisions each one pushes them toward. We started with temporal drift, where old documents outrank newer ones, fixed by adding temporal metadata at ingestion and filtering on it at retrieval. Keyword exact match retrieval for identifiers like error codes and ticket numbers comes next, added through hybrid search. Adding a second source like Jira surfaces volume and schema pressures that shape the data architecture choices in the second half, plus the multi-hop case where a single search cannot pull every piece of an answer from a chain. Code retrieval turned out to be its own story, because in 2026 the industry has largely walked away from indexing code at all. In the second half of the chapter, I walked through the data architecture that avoids these failures from the start: chunking that matches the shape of each source, hybrid search inside every index, access control enforced at both ingestion and query time, retention with expires_at fields, and the audit logging most teams wish they had turned on earlier.

The technology in this area is moving fast, but the data architecture stays longer. *Chapter 4* will not age because it shows you the basics of RAG and helps you to make choices. This chapter shows you the real failures from real production data. While new RAG technologies will evolve, the basics discussed in both chapters will stay useful and will help you to make the right choices with any advancement. The code retrieval space moved especially fast through 2025 and 2026. RepoGraph at ICLR 2025 made repository-level code graphs an academic state of the art. The Code Graph Model paper at NeurIPS 2025 pushed that further to 43 percent on SWE-bench Lite by integrating the graph into the LLM itself. By April 2026, GitNexus and Graphify shipped open-source and proved that graph-based code retrieval works in production at the token-cost tier readers actually care about. The patterns in this chapter were written with that trajectory already visible. For day-to-day tracking, I

write daily on LinkedIn in my AI Solutions Architect newsletter (https://www.linkedin.com/build-relation/newsletter-follow?entityUrn=7144440575503609856), where I post patterns from production as I encounter them and the latest advancements in the AI domain.

In the next chapter, we move from retrieving data to generating it, with synthetic data generation for fine-tuning and coverage gaps.

References and further reading

What follows is grouped by topic, in roughly the order each subject came up in the chapter. I kept each list to the few sources I went back to most while writing, rather than everything I read.

- **Temporal decay and freshness**
  - Solving Freshness in RAG: A Simple Recency Prior. arXiv:2509.19376. https://arxiv.org/abs/2509.19376
- **Chunking strategies**
  - Anthropic. (2024, September 19). *Introducing Contextual Retrieval*. https://www.anthropic.com/news/contextual-retrieval.
  - Günther, M., et al. (2024). *Late Chunking: Contextual Chunk Embeddings Using Long-Context Embedding Models*. Jina AI. https://openreview.net/pdf?id=74QmBTV0Zf.
  - *Reconstructing Context: Evaluating Advanced Chunking Strategies for Retrieval-Augmented Generation*. (2025, April). arXiv:2504.19754. https://arxiv.org/html/2504.19754v1.
- **Code retrieval and knowledge graphs**
  - Chinthareddy, M. R. (2026, January). *Reliable Graph-RAG for Codebases: AST-Derived Graphs vs LLM-Extracted Knowledge Graphs*. arXiv:2601.08773. https://arxiv.org/abs/2601.08773.
  - Ouyang, S., et al. (2025). RepoGraph: *Enhancing AI Software Engineering with Repository-Level Code Graph*. ICLR 2025. arXiv:2410.14684. https://arxiv.org/abs/2410.14684.
  - Tao, X., et al. (2025). Code Graph Model (CGM): *A Graph-Integrated Large Language Model for Repository-Level Software Engineering Tasks*.

NeurIPS 2025. arXiv:2505.16901. https://arxiv.org/abs/2505.16901.

- **Agent architecture and agentic search**
  - Anthropic. *Effective context engineering for AI agents*. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents.
  - *Agentic Artificial Intelligence: Architectures, Taxonomies, and Evaluation of Large Language Model Agents*. (2026). arXiv:2601.12560. https://arxiv.org/html/2601.12560v1.
  - Weng, L. (2023, June 23). *LLM Powered Autonomous Agents*. Lil'Log. https://lilianweng.github.io/posts/2023-06-23-agent/.
- **Embeddings and benchmarks**
  - *Massive Text Embedding Benchmark (MTEB) Leaderboard*. Hugging Face. https://huggingface.co/spaces/mteb/leaderboard.
  - Thakur, N., et al. (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*.
- **Hybrid search**
  - Elastic. *A Comprehensive Hybrid Search Guide*. https://www.elastic.co/what-is/hybrid-search.
  - *Hybrid Retrieval for Enterprise RAG: When to Use BM25, Vectors, or Both*. RagAboutIt. https://ragaboutit.com/hybrid-retrieval-for-enterprise-rag-when-to-use-bm25-vectors-or-both/.
- **Graph RAG**
  - Microsoft. GraphRAG. https://microsoft.github.io/graphrag/.
  - Neo4j. What Is GraphRAG? https://neo4j.com/blog/genai/what-is-graphrag/.
- **Access control, governance, and multi-tenant RAG**
  - Amazon Web Services. *Authorizing access to data with RAG implementations*. https://aws.amazon.com/blogs/security/authorizing-access-to-data-with-rag-implementations/.
  - Microsoft. *Design a Secure Multitenant RAG Inferencing Solution. Azure Architecture Center*. https://learn.microsoft.com/en-

us/azure/architecture/ai-ml/guide/secure-multitenant-rag.
    - Pinecone. RAG with Access Control. https://www.pinecone.io/learn/rag-access-control/.
- **SWE-bench and code evaluation**
    - *SWE-bench Leaderboards*. https://www.swebench.com/.
    - Vals AI. *SWE-bench*. https://www.vals.ai/benchmarks/swebench.
- **Cursor and IDE retrieval**
    - *How Cursor Actually Indexes Your Codebase*. Towards Data Science. https://towardsdatascience.com/how-cursor-actually-indexes-your-codebase/.
    - Cursor. *Securely indexing large codebases*. https://cursor.com/blog/secure-codebase-indexing.
- **Claude Code and Anthropic's agentic approach**
    - Anthropic. *Claude Code Documentation*. https://code.claude.com/docs/en/best-practices.
    - Vadim. *Claude Code Doesn't Index Your Codebase. Here's What It Does Instead*. https://vadim.blog/claude-code-no-indexing.
    - SmartScope. *Settling the RAG Debate: Why Claude Code Dropped Vector DB-Based RAG*. https://smartscope.blog/en/ai-development/practices/rag-debate-agentic-search-code-exploration/.
- **Clearwater blog sources**
    - Christianson, E. (2025, November 24). *Vibe Check: Software Engineering with AI. CWAN Engineering / Medium*. https://medium.com/cwan-engineering/vibe-check-software-engineering-with-ai-41d9808b802c
    - Christianson, E. (2025, December 12). *Beyond Vibe Coding: Software Engineering with AI in Practice. CWAN Engineering / Medium*. https://medium.com/cwan-engineering/beyond-vibe-coding-software-engineering-with-ai-in-practice-29075461a527.
    - Barnett, K. (2026, March 13). *Scaling Architecture Diagramming with Parallel Agents and Feedback-Driven Prompting*. CWAN Engineering / Medium. https://medium.com/cwan-engineering/scaling-

architecture-diagramming-with-parallel-agents-and-feedback-driven-prompting-717c61aaa1b3.

- **Author-published sources**
  - Elhousieny, R. (2024, July 1). *Retrieval-Augmented Generation (RAG) with LangChain: Refining the Future of AI Conversations*. LinkedIn Pulse. https://www.linkedin.com/pulse/retrieval-augmented-generation-rag-langchain-refining-rany-adgmc/.
  - Elhousieny, R. (2026, January). *How Knowledge Graphs Transform GenAI Code Assistants from Guessing to Knowing. LinkedIn Pulse.* https://www.linkedin.com/pulse/copy-how-knowledge-graphs-transform-genai-code-from-rany-jml7c.
  - Elhousieny, R. (2026, April). *Why Your AI Coding Assistant Needs a Markdown Knowledge Base* (Not Just a Better Prompt). Medium/gitconnected. https://medium.com/gitconnected/why-your-ai-coding-assistant-needs-a-markdown-knowledge-base-not-just-a-better-prompt-31fb45b694ac.
  - Elhousieny, R. (2026, April 24). *Building an AI-Powered Markdown Knowledge Base System for Your Engineering Team*. CWAN Engineering Blog (Medium). https://medium.com/cwan-engineering/building-an-ai-powered-markdown-knowledge-base-system-for-your-engineering-team-4bccea3cdbfe.
- **Staying current**
  - Anthropic Engineering Blog. https://www.anthropic.com/engineering
  - Hugging Face Papers Daily. https://huggingface.co/papers
  - ArXiv cs.IR (Information Retrieval). https://arxiv.org/list/cs.IR/recent

Part 3

Shaping LLM Behavior with Data

This part explores how data shapes the behavior of LLM-powered systems. You will learn to generate synthetic data that addresses coverage gaps and supports training and evaluation. You will then examine post-training techniques, their data requirements, and when they are justified after improving inference-time controls. Finally, you will bring these techniques into a system-level customization workflow that combines prompts, retrieval, tools, and model adaptation where needed. By the end of this part, you will be able to connect application failures to the data and customization methods most likely to address them.

This part of the book includes the following chapters:

- *Chapter 6, Synthetic Data Generation*
- *Chapter 7, Understanding LLM Post Training*
- *Chapter 8, LLM Model Customization*

6

Synthetic Data Generation

The evolution of **Large Language Models** (**LLMs**) has always been inseparable from the data that powers them. From early experiments in natural language processing that relied on small, hand-crafted corpora to today's trillion-parameter models trained on massive multilingual datasets, data has consistently been the key driver of progress. Yet as models grow more capable, the demand for high-quality, diverse, and representative training data has outpaced what can be realistically collected from the real world.

Real-world data collection faces significant challenges, such as high collection or acquisition costs, slow annotation cycles, privacy and security risks, especially in sensitive domains such as healthcare and finance. Additionally, real data does not cover underserved languages. Human labeling introduces inconsistencies and bias, especially when different sources provide the datasets or different vendors label them. These challenges create a gap between the available real-world data and what is required for LLM systems.

Synthetic data helps close this gap by artificially generating training examples. Researchers and engineers can skip the slowness associated with manual data collection, simulate rare or unsafe scenarios, and extend coverage into domains where real-world examples are insufficient or inaccessible. Synthetic data has become a cornerstone in modern LLM development. It is not a replacement for real data, but a strategic complement that fills critical gaps and accelerates innovation.

In this chapter, you will explore techniques for creating synthetic data to supplement real-world datasets and address data scarcity. You will learn the motivations and practical scenarios that drive adoption of synthetic data, how to generate question–answer pairs that expand coverage for evaluation and

training, how to use structured data to create rich, natural language examples, and techniques for producing preference alignment data for fine-tuning models toward helpfulness and clarity.

Additionally, you'll explore data augmentation strategies that improve robustness against real-world noise and extend synthetic generation to multimodal contexts, including images and audio. Finally, you will explore techniques for integrating automated data generation with human oversight for quality assurance.

By the end of this chapter, you will understand why synthetic data is not just a workaround. You will see it as a critical enabler for building scalable, trustworthy, and adaptable LLM applications.

What is synthetic data

The term *synthetic data* refers to **artificially generated datasets** designed to resemble real-world examples. In the context of LLM development, this includes prompted text, paraphrased sentences, simulated conversations, and even multimodal pairs, such as text–image or text–audio. Unlike raw data scraped from the web or collected from enterprise systems, synthetic data is created intentionally, guided by specific goals, such as improving coverage, reducing bias, or enhancing robustness.

Why synthetic data has become essential

As LLMs scale, the limitations of real-world data become increasingly problematic. Real data has these key limitations:

- **Scarcity**: The web contains a lot of data, but it overrepresents the dominant domains and languages. This overrepresentation leaves niche domains and underserved languages without sufficient examples for LLM training and evaluation.
- **Privacy Constraints**: Data for sensitive domains, such as healthcare, finance, and human resources (HR) are not exposed on the web or in licensed datasets and cannot be exposed without privacy and compliance risks.
- **Cost and Speed**: Manual data collection and annotation cycles are slow and expensive, making it impractical to keep pace with model

development and evaluation needs.

- **Bias and Drift**: Human-labeled datasets include inconsistencies due to different labeling contexts and data vendors, and they fail to adapt to evolving language and business patterns without massive re-labeling projects.

Synthetic data generation addresses these challenges in more than one way. Synthetic data solves **scarcity** as you can tailor your data generation process to specific domains and languages. It improves **privacy** posture and reduces exposure to sensitive source data by creating examples without giving annotators direct access to the original records. However, this protection is not absolute. Teams should minimize sensitive inputs and check generated outputs for personal and confidential information before using them. Synthetic data addresses **cost and speed** by generating millions of data examples on demand at low cost. Finally, it provides solutions for **bias and drift** because you can easily re-create the generated data with unified and refreshed instructions.

Visualizing the role of synthetic data

The following diagram illustrates how synthetic and real data complement each other in LLM training, highlighting their overlapping roles in coverage and robustness:

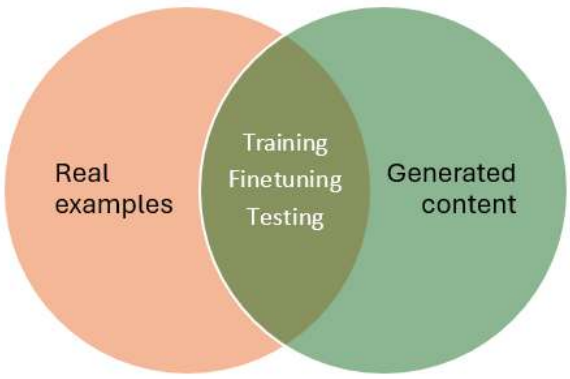


Figure 6.1: Real and synthetic data serving similar purposes: LLM training, testing, and fine-tuning

Enterprise example: HR policy queries

Here is how synthetic data helps an enterprise assistant handle nuanced HR policy questions without exposing sensitive employee information.

Consider an enterprise knowledge assistant designed to answer HR policy questions. Real-world queries about nuanced scenarios – such as maternity leave exceptions or cross-border employment rules – are rare and often buried in confidential employee communications. Synthetic generation can create hundreds of realistic variations without exposing personal data, ensuring the assistant is prepared for edge cases while maintaining compliance.

Synthetic data as a strategic complement

Synthetic data does not replace real-world data – it **complements it**. Real data provides authenticity and grounding, while synthetic data fills gaps, accelerates iteration, and enables controlled experimentation. Together, they form a balanced ecosystem for training and evaluating LLMs.

Think of it this way: real-world data gives your model a sense of reality, while synthetic data gives it resilience and adaptability. For example, real-world customer support logs teach an enterprise assistant how users phrase questions, while synthetic variations of those logs prepare the assistant for edge cases, multilingual queries, and tone adjustments.

This synergy is especially critical in high-stakes domains where errors can have serious consequences. In healthcare, for instance, synthetic question–answer pairs (QA pairs) for rare conditions ensure that a medical assistant can respond accurately even when real-world examples are scarce.

Domain example: content moderation challenges

In content moderation, synthetic data plays a critical role in preparing systems for harmful or rare scenarios without exposing moderators to distressing material.

Consider a platform moderation system tasked with detecting harmful content across text, images, and audio. Real-world examples of policy violations are limited and often sensitive. Synthetic generation can create controlled variations of offensive language, image overlays, and audio distortions – allowing the system to learn without exposing moderators to harmful material. This approach improves coverage, reduces human risk, and accelerates model readiness for real-world deployment.

Visualizing complementary roles in training

The diagram shows how real data forms the foundation for authenticity while synthetic data overlays additional layers for coverage, robustness, and alignment. Together, they form a complementary foundation for training, testing, and fine-tuning LLMs.



Figure 6.2: Relationship between real-world and synthetic data in LLM development

With this foundation in place, you can now explore practical techniques for generating synthetic data, starting with one of the most widely used methods: **Question-answer generation**.

Question–answer generation

Question–answer (QA) datasets are the backbone of LLM training and evaluation. They provide a structured way to measure understanding, reasoning, and factual accuracy. However, creating large, high-quality QA datasets manually is expensive and time-consuming. Synthetic QA generation solves this problem by automating the creation of realistic question–answer pairs at scale.

Why QA generation matters

For the most popular LLM applications, such as personal assistants and enterprise bots, users interact with the application by asking questions and expect precise context-aware answers to their questions. QA datasets directly serve this purpose by contributing to both training and evaluation. Training data teaches the model how to respond accurately and concisely, while evaluation data benchmarks the application performance under different conditions.

Real-world QA data is limited, especially for specialized high-value domains. For example, a medical assistant might encounter only a handful of queries about a rare genetic disorder in real-world logs. Synthetic QA generation can expand this to thousands of examples, ensuring the model is robust and reliable.

Techniques for synthetic QA generation

There are several proven approaches to creating QA pairs synthetically:

- **Prompt-Based generation**
  Start with a small set of seed prompts and use an LLM to generate diverse question–answer pairs. For example, a seed prompt, such as *"Explain the leave policy for new parents"*, can produce multiple variations:
  - "What is the maternity leave duration for employees?"
  - "Does the company offer paternity leave?"
  - "Are adoption-related leaves covered under HR policy?"
- **Coverage expansion**
  Generate examples for rare entities or long-tail phenomena. This approach is especially useful in specialized domains, such as medicine or law, where certain topics appear infrequently in real-world data.
- **Few-shot bootstrapping**
  Provide a handful of labeled examples and let the model generalize. This technique leverages the LLM's ability to infer patterns from minimal input, making it ideal for domains where annotated data is scarce.

The following diagram illustrates the end-to-end process of QA generation, starting from seed prompts and expanding into a large-scale dataset through model generation and human review.



Figure 6.3: QA generation pipeline with human review

Enterprise example: HR policy FAQs

QA generation helps enterprise assistants create comprehensive HR FAQs without exposing sensitive employee data. Consider the problem of answering HR policy questions. Production logs might contain a limited number of queries about maternity leaves and remote work exceptions. Synthetic QA generation can create hundreds of realistic examples of these questions and their answers based on documented HR policies and a few seed real examples. The questions will take different forms such as:

- "What is the maternity leave duration for full-time employees?"
- "Does the company offer paternity leave for adoptive parents?"
- "Are remote employees eligible for relocation benefits?"

These examples ensure that the assistant is prepared for nuanced questions in production time without exposing confidential employee data at training time.

Domain example: medical knowledge expansion

In healthcare, synthetic QA generation can improve coverage for rare conditions that are underrepresented in historical medical records cannot provide. For example, rare genetic disorders may have too few real-world examples for training and evaluation. Synthetic generation can expand this coverage by producing QA pairs about symptoms, treatment options, and diagnostic guidelines. Before these pairs are used, qualified medical experts must verify them against current, trusted clinical sources and reject examples that cannot be supported. Here are some example questions that can be synthetically generated:

- "What are the early signs of Huntington's disease?"
- "Is genetic counseling recommended for families with Huntington's history?"
- "What medications are commonly prescribed for symptom management?"

By generating these examples synthetically, the assistant becomes more robust and reliable in practice.

Implementing QA generation in practice

The following example shows a snippet of how to generate QA pairs from HR policy text using OpenAI's API in a Jupyter notebook. You can find the full code of this example and the following examples in this chapter in the book's GitHub repository by searching for the relevant cell name under `Chapter_6_Notebook.ipynb`. Please note there is a Shared Setup cell that you must execute first, before any other cells. The shared `Setup` cell discovers and verifies an available model at runtime and stores its name in `MODEL`. The following examples reuse it:

```
def generate_qa_pairs(policy_text, policy_name, policy_id):
    """Generate QA pairs using OpenAI API"""
    prompt = f"""Generate 2 question-answer pairs for this HR policy.

Policy: {policy_name}
Text: {policy_text}

Return JSON: [{{"question": "...", "answer": "..."}}]"""

    response = client.chat.completions.create(
        model=MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7
    )

    content = response.choices[0].message.content.strip()
    if "```json" in content: content = content.split(
        "```json")[1].split("```")[0]
    qa_pairs = json.loads(content)

    for pair in qa_pairs:
        pair['source_policy'] = f"{policy_id} - {policy_name}"
    return qa_pairs

# Generate QA pairs for all policies
all_qa = []
for _, row in hr_policies.iterrows():
    qa_pairs = generate_qa_pairs(
        row['policy_text'], row['policy_name'], row['policy_id'])
    all_qa.extend(qa_pairs)

# Chapter_6_Notebook.ipynb: Implementing QA Generation in Practice
```

The expected output is a table with columns `question`, `answer`, and `source_policy`.

To make these ideas more concrete, the following table summarizes the real-world applications of QA generation discussed so far.

Technique	Assistant Type	Example Application
QA Generation	Enterprise Knowledge Bot	Expand HR FAQs for compliance queries
QA Generation	Domain-Specific Assistant	Generate rare disease questions for a medical advisor

Table 6.1: Recap of QA generation applications

So far, you have seen generation of QA pairs based on natural language documents, historical notes and FAQs. Beyond natural language resources, a large portion of enterprise knowledge is encoded in structured data sources. LLM applications are expected not to just understand structured data, but also to include structured information in generated answers.

Utilization of structured data

Structured data, such as tables, logs, and relational databases, encode very valuable information in enterprise environments. HR, CRM and financial data are mostly structured in relational databases. Operational logs and metadata from line of business applications are stored in JSON or tabular formats.

On the other hand, LLMs are designed to process and generate unstructured natural language text. Bridging this gap unlocks the potential for building intelligent assistants that can reason over enterprise data. Synthetic data generation addresses this gap by transforming structured records into realistic natural-language statements and question-answer pairs for training, evaluation and retrieval. Human review is still required to confirm that the generated text preserves the semantics of the source data.

Structured data in enterprise systems

Theoretically, structured data looks like an ideal foundation for building LLM systems, but there are challenges in consuming it in practice. The top issue is the presence of **unreliable fields**. These unreliable fields exist in the schema but are not properly populated or cannot be trusted. The key lesson is to know

that **structure does not mean authoritativeness** and that you should depend on the fields that are consistently populated and enforced through business processes. If people are continuously updating a field and critical business processes depend on it, that is a good sign to use it for synthetic data generation.

Another common issue is **silent disagreement** across different systems. Some concepts, such as *employee status, customer tier,* and *order state* might look like standardized but be interpreted and populated differently across systems. Humans may reconcile these differences mentally. For your LLM assistant, you want it to produce consistent outputs for these concepts, and thus explicit reconciliation is required for training and inference.

While working with enterprise structured data, do not ingest everything, but identify which tables, records, and fields reflect the real business semantics and use them. This differentiation determines whether you should treat a structured record as a fact or ignore it entirely.

Transforming structured data

Data transformation is the step that converts structured data into useful training and evaluation data. There are *three* transformation choices that control the transformation quality:

1. What **information** should you select for transformation?
2. Does the transformation preserve **structure**?
3. What **variations** should you generate around the original information?

First, for **information selection**, target the questions that people ask, find the records and fields that answer these questions, then filter for fields with a reasonable fill rate. Consider a simple cutoff, such as a 70% population rate or higher. Fields with low fill rates tend to inject noise.

Second, observe that a structured record is not a bag of words. It encodes **relationships, constraints, and units** that the generated sentences or QA pairs must preserve. For example, a monetary value must retain its currency code during transformation. A value such as 15 million cannot be interpreted reliably unless the output specifies USD, EUR, or another currency.

Third, **generate multiple phrasing variations** for the same facts. Variations teach the model to generalize and avoid overfitting to specific patterns. Here

are some tips to apply:

- *Ask about the same fact as a direct question and as a constraint check.*
- *Ask with and without the subject.*
- *Use short phrasing and formal phrasing.*
- *Use "how much" and "what is" forms.*

Heading into the next level of details, there are two key methods to transform structured data based on the desired natural language output being *statements* or *QAs*. **Structured data to text** generation converts records into natural language statements. **Retrieval-augmented generation** (**RAG**) pipelines index these statements and retrieve the relevant ones as context for the model at inference time. The statements also provide a baseline for question–answer generation based on grounded statements.

Structured data to QA generation builds question–answer pairs from structured records directly without producing natural language facts in the middle. It is a direct fit for AI assistants because it matches the primary user behavior, which is asking questions.

This diagram illustrates how structured enterprise data flows through transformation steps to become natural language examples and QA pairs for LLM training.



Figure 6.4: Workflow for converting structured data into natural language examples

You can look at a couple of examples that put these transformation choices and techniques in context.

Enterprise example: HR policy conversion

The first example is from the HR domain. In this example, you start with text generation followed by QA generation. Consider an HR database that stores leave policies in tabular form. One row might look like this:

```
Leave_Type: Maternity | Duration: 16 weeks | Eligibility: Full-time
                                    employees
```

The text generation technique would transform this record into a natural language statement such as:

- *"Full-time employees are eligible for 16 weeks of maternity leave."*

From this sentence, you can generate multiple QA pairs as follows:

- *"How long is maternity leave for full-time employees?"*
- *"Who qualifies for maternity leave benefits?"*

These examples teach your LLM how to answer nuanced variations of the same question in a consistent way. Note how the relationship between *full-time employees* and *maternity* leave type was preserved and that *weeks* as a unit was also included. A generation process that drops eligibility constraints can over-generalize the policy and produce incorrect claims such as "*all workers are eligible for maternity leave.*"

Enterprise example: financial data transformation

For this example, consider sample pieces of information from a quarterly financial report and apply both text generation and QA generation. Financial information is usually scoped by time, currency, and aggregation level. Losing any of these details during transformation produces wrong answers, even if they look ok on the surface.

Here is an example record for quarterly performance:

```
Quarter: Q2 | Year: 2024 | Revenue: $15M | Region: Europe | Currency: USD
```

A naive transformation might generate a sentence such as:

- "*The company reported 15 million in revenue from Europe.*"

This statement reads fluently but omits the *quarter* information and assumes a default currency. For financial assistants, those omissions are errors. A structure-preserving transformation would produce:

- *"In Q2 2024, the company reported $15 million USD in revenue from the European region."*

From this sentence, you can generate multiple QA pairs such as:

- *"What was the company's revenue in Europe in Q2 2024?"*

- *"What was the revenue for Q2 2024 in Europe?"*
- *"Which region contributed $15 million in Q2 2024?"*

This approach ensures that your financial assistant can handle structured financial data with precision.

Building synthetic QA from structured data

The following example demonstrates how to generate QA pairs from HR and financial tables using Python and OpenAI's API:

```
# Sample structured data: Employee and financial records
employee_data = pd.DataFrame({
    'employee_id': ['E001', 'E002', 'E003'],
    'name': ['Alice Johnson', 'Bob Smith', 'Carol Williams'],
    'department': ['Engineering', 'Marketing', 'Finance'],
    'salary': [95000, 72000, 88000],
    'years_employed': [3, 5, 2]
})

financial_data = pd.DataFrame({
    'quarter': ['Q1 2024', 'Q2 2024', 'Q3 2024'],
    'revenue': [2500000, 2800000, 3100000],
    'expenses': [1800000, 2000000, 2200000],
    'profit_margin': [28.0, 28.6, 29.0]
})

def create_context_sentence(row, table_name):
    """Create descriptive sentence from table row"""
    if table_name == 'employee':
        return (
            f"{row['name']} works in "
            f"{row['department']} with a salary of"
            f"${row['salary']:,} and has been employed "
            f"for {row['years_employed']} years."
        )
    else:  # financial
        return (
            f"In {row['quarter']}, revenue was"
            f"${row['revenue']:,}, expenses were"
            f"${row['expenses']:,}, and profit margin"
            f"was {row['profit_margin']}%."
        )

def generate_qa_from_table(context, source_table):
    """Generate QA pairs from structured data context"""
    prompt = f"""Generate 2 question-answer pairs based on this data.
Context: {context}
```

```
Return ONLY valid JSON in this exact format (no extra text):
[{{"question": "...", "answer": "..."}}]"""

    response = client.chat.completions.create(
        model=MODEL,
        messages=[
            {"role": "system",
             "content": (
                 "You are a JSON generator. Always return valid JSON"
                 "arrays only, with no additional text or markdown."}),
            {"role": "user", "content": prompt}
        ],
        temperature=0.7
    )

    content = response.choices[0].message.content.strip()

    # Extract JSON from markdown code blocks if present
    if "```json" in content:
        content = content.split("```json")[1].split("```")[0].strip()
    elif "```" in content:
        content = content.split("```")[1].split("```")[0].strip()

    # Find JSON array in the content
    import re
    json_match = re.search(r'\[.*\]', content, re.DOTALL)
    if json_match:
        content = json_match.group(0)

    qa_pairs = json.loads(content)

    for pair in qa_pairs:
        pair['source_table'] = source_table
        pair['context'] = context
    return qa_pairs

# Generate QA pairs from employee data
all_structured_qa = []
for _, row in employee_data.iterrows():
    context = create_context_sentence(row, 'employee')
    qa_pairs = generate_qa_from_table(context, 'employee_data')
    all_structured_qa.extend(qa_pairs)

# Generate QA pairs from financial data
for _, row in financial_data.iterrows():
    context = create_context_sentence(row, 'financial')
    qa_pairs = generate_qa_from_table(context, 'financial_data')
    all_structured_qa.extend(qa_pairs)

# Chapter_6_Notebook.ipynb: Building Synthetic QA from Structured Data
```

Expected output is a table with columns `question`, `answer`, `source_table` and context.

To make these ideas more concrete, the following table summarizes how structured data usage applies to different assistant types and their practical scenarios:

Technique	Assistant Type	Example Application
Structured Data → Text	Enterprise Knowledge Bot	Convert HR tables into natural language statements
Structured Data → QA	Domain-Specific Assistant	Generate financial Q&A from quarterly reports

Table 6.2: Mapping structured data transformation techniques to applications

Preference alignment data

Accuracy alone is not enough for modern LLMs. Users expect responses that are not only correct but also **helpful, clear, and aligned with human preferences**. A legal assistant that answers with vague disclaimers or a personal assistant that responds in an overly formal tone can frustrate users – even if the information is technically correct. Preference **alignment data** is useful here.

Preference alignment ensures that models learn to produce outputs that match desired qualities, such as tone, helpfulness, and clarity. Traditionally, this is achieved through **reinforcement learning from human feedback (RLHF)**, where humans rank multiple outputs to train a reward model. However, doing RLHF at scale is costly and time-consuming. Synthetic generation can significantly reduce manual effort by creating **preferred versus non-preferred response pairs** automatically. Human evaluators review these pairs to check them against preference and safety criteria.

Synthetic preference alignment in action

The process involves generating two responses for the same instruction:

- **Preferred response**: High-quality, helpful, and aligned with guidelines.

- **Non-preferred Response**: Intentionally degraded – too short, verbose, or hedged.

These pairs are then used in policy optimization methods, such as **direct preference optimization (DPO)** to teach the model which behaviors to favor.

For example:

- Instruction: *"Summarize the company's parental leave policy."*
- Preferred: *"Full-time employees are eligible for 16 weeks of maternity leave and eight weeks of paternity leave."*
- Non-preferred: *"Employees have leave options."*

By training on these pairs, the model learns to prioritize clarity and completeness over vague or incomplete answers.

The following diagram illustrates how instructions are transformed into pairs of preferred and non-preferred responses. Pairs are labeled and used in reinforcement learning to align model behavior with human expectations.



Figure 6.5: Workflow of creating and consuming preference pairs

Enterprise example: aligning HR responses for clarity

Here is how preference alignment ensures that an enterprise assistant provides clear, helpful answers to HR queries instead of vague responses.

An enterprise assistant answering HR queries must balance accuracy with tone. For instance, when asked: *"Explain the company's parental leave policy,"* a preferred response might be:

"Full-time employees are eligible for 16 weeks of maternity leave and eight weeks of paternity leave. Adoption-related leave is also covered."

A non-preferred response could be: *"Employees have leave options."*

Both are factually correct, but the first is clear and helpful, while the second is vague and uninformative. Training on such pairs teaches the model to prioritize clarity and completeness.

Domain example: moderation messages with the right tone

In content moderation, synthetic preference pairs help bots enforce policies firmly yet constructively, avoiding abrupt or hostile language.

Moderation systems must enforce policies without sounding hostile or ambiguous. For example:

Instruction: *"Respond to a user who posted prohibited content."*

Preferred: *"Your post violates our community guidelines on hate speech. Please review the policy and edit your content to comply."*

Non-preferred: *"This is not allowed."*

The preferred response is firm yet constructive, while the non-preferred one lacks guidance. Synthetic generation of these pairs helps moderation bots maintain professionalism and consistency.

Generating preference pairs programmatically

The following example demonstrates how to create preferred versus non-preferred responses for a set of instructions using OpenAI's API in a Jupyter notebook:

```
def generate_preferred_response(instruction):
    """Generate a high-quality, helpful response"""
    prompt = f"""Provide a clear, professional, and helpful response to this question:
{instruction}

Be specific, accurate, and include relevant details."""

    response = client.chat.completions.create(
        model=MODEL,
        messages=[
            {"role": "system",
             "content": (
                 "You are a helpful, professional HR assistant."
                 "Provide clear and accurate information."}),
            {"role": "user", "content": prompt}
        ],
        temperature=0.3
    )
    return response.choices[0].message.content.strip()
```

```
def generate_non_preferred_response(instruction):
    """Generate a lower-quality or degraded response"""
    prompt = f"""Provide a vague, unhelpful, or overly brief response to this question:
{instruction}

Make it unclear, missing details, or too generic."""

    response = client.chat.completions.create(
        model=MODEL,
        messages=[
            {"role": "system",
             "content": (
                 "Provide minimal,"
                 "vague responses without helpful details."}),
            {"role": "user", "content": prompt}
        ],
        temperature=0.7
    )
    return response.choices[0].message.content.strip()

# Generate preference pairs
preference_pairs = []
for instruction in instructions:
    preferred = generate_preferred_response(instruction)
    non_preferred = generate_non_preferred_response(instruction)

    preference_pairs.append({
        'instruction': instruction,
        'preferred_response': preferred,
        'non_preferred_response': non_preferred
    })

# Chapter_6_Notebook.ipynb: Generating Preference Pairs Programmatically
```

Expected output is a dataset with multiple columns, including `instruction`, `preferred_response`, and `non_preferred_response`.

To make these ideas more concrete, the following table summarizes how preference alignment applies to different assistant types and their practical scenarios:

Technique	Assistant Type	Example Application
Synthetic Preference Pairs	Enterprise Knowledge Bot	Align tone and clarity in HR policy responses

Technique	Assistant Type	Example Application
Synthetic Preference Pairs	Content Moderation System	Generate constructive enforcement messages

Table 6.3: mapping preference alignment techniques to applications

Data augmentation for robustness

Real-world data is rarely perfect. Typos, speech recognition errors, and optical character recognition (OCR) distortions can easily derail a model's performance. These imperfections are common in enterprise environments where scanned documents, multilingual inputs, and voice commands coexist. If a model has only seen clean, well-structured text during training, it will struggle when faced with noisy inputs. Robustness matters, and synthetic augmentation is a critical technique.

Data augmentation introduces controlled variations into training corpora, simulating the kinds of errors and diversity that occur in practice. By exposing models to these variations during training, you help them generalize better and remain accurate under less-than-ideal conditions.

Techniques for Robustness

Common augmentation strategies for exposing LLMs to controlled input variations include:

- **Paraphrasing** generates multiple semantically equivalent versions of a sentence. For example, it rewrites this original sentence "Employees must complete security training annually." into a paraphrased version "Annual security training is mandatory for all employees."
- **Back-translation** translates text into another language and back to diversify phrasing. For instance, English → French → English often introduces subtle variations that improve generalization.
- **Noise injection** adds typos, phonetic confusions, or OCR artifacts to simulate real-world errors.
- Example: *"Leave policy"* → *"Levae policy"* or *"L3ave policy."*

You can apply these techniques individually or combine them to create a rich, varied dataset that strengthens model resilience.

The following diagram illustrates how paraphrasing, back-translation, and noise injection work together to create a resilient dataset for LLM training.

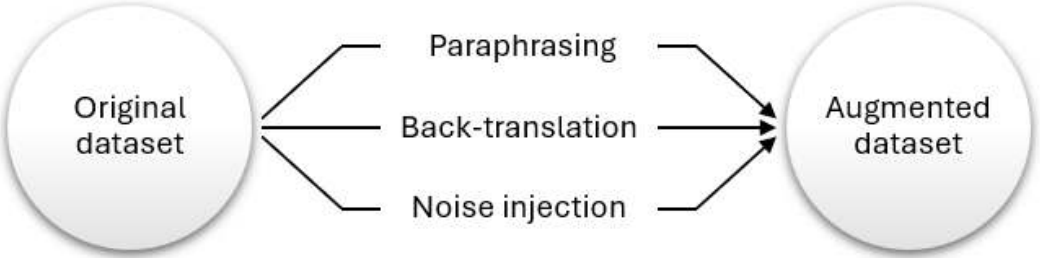


Figure 6.6: Common augmentation techniques

Enterprise example: overcoming OCR challenges

Here is how synthetic augmentation helps enterprise assistants remain accurate when OCR introduces distortions in scanned documents.

Enterprise assistants often process scanned documents, such as contracts or compliance reports. OCR systems can introduce distortions, such as misreading "ll" as "I1" or inserting random spaces. Synthetic augmentation simulates these errors by injecting controlled noise into clean text. For example:

Original: *"Employees must complete security training annually."*

Augmented: *"EmpIoyees must compIete secur1ty tra1ning annuaIIy."*

Training on such variations ensures the assistant remains accurate even when document quality is poor.

Personal assistant example: handling speech-to-text errors

In voice-based assistants, synthetic augmentation prepares models to handle transcription mistakes caused by phonetic similarities. Voice commands are prone to transcription mistakes. For example, when the user says, *"end session"*, the model may receive the command as *"and session"* from the upstream automated speech recognition component due to the phonetic similarity between the two phrases in English. Synthetic augmentation can

generate these variants systematically, preparing the model to handle incoming audio imperfections without confusion.

Implementing text augmentation techniques

The following example demonstrates how to apply paraphrasing, back-translation, and noise injection using OpenAI API in a Jupyter notebook:

```
def generate_paraphrase(text):
    """Generate paraphrase using OpenAI API (fast and high-quality)"""
    prompt = f"Paraphrase this text while keeping the same meaning: {text}"

    response = client.chat.completions.create(
        model=MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7,
        max_tokens=150
    )
    return response.choices[0].message.content.strip()

def back_translate(text):
    """Perform back-translation:
       English -> French -> English (using OpenAI API)"""
    try:
        # Translate to French
        french_response = client.chat.completions.create(
            model=MODEL,
            messages=[{
                "role": "user",
                "content": f"Translate this English text to French: {text}"
            }],
            temperature=0.3,
            max_tokens=150
        )
        french_text = french_response.choices[0].message.content.strip()

        # Translate back to English
        english_response = client.chat.completions.create(
            model=MODEL,
            messages=[{
                "role": "user",
                "content": f"Translate this French text to English: {french_text}"
            }],
            temperature=0.3,
            max_tokens=150
        )
        return english_response.choices[0].message.content.strip()
    except Exception as e:
        print(f"Back-translation error: {e}")
```

```
        return text

def inject_noise(text, noise_level=0.05):
    """Inject typos and OCR-like distortions"""
    chars = list(text)
    num_changes = max(1, int(len(chars) * noise_level))

    # Common OCR and typing errors
    substitutions = {
        'o': '0', 'l': '1', 'i': '1', 'e': '3', 'a': '@',
        's': '5', 'b': '6', 't': '7', 'g': '9', 'O': '0'
    }

    for _ in range(num_changes):
        if len(chars) == 0:
            break
        idx = random.randint(0, len(chars) - 1)

        # Apply different noise types
        noise_type = random.choice(['substitute', 'delete', 'swap', 'ocr'])

        if noise_type == 'substitute' and chars[idx].isalpha():
            chars[idx] = random.choice('abcdefghijklmnopqrstuvwxyz')
        elif noise_type == 'delete' and len(chars) > 1:
            chars.pop(idx)
        elif noise_type == 'swap' and idx < len(chars) - 1:
            chars[idx], chars[idx + 1] = chars[idx + 1], chars[idx]
        elif noise_type == 'ocr' and chars[idx] in substitutions:
            chars[idx] = substitutions[chars[idx]]

    return ''.join(chars)

# Chapter_6_Notebook.ipynb: Implementing Text Augmentation Techniques
```

Expected output is a dataset with multiple columns including `original_text`, `augmented_text` and `augmentation_type`.

The following table summarizes practical applications of augmentation strategies for different types of assistants and their real-world benefits.

Technique	Assistant Type	Example Application
Paraphrasing	Enterprise Knowledge Bot	Generate multiple versions of compliance statements
Noise Injection	Personal Productivity Bot	Simulate speech-to-text errors for voice commands

Table 6.4: Mapping augmentation techniques to applications

Multimodal data augmentation

As LLMs evolve into multimodal systems, they no longer process text alone – they interpret and generate across images, audio, and even video. This shift introduces new challenges: how do you ensure models are robust when handling diverse modalities? Real-world multimodal datasets are expensive to collect and often limited in coverage. Synthetic augmentation provides a practical solution by creating controlled variations that expand training data without compromising quality.

Why multimodal augmentation matters

Consider a content moderation system tasked with detecting harmful content in images and videos. Real-world examples of policy violations are scarce and sensitive, so they require strict handling controls. Teams can apply automated transformations to a policy-approved set of access-restricted seed images. These transformations include scaling, rotation, adding text overlays, adding noise and other transformations that modify the image content without changing their labels. They help the model recognize harmful content under different conditions. Source and derived images should remain in access-controlled storage, retain their policy labels and provenance, and pass safety checks before training. Human review should be limited to authorized reviewers trained to handle sensitive content.

Similarly, voice-based assistants must handle variations in pitch, tone, and background noise. Synthetic audio augmentation can simulate these conditions, ensuring the assistant performs reliably in noisy environments or across diverse accents.

Techniques for multimodal augmentation

Different modalities require different augmentation techniques.

- **Image augmentation** applies transformations, such as auto-scaling, rotation, flipping, and noise injection. For example, you can rotate an image used for moderation training 90 degrees or blur it slightly to simulate real-world distortions.

- **Audio augmentation** adjusts pitch, tone, and style and adds background noise or reverberation. These variations prepare speech-enabled assistants for real-world acoustic challenges:
- **Cross-modal pairing** generates captions for synthetic images or transcripts for audio clips. This technique enriches multimodal datasets by linking visual or auditory content with descriptive text.

Domain example: content moderation across modalities

The following steps show how controlled image and audio transformations create labeled variations for helping the moderation system recognize the same policy violations under different distortions:

- Start with an image containing offensive text.
- Apply transformations, such as rotation, scaling, and color adjustments.
- Overlay additional text or introduce noise to simulate real-world distortions.

Similarly, for audio:

- Take a clean voice sample and adjust pitch or tone.
- Add background noise or reverberation to mimic challenging acoustic environments.

These controlled variations broaden coverage while limiting additional moderator exposure to sensitive source material.

Building multimodal augmentation workflows

The following example demonstrates how to apply image augmentation using Python libraries and generate captions for visual data for multimodal training:

```
# Image augmentation functions
def rotate_image(image_path, angle=45):
    """Rotate image by specified angle"""
    img = Image.open(image_path)
    rotated = img.rotate(angle, expand=True, fillcolor=(255, 255, 255))
    output_path = image_path.replace(
        'sample_images', 'augmented_images'
    ).replace('.jpg', '_rotated.jpg')
    rotated.save(output_path)
```

```
    return output_path

def flip_image(image_path):
    """Flip image horizontally"""
    img = Image.open(image_path)
    flipped = img.transpose(Image.FLIP_LEFT_RIGHT)
    output_path = image_path.replace(
        'sample_images', 'augmented_images'
    ).replace('.jpg', '_flipped.jpg')
    flipped.save(output_path)
    return output_path

def add_gaussian_noise_image(image_path):
    """Add Gaussian noise to image using PIL and numpy"""
    img = Image.open(image_path)
    img_array = np.array(img)
```

```
    noise = np.random.normal(0, 25, img_array.shape)
    noisy_img_array = np.clip(
        img_array.astype(np.float32) + noise, 0, 255
    ).astype(np.uint8)
    noisy_img = Image.fromarray(noisy_img_array)
    output_path = image_path.replace(
        'sample_images', 'augmented_images'
    ).replace('.jpg', '_noisy.jpg')
    noisy_img.save(output_path)
    return output_path

def generate_image_caption(image_path, augmentation_type):
    """Generate caption for image using OpenAI API"""
    # Since we can't send actual images in this demo,
    # we'll generate descriptive captions
    prompt = f"""Generate a short, descriptive caption for a sample image
                 that has been augmented using {augmentation_type}.
             The caption should be 1-2 sentences describing
                 what might be visible in the image."""

    response = client.chat.completions.create(
        model=MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7,
        max_tokens=100
    )
    return response.choices[0].message.content.strip()

# Chapter_6_Notebook.ipynb: Building Multimodal Augmentation Workflows
```

Expected output is a dataset with multiple columns: `image_path`, `caption`, and `augmentation_type`.

To make these ideas more concrete, the following table summarizes how multimodal augmentation applies to different assistant types and their practical scenarios:

Technique	Assistant Type	Example Application
Image Augmentation	Content Moderation System	Generate rotated and noise-injected offensive images
Audio Augmentation	Personal Productivity Bot	Simulate noisy environments for voice commands

Table 6.5: mapping multimodal augmentation techniques to applications

Human-in-the-loop strategies

Synthetic data can scale quickly, but automation alone is not enough. Quality matters – and in many cases, human judgment is irreplaceable. Tasks involving nuanced interpretation, ethical considerations, or domain-specific expertise require oversight that only humans can provide. Human-in-the-loop strategies are essential for building reliable LLM systems.

HITL approaches combine the speed and scalability of synthetic generation with the precision and contextual understanding of human review. Instead of choosing between automation and manual curation, you use both in a complementary workflow.

Why human oversight matters

- **Nuance and ethics**: Models might struggle with ambiguous or sensitive topics, such as legal compliance or medical advice.
- **Gold-standard quality**: Evaluation datasets often require the highest level of accuracy, which automated systems cannot guarantee.
- **Continuous improvement**: Human feedback helps refine prompts, improve generation quality, and identify weak areas in the model.

Hybrid approaches

Several patterns have emerged for integrating humans into synthetic data workflows:

- **Synthetic first, human refined**: Models generate large volumes of data, and humans review a small subset for quality assurance. This approach is cost-effective and accelerates iteration.
- **Human first, model expanded**: Humans create a small set of "golden" examples, and models generate variations based on these seeds. The seeds provide a quality target, but they do not guarantee that every variation preserves it. Generated examples should be validated by human reviewers and filtered against quality criteria before use.
- **Self-curation loops**: Models iteratively refine their own generations, guided by human feedback at key checkpoints. This technique is common in advanced RLHF pipelines.

Visualizing human-in-the-loop workflows

The following diagram illustrates how human review integrates with synthetic generation in a feedback loop that improves data quality and model performance.

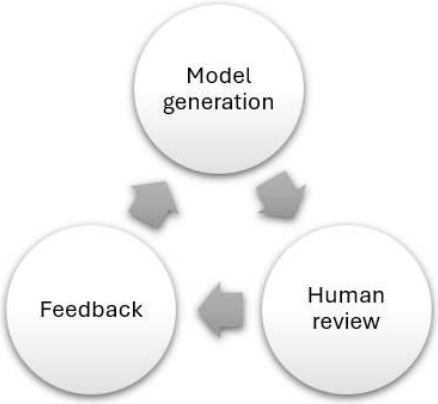


Figure 6.7: human-in-the-loop workflow

Enterprise example: legal review for compliance Q&A

Here is how human oversight supports accuracy and clarity in compliance-related Q&A pairs generated synthetically. Imagine an enterprise assistant designed to answer compliance-related questions. Synthetic generation can produce thousands of Q&A pairs from policy documents, but tone and interpretation matter. A legal team might review a subset of these pairs selected through both random and targeted sampling. The random sampling

may reveal unexpected errors, while targeted sampling covers high-risk questions and major policy areas. Reviewers then verify that:

- Responses are accurate and reflect current regulations.
- Language is clear and avoids ambiguity.
- Sensitive topics are handled with appropriate caution.

This hybrid approach combines the efficiency of automation with the assurance of expert oversight, creating a dataset that is both scalable and trustworthy.

The following table summarizes how different human-in-the-loop strategies apply to various assistant types and their practical benefits.

Approach	Assistant type	Example application
Synthetic first, human refined	Enterprise knowledge bot	Legal team reviews compliance Q&A for accuracy
Human first, model expanded	Domain-specific assistant	Experts create golden examples for medical queries

Table 6.6: mapping human-in-the-loop approaches to applications

Summary

Synthetic data has evolved from a niche technique into a **strategic pillar** of modern LLM development. Throughout this chapter, you explored why real-world data alone cannot meet the demands of today's large-scale models and how synthetic generation bridges that gap.

You learned how to:

- Generate **question–answer pairs** to expand coverage for evaluation and training.
- **Leverage structured data – s**uch as HR tables and financial reports – to create rich, natural language examples.
- Produce **preference alignment data** that teaches models to prioritize clarity, helpfulness, and tone.
- Apply **augmentation, such as** paraphrasing, back-translation, and noise injection to improve robustness.

- Extend synthetic generation to **multimodal contexts**, including images and audio.
- Combine automation with **human oversight** for quality assurance through human-in-the-loop workflows.

These techniques are not just about filling gaps. They enable scalability, improve resilience, and ensure alignment with human expectations. Synthetic data complements real-world datasets, accelerates iteration, and prepares models for edge cases that matter in practice.

As you move forward, remember this key insight: **synthetic data is not a shortcut. It is a deliberate strategy for building trustworthy, adaptable, and high-performing LLM applications.**

In the next chapter, we will build on this foundation by exploring **post-training techniques**. You will learn when and why to go beyond retrieval and synthetic generation and how to fine-tune models for specialized domains. You will also learn how reinforcement learning from human feedback (RLHF) and other alignment methods transform raw capability into real-world reliability.

References

- *Can synthetic data help to keep people safe online?* (n.d.). The Alan Turing Institute. https://www.turing.ac.uk/blog/can-synthetic-data-help-keep-people-safe-online
- Clyburn, C., & Chen, C. (2025, November 25). *Synthetic data: A secret ingredient for better language models*. Red Hat Blog. https://www.redhat.com/en/blog/synthetic-data-secret-ingredient-better-language-models
- Groves, I. (2024, July 12). *Automatically generating text from structured data - Amazon Science*. Amazon Science. https://www.amazon.science/blog/automatically-generating-text-from-structured-data
- Hassani, H. S. (2025, January 7). *OpenAI's Preference Fine-Tuning: A Guide With Examples*. https://www.datacamp.com/tutorial/preference-fine-tuning

- Hughes, A. (2024, May 30). *The crossroads of innovation and privacy: Private synthetic data for generative AI - Microsoft Research*. Microsoft Research. https://www.microsoft.com/en-us/research/blog/the-crossroads-of-innovation-and-privacy-private-synthetic-data-for-generative-ai/
- Mrbullwinkle. (n.d.). *Direct preference optimization - Azure OpenAI*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/fine-tuning-direct-preference-optimization
- Upasham, R., Bhattacharyya, P., & Department of Computer Science and Engineering, Indian Institute of Technology Bombay. (2021). *Table to Text Generation: A survey of datasets and techniques*. In Department of Computer Science and Engineering, Indian Institute of Technology Bombay [Journal-article]. https://www.cfilt.iitb.ac.in/resources/surveys/2025/survey_tab2txt_ronak.pdf
- WUJEK, B. & SAS Institute Inc. (2024). *Best practices for ensuring quality of synthetic data* [Report]. https://www.sas.com/content/dam/SAS/documents/infographics/2024/why-synthetic-data-is-essential-for-your-organizations-ai-driven-future-113845.pdf
- [x]cube LABS. (2024, September 13). *Data augmentation Strategies for training Generative AI models*. [X]Cube LABS. https://www.xcubelabs.com/blog/data-augmentation-strategies-for-training-robust-generative-ai-models/

7

Understanding LLM Post Training

Teams that use post training effectively do not usually use it as the first solution. They usually try to control the inference first by improving prompts, refining instructions, adding retrieval, and analyzing failures carefully. Before doing post training, you want to understand whether the model cannot do the task or whether the system is not providing enough structure and context for the model to behave correctly.

This order matters because changing model weights is expensive and difficult to reverse. Prompt updates and **retrieval-augmented generation** (**RAG**) pipelines are system components that you can inspect and fix independently. Weight updates are more complex because their impact is harder to isolate and reason about. Changing weights is the most aggressive control in an LLM system. For this reason, **you should treat post training as an escalation step** rather than the default option. It is useful when you try strong inference-time controls and failures persist.

In this chapter, you will explore the major post-training techniques used to customize LLM behavior and learn when to use each technique. We will compare inference-time controls with weight updates, understand the different types of training data required for **Supervised Fine-tuning** (**SFT**), **Low-Rank Adaptation** (**Lora**), preference alignment, and reinforcement learning, and work through practical examples that illustrate applying these techniques in modern LLM systems. Finally, you will evaluate post-training results and integrate post training into continuous learning workflows.

You will learn the following topics:

- Understanding the role of post training in relationship to inference-time controls

- Using popular inference-time controls before post training
- Applying supervised fine-tuning to stabilize model behavior
- Adapting models with **Low-Rank Adaptation** (**LoRA**)
- Optimizing model behavior with preference alignment and reinforcement learning
- Managing data scarcity, hidden disagreement, and overfitting
- Integrating post training into continuous learning loops

Technical requirements

This chapter builds on the shared development environment introduced in *Chapter 1*. If you have completed the setup in *Chapter 1*, no additional software installation is required. The examples in this chapter use the same Python environment and Jupyter notebooks introduced earlier. This chapter introduces the fundamentals of LLM post-training and assumes only a basic familiarity with LLM applications and RAG from the previous chapters. No prior experience with fine-tuning, preference alignment, or reinforcement learning is necessary. To follow the examples in this chapter, you should have:

- Basic knowledge of RAG, synthetic data generation, and the AI assistant architecture introduced in Chapters 1–6 to understand the chapter
- Access to the book's GitHub repository and the accompanying Jupyter notebooks to follow the notebook examples
- The development environment configured as described in *Chapter 1* to run the notebook examples

The practical examples in this chapter focus on post-training data formats and workflows rather than running full-scale fine-tuning jobs. The accompanying Python notebook demonstrates the structure of SFT, LoRA, preference-alignment, and reinforcement-learning datasets, along with the relevant API calls and configurations. Running large-scale post-training experiments is optional and might require additional GPU resources or cloud-based training services depending on the selected framework and model. The source code and Jupyter notebook for this chapter are available in the book's GitHub repository.

The role of post training

Post training exists to change the model's default behavior. Post training does not exist to compensate for weak retrieval, stale information, or missing context. Weak retrieval, stale information, and missing context are usually system-level problems rather than model-weight problems. Using post training to hide them often creates systems that look better during narrow evaluations while becoming less reliable under distribution shift.

Inference-time controls operate outside the model itself. Prompts, system instructions, retrieval pipelines, few-shot examples, and tools guide the model during runtime and remain relatively easy to inspect and iterate on. When these controls fail, the failure is usually visible in the inputs and surrounding context provided to the model.

Post training operates differently because it changes how the model behaves internally across prompts and contexts. This internal change allows the model to become more behaviorally consistent, but it also makes failures significantly harder to diagnose. Once weights change, behavior becomes *implicit* rather than directly observable through prompts and runtime context alone.

This difference explains how mature teams approach model-quality improvements. They usually begin with inference-time controls because these mechanisms are easier to test, cheaper to modify, and simpler to reverse. Over time, some failures continue to appear even after repeated improvements to prompts, retrieval quality, and tools. These failures are usually behavioral. The model may ignore instructions inconsistently, violate response constraints, or produce unstable outputs despite having the correct information already available in context.

A practical diagnostic is straightforward. If the model already has the right information and still produces the wrong behavior, inference-time controls are probably reaching their limits.

This also explains one of the most common mistakes in LLM systems. Teams sometimes apply fine-tuning to compensate for retrieval instability or incomplete context. The resulting model may appear stronger during controlled evaluations, but these improvements usually fail once prompts, workflows, or data distributions change. For example, a model fine-tuned on a company's current policies and terminology might perform well during

internal testing. However, it can become less effective when users adopt new workflows, document formats, or business policies that the training data does not represent. The system becomes harder to reason about without becoming more reliable.

Post training should therefore improve default behaviors that remain unstable after prompting techniques including RAG and tools have already converged.

In this section, we established that post training is not a replacement for prompting, retrieval, or tools. It is an escalation step that you need only after inference-time controls have been fully explored. In the following section, you will examine the major inference-time controls.

Exhausting inference-time controls

The previous section framed post training as an escalation step. In practice, teams usually arrive there gradually. They refine prompts repeatedly, add examples, improve retrieval quality, and redesign tools. Over time, the improvements become smaller while the same categories of failures continue to appear. That plateau is usually the signal.

Inference-time controls remain attractive because they are reversible and relatively fast to iterate . Most quality improvements in production LLM systems happen at this layer first. Before changing model weights, teams should explore the major inference-time techniques carefully.

Structuring instructions for behavior

Instruction design is the primary behavior control mechanism in most LLM systems. Before retrieval, before tools, and before post training, the model still operates under instructions. Weak instruction structure creates unstable behavior even when the model already knows how to perform the task correctly.

These failures usually appear as inconsistent formatting, unstable tone, partial compliance with constraints, or different interpretations of the same request. Many teams interpret these failures as capability problems and immediately move toward fine-tuning. In practice, the issue is usually ambiguity in the instruction structure itself.

A mature instruction structure usually defines three things clearly: the role of the assistant, the constraints that the assistant must follow, and the expected output structure. The instruction itself does not need to be complicated. What matters more is whether the model interprets the task consistently across different prompts and contexts. Most instruction failures are ambiguity failures.

The following example separates the assistant's role, behavioral constraints, and expected output format into explicit parts:

```
System:
  You are an enterprise support assistant.
  You must answer using only the provided context.
  If the answer is not present, respond with "Information not available."

Task: Summarize the issue and propose next steps.

Output format:
  Issue summary:
  Proposed next steps:
```

This prompt is intentionally simple. Once instruction structure becomes mature, small wording changes should not significantly alter the model's behavior.

Using few-shot examples

Few-shot examples complement instructions by showing the expected behavior directly. They become useful when the target behavior is easier to demonstrate through examples than describe through instructions. This commonly happens with uncertainty handling, refusal behavior, response tone, and workflow-specific formatting.

One common mistake is continuously adding examples whenever the model fails. Over time, prompts become large collections of patches that introduce competing behavioral patterns and unstable expectations. For example, one few-shot example might encourage detailed explanations, while another favors concise responses. Or one example might demonstrate when to refuse a request while another answers a similar request directly. Instead of stabilizing the behavior, the examples start creating additional ambiguity. Few-shot examples work best when they reinforce one consistent behavioral pattern.

Good few-shot examples are intentionally small and representative. They establish the expected structure and decision boundaries while keeping the prompt behaviorally consistent.

```
Example 1:
  Input: Customer reports intermittent login failures.

  Output:
    Issue summary: Intermittent authentication errors.
    Proposed next steps:
      Check identity provider logs and recent configuration changes.

Example 2:
  Input: Customer asks about an undocumented internal system.

  Output:
    Issue summary: Information not available.
    Proposed next steps: Escalate to system owner.
```

At this stage, the remaining failures should mostly involve reasoning quality or missing information rather than formatting instability or tone drift.

Retrieval as a behavioral control

RAG is usually used as a mechanism for improving knowledge freshness and grounding. RAG also works for behavioral control because retrieval determines the information that the model can use at inference time.

Many teams misinterpret retrieval issues as model failures. The assistant might produce inconsistent responses simply because retrieved documents are incomplete, poorly ranked, or contradictory. Before considering post training, you should evaluate whether the retrieval system is working as intended.

No retrieval system is perfect, but strong retrieval systems degrade gracefully and fail safely. When retrieval quality decreases or completely fails, the assistant should adjust its responses based on that, rather than pretending everything is normal. For example, if retrieval returns no relevant documents, the assistant should acknowledge that it cannot answer the question rather than inventing information. If only partial evidence is available, it should answer only based on the supported portions and identify what is missing. When retrieved documents appear outdated, it should communicate that limitation or recommend consulting the latest version instead of treating stale

information as authoritative. Small context changes should not create large behavioral shifts.

This distinction matters because post training does not fix weak retrieval systems. The opposite is true. When post training is used to compensate for weak retrieval, the training data teaches the model to produce acceptable answers despite incomplete or irrelevant retrieved context. Over time, the model learns to ignore the retrieved information and rely more on its internal knowledge that the training data provided. In this case, instead of fixing the retrieval problem, post training reinforces the wrong behavior by teaching the model to ignore retrieved context. Many apparent model limitations are retrieval problems in the underlying system.

Using tools for actions and computation

Tools extend inference-time controls beyond language generation. Giving the model access to APIs, databases, calculators, and MCP servers allows the model to perform operations that language models are not designed to handle on their own.

Tool use becomes important when correctness matters. Pricing calculations, transactional operations, compliance checks, and structured lookups are usually more reliable when delegated to runtime systems rather than memorized model behavior. *The more operational the workflow becomes, the more important tools become.*

For example:

```
System:
  You can use tools exposed by MCP servers.
  For pricing, use mcp.pricing.quote.
  Never guess prices.
  If the server fails or returns an error,
  explain the failure and ask for alternatives.

MCP tool (mcp.pricing.quote):
  Inputs: sku, quantity, region
  Returns: currency, unit_price, total_price, assumptions
```

Well-designed tools simplify debugging because the inputs and outputs remain observable. Failures become easier to isolate between the model, the tool, and the surrounding system logic. Without tools, teams sometimes do

model fine-old use might become more reliable tuning to memorize workflows that should remain external to the model itself.

When is post training required?

Inference-time controls are not exhausted because every possible technique was attempted. In practice, teams reach this point when repeated improvements stop changing the outcome. Retrieval quality might get better and tool use might become more reliable, but similar behavioral failures continue to appear, because inference-time controls can improve the information and capabilities available to the model, but they do not change its default behavior.

At this stage, the issue is usually no longer the surrounding context. The model already has the correct information and still behaves incorrectly. The assistant might violate instructions inconsistently, ignore workflow constraints, or produce unstable outputs despite having sufficient runtime guidance.

A practical diagnostic is straightforward. If the model already has the right information and still makes the wrong decision, inference-time controls are probably reaching their limits.

This point is the boundary where post training becomes useful. The remaining failures are usually behavioral rather than informational. The system already has the right inputs. The issue is how the model responds to them.

The post-training spectrum

Post-training techniques often appear as interchangeable solutions. In practice, each technique changes a different aspect of model behavior and requires different assumptions about the data.

Three dimensions explain most of the practical differences between these methods: what behavior is being changed, what type of data is required, and how difficult the resulting system becomes to deploy, debug and reverse. *Table 7.1* summarizes these differences before examining each technique in more detail. Use these dimensions to identify the technique that best matches the behavior you need to change, the data you have available, and the operational complexity you can support.

Technique	Primary goal	Training data	Best suited for	Operational complexity
SFT	Stabilize default behavior	Prompt-response examples	Formatting, workflow, tone, refusal behavior	Low
LoRA	Efficiently customize model behavior	Same supervised examples as SFT	Fast experimentation, multiple model variants, rapid rollback	Low
Preference alignment	Learn preferred behaviors	Preferred vs. non-preferred response pairs	Helpfulness, safety, style, behavioral tradeoffs	Medium
Reinforcement learning	Optimize decisions through rewards	Reward signals from humans or automated graders	Multi-step reasoning, tool use, agentic workflows, outcome optimization	High

Table 7.1: Comparing major post-training techniques by goal, data requirements, use case, and operational complexity

Supervised fine-tuning (**SFT**) changes output defaults directly by showing the model examples of the desired behavior. **Low-Rank Adaptation** (**LoRA**) changes the model more selectively by constraining the weight updates to adapter layers outside the main model. Preference alignment and reinforcement learning optimize how the model chooses between competing outputs rather than teaching one exact response.

These techniques are not maturity stages. Teams do not "graduate" from one technique to another in a fixed sequence. The choice depends on the type of failure we are trying to address.

Supervised fine-tuning (SFT)

SFT teaches the model what acceptable outputs look like. SFT works best when the task is well defined and the desired behavior can be represented clearly through examples.

SFT is particularly effective for stabilizing structure, tone, formatting, workflow order, and refusal behavior. In these cases, the model usually already knows how to perform the task but behaves inconsistently across prompts and contexts.

Here is one training example in chat format.

```
{
  "messages": [
    { "role": "system",
      "content": "You are a support assistant. Use only provided context." },
    { "role": "user",
      "content": "Summarize the incident and list next steps." },
    { "role": "assistant",
      "content": "- Issue summary: ...\n- Proposed next steps: ..." }
  ]
}
```

The value of SFT is directness. You show the model the exact behavior that you want to become stable across prompts and contexts.

At the same time, SFT performs poorly when the desired behavior itself is inconsistent. If different examples encode conflicting expectations, the model learns to average them together. For example, one training example might refuse to answer a request for financial advice, while another example answers a similar request. During training time, the model does its best to fit most of the training examples. The training process will find a way to make sense of this conflict, but most probably it is not going to match what you want. The result would be a model that sometimes refuses and sometimes answers without a human-friendly explanation.

The underlying desired behavior could be to give financial advice only when retrieval pipeline finds relevant and authoritative results and otherwise refuse. To stabilize this behavior with SFT, define the behavior explicitly and take the necessary steps to make it clear to the model at training time by

providing enough training examples of each case and making sure the refusal phrasing reflects the lack of authoritative evidence.

A useful way to think about SFT is that it programs default behavior into the model. SFT works best when the target behavior is already operationally clear and internally consistent.

Low-Rank Adaptation (LoRA)

LoRA solves a different problem than SFT. The primary value of LoRA is combining learning ability with operational flexibility. Instead of updating the full model weights, LoRA freezes the base model weights and inserts small trainable adapters. This limits the blast radius of the update and makes rollback significantly easier. *The model behavior changes locally rather than globally.*

Teams choose LoRA when they want to run experimentation cycles faster or support serving of multiple model variants at the same time. In larger organizations, LoRA allows different teams to customize model behavior independently while sharing the same foundation model.

LoRA uses the same type of supervised data as SFT. Consider this LoRA data example:

```
{
  "messages": [
    { "role": "system",
      "content": "You are a support assistant. Follow the output format exactly." },
    { "role": "user",
      "content": "Classify the request and return JSON." },
    { "role": "assistant",
      "content": "{\"intent\":\"billing_issue\",\"needs_handoff\":false}" }
  ],
  "metadata": {
    "scenario": "intent_classification",
    "eval_bucket": "edge_cases",
    "source": "golden_set_v3"
  }
}
```

The `messages` part is the same as SFT. The `metadata` section is not part of the training data itself, but it helps with evaluation and isolation of failures during training experiments. The LoRA method improves a team's ability to roll back bad data additions and support deployment flexibility, but it does

not reduce the importance of data quality. LoRA limits how much it can break but does not increase correctness.

Preference Alignment and Reinforcement Learning

SFT and LoRA work well when the desired behavior has a clear definition and is consistently represented in training data responses. For many production prompts, it is difficult to define a single correct response. The model might generate multiple technically correct responses that differ in tone, safety posture, helpfulness level, or decision quality.

Preference alignment learns from comparisons between different LLM responses. Instead of asking humans to write the perfect answer, development teams capture multiple model responses for the same prompt and context and answer which response is better. Humans are usually more consistent at comparing outputs than giving gold responses.

The following is an example preference pair record with two completions and one winner:

```
{
  "input": {
    "messages": [
      { "role": "system",
        "content": "You are a policy-aware assistant. Be concise." },
      { "role": "user",
        "content": "Can I export all customer emails to a CSV?" }
    ]
  },
  "preferred_output": [
    { "role": "assistant",
      "content": "I can help. Where will the file be stored?" }
  ],
  "non_preferred_output": [
    { "role": "assistant",
      "content": "Sure. Here are the steps to export all customer emails..." }
  ]
}
```

The important distinction here is that preference data does not define one exact response. Preference data defines which behavior people prefer for the same context. This property makes preference alignment useful when multiple outputs are technically valid but qualitatively different.

Reinforcement learning extends this idea further and learns from rewards. Instead of optimizing against labeled responses, the model optimizes against rewards attached to responses based on quality or business outcomes. The reward can come from humans, automated graders, rules, or combinations of all three.

The following example shows a very simple form of RL data:

```
{
  "messages": [
    { "role": "system",
      "content": "You are a support assistant. Use the provided incident context." },
    { "role": "user",
      "content": "Explain why the deployment failed and what to do next." },
    { "role": "assistant",
      "content": "The deployment failed because ... Next steps are ..." }
  ],
  "reward": 0.8,
  "reward_rationale": "Grounded, actionable, no fabricated metrics."
}
```

The key difference is that RL optimizes behavior through outcomes rather than matching predefined target responses. For example, in a technical troubleshooting scenario, there are multiple valid ways to resolve the issue. Instead of requiring the model to produce one exact response, RL rewards behaviors that successfully complete the workflow, such as gathering the required information, selecting the correct tools, and resolving the issue without violating business policies. This approach is useful in workflows where success depends on decision quality, sequencing, or environment interactions instead of surface-level phrasing.

In many modern pipelines, rewards come from graders. The grader scores model responses during training using rule-based, model-based, or hybrid techniques. In this type of grader-based workflow, you do not need to pre-label every completion because the grader scores new completions during training. Labeled or preference-based data are still required to build, calibrate, and evaluate the grader itself.

Here is one grader-based RL data example. This grader provides a reward based on computation and lookup.

```
{
  "messages": [
```

```
    { "role": "system",
      "content": "You are a finance assistant. Provide a final numeric estimate." },
    { "role": "user",
      "content": "What is the cost of 3 VMs with 8 vCPUs and 32GB RAM in Europe." },
    { "role": "assistant",
      "content": "Estimated monthly cost is €1,150 assuming on-demand pricing." }
  ],
  "reward_spec": {
    "type": "composite",
    "components": [
      { "check": "pricing_lookup", "tolerance_percent": 10 },
      { "check": "assumption_presence" },
      { "check": "numeric_output" }
    ]
  }
}
```

RL provides greater flexibility and control over previously discussed techniques, but it introduces additional risks. Weak reward functions produce unstable optimizations and diminish the overall value. If the reward signal is noisy or easily gameable by the training algorithm, the model learns to trick the grader rather than the real objective. For example, if a reward function favors short responses but does not measure helpfulness, the model can receive higher rewards for brief but incomplete answers than for slightly longer, more useful responses. As a result, the model learns to optimize for brevity at the expense of helpfulness. Reward quality therefore becomes part of the system design itself.

Preference alignment and RL are closely related in goals and operations but different in implementation details. The good news is that teams can convert preference data into scalar rewards and reuse them inside RL pipelines, by assigning a reward of one to the preferred pair and a reward of 0 to the non-preferred pair. This practice is common in systems where RL infrastructure already exists or where teams want RL optimization on top of preference datasets.

The following is a preference data example converted into RL data:

```
{
  "messages": [
    { "role": "system", "content": "You are a safety-aware assistant." },
    { "role": "user", "content": "Can I share customer data with a vendor?" }
  ],
  "completions": [
```

```
    {
      "assistant": "Only if a data processing agreement is in place.",
      "reward": 1.0
    },
    {
      "assistant": "Yes, as long as the vendor is trusted.",
      "reward": 0.0
    }
  ]
}
```

This conversion is useful because the same preference dataset can support multiple training approaches. Many teams can start with preference comparisons due to simplicity, then reuse the same data later inside RL optimization pipelines for greater flexibility and control.

Comparing techniques in practice

At this stage, the differences between post-training techniques are getting clear. SFT changes default outputs directly by reinforcing examples of the target behavior. LoRA improves rollback flexibility and supports multiple variations in parallel. Preference alignment and RL optimize model behavior when multiple outputs are correct.

These techniques solve different categories of problems. The choice is not about the level of complexity of each approach but rather depends on the problems you are trying to solve and the availability of resources.

SFT works well when the target behavior is clear and repeatable. LoRA is useful when deployment flexibility and rollback matter. Preference alignment and RL are valuable when the problems involve tradeoffs and decision quality.

As we move across the post-training spectrum from SFT to RL, behavioral control can be more explicit and fine-grained. SFT provides control by showing the model examples of desired outputs, while RL uses explicit rewards or graders that score specific criteria and directly reinforce them during training. This additional control also increases sensitivity to the quality of the training and reward signals. More control means more responsibility. The more precisely we optimize behavior, the more important it is to ensure that we are optimizing the right behavior.

Reading the Spectrum

The post-training spectrum is not exactly a maturity ladder. Teams do not progress from SFT to LoRA to RL in a fixed order. Each technique addresses a different type of problem, and thus different techniques might be used at different times or in parallel. Each technique answers a different question. You select the technique based on the questions you want to answer.

If the question is *What should typical model responses look like?* SFT answers this question. If your concern is *how you can adjust behavior safely and reversibly,* then you need to use LoRA. When *multiple responses are possible* and you want more control, RL and preference alignment techniques help.

This framing is more useful than treating these methods as increasingly advanced versions. You should select the technique based on the problem you are trying to solve. Most teams use SFT first because it is the most straightforward well-known and supported method for adjusting model responses when inference-time optimizations do not work. Through the next sections, we will revisit SFT, LoRA and RL in detail.

Applying supervised fine-tuning

SFT is usually the first weight-updating technique teams try because it directly resets model behavior with examples. It is also well documented and supported on most AI training platforms and services. SFT makes sense when inference-time controls have already converged and the remaining failures are mostly behavioral.

SFT is not the best way to augment the model with new knowledge or capabilities. SFT stabilizes the model behavior across variable prompts and runtime conditions. This distinction matters because SFT is useful when the model knows how to perform a task but does not do it reliably across contexts or retrieval variations.

SFT is useful when the model drifts in formatting, occasionally violates constraints, or skips parts of a standard workflow while having the correct information in context. Another symptom is when your system prompt becomes longer, few-shot examples grow continuously, and model behavior changes with small prompt variations. This pattern is usually a sign that prompting alone is no longer stabilizing the behavior.

A simple diagnostic step helps. If the model has the right information and context but still makes the wrong decisions, SFT is a strong candidate solution. If the model lacks the right information entirely, SFT is usually the wrong tool.

Think of SFT as a technique for *programming default behavior*. SFT works well when the target behavior is clear and repeatable. Target behavior includes style, workflow sequence, uncertainty handling, formatting conventions, and refusal guidelines. SFT works best when the target behavior has a clear definition.

SFT is weaker for resolving tradeoffs. If the training examples encode conflicting expectations, such as "be concise" and "be thorough" or "be safe" and "be helpful," the model learns to average these guidelines. In such cases, SFT moves the conflicts from prompting to the model itself and results in a model that is not predictable on edge cases.

This table summarizes common behavioral failures and whether SFT is usually a good fit for them.

Observed failure	SFT helps?	Why?
Format drift (JSON breaks, schema violations)	Yes	SFT stabilizes output structure and formatting behavior across prompts.
Refusal inconsistency	Yes	Refusal behavior is usually a default-behavior problem rather than a retrieval problem.
Missing or stale knowledge	No	SFT does not reliably add or refresh knowledge.
Retrieval instability	No	The issue is usually retrieval quality or context assembly rather than model defaults.
Tool misuse	Sometimes	SFT helps if the misuse is workflow-related, but not if the tool design itself is weak.

Observed failure	SFT helps?	Why?
Subjective tradeoffs (helpful vs safe vs concise)	No	These failures usually require preference signals or reward-based optimization.

Table 7.2: List of common failures and SFT applicability for solving them

Most SFT failures are not because of problems in the learning algorithm or training process. They are usually because of mismatches between the target runtime behavior and the behavior encoded in the training data. The training examples implicitly define what acceptable behavior is, how to deal with uncertainty, and when the assistant should refuse a user request. SFT data becomes the behavioral contract of the system.

If the runtime expectations are different from the training examples, the model optimizes toward the wrong target. The fine-tuned model might appear stronger on narrow evaluations while becoming less predictable in production cases. SFT amplifies the behavioral contract that exists in the training data.

Good SFT data is intentionally consistent. Similar scenarios should produce similar responses in content, tone and output structure. When user questions are not clear, the model's handling of uncertainty should remain stable across examples. Refusal policy should follow a consistent pattern rather than being random.

If similar scenarios produce different behaviors in the dataset, the model learns this instability directly from the dataset. This often happens when product expectations are still evolving while development teams start fine-tuning too early. In such cases, the model learns unresolved disagreements rather than stable operational behavior.

Another common failure reason is when evaluation data resembles the training data too closely. In this case, offline metrics improve and the model appears more compliant. At the same time, nearby cases regress because the model overfits to the training examples rather than capturing the intended underlying behavior. Evaluation datasets should measure generalization rather than memorization. For example, a system trained heavily on incident summaries might apply the same structure on unrelated prompts.

Strong SFT systems usually share the same characteristics. The runtime contract is stable before training begins. The uncertainty policy is consistent. Similar scenarios produce similar responses. Evaluation datasets are independent of training data. Rollback plans exist before deployment starts. Stable behavior is the result of stable definitions rather than the training process.

The following diagram illustrates SFT contract alignment. The process starts with defining the behavioral contract, then generating training examples that match it, evaluating independently, and refining the contract as needed.

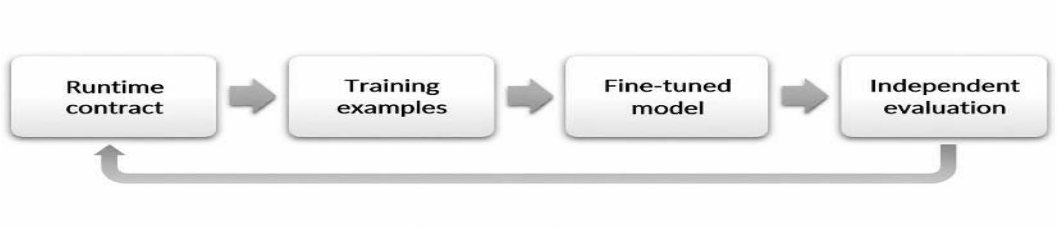


Figure 7.1: SFT succeeds when training and runtime contracts match, and progress depends on evaluation

The following example shows SFT training and validation records along with a simple fine-tuning workflow using local libraries. Because hosted fine-tuning services change over time, training locally keeps the implementation reproducible.

```
# One example from sft_train.jsonl

{
  "messages": [
    {
      "role": "system",
      "content": "You are a policy-aware support assistant. If context is insufficient, say
'Information not available' and ask a clarifying question. Output: Summary, Next steps, Risks."
    },
    {
      "role": "user",
      "content": "Export customer emails to CSV and send to vendor. Context: none."
    },
    {
      "role": "assistant",
      "content": "Summary\nInformation not available to proceed.\n\nNext steps\n- Confirm business
purpose and approval path\n- Confirm data classification and sharing policy\n- Confirm whether a data
processing agreement exists\n\nRisks\nSharing customer data without approvals and controls can
violate policy and contracts."
    }
```

```
  ]
}

---------------------------------

# One example from sft_valid.jsonl

{
  "messages": [
    {
      "role": "system",
      "content": "You are a policy-aware support assistant. If context is insufficient, say
'Information not available' and ask a clarifying question. Output: Summary, Next steps, Risks."
    },
    {
      "role": "user",
      "content": "Latency spiked after enabling new cache. Disabling cache restored performance.
Summarize and propose next steps."
    },
    {
      "role": "assistant",
      "content": "Summary\nLatency increased after enabling the cache; disabling it restored
performance.\n\nNext steps\n- Compare cache config (TTL/keying/eviction) before vs after\n- Inspect
hit rate and backend saturation during spike\n- Re-enable via canary and isolate the specific
setting\n\nRisks\nRe-enabling without isolating the setting can recreate the incident."
    }
  ]
}

------------------

# SFT code snippet

# Load pre-stored SFT datasets (10 train, 3 validation examples)
train_path = dataset_dir / "sft_train.jsonl"
valid_path = dataset_dir / "sft_valid.jsonl"

train_examples = load_jsonl(train_path)
valid_examples = load_jsonl(valid_path)

# Load a small base model for local supervised fine-tuning.
tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL)
tokenizer.pad_token = tokenizer.eos_token
model = AutoModelForCausalLM.from_pretrained(BASE_MODEL)

# Convert chat-style examples into training batches.
train_loader = DataLoader(
    ChatSFTDataset(train_examples, tokenizer),
    batch_size=2,
    shuffle=True,
```

```
)

# Run a compact supervised fine-tuning loop.
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)

model.train()
for epoch in range(3):
    total_loss = 0.0

    for batch in train_loader:
        optimizer.zero_grad()
        loss = model(**batch).loss
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    print(
        f"Epoch {epoch + 1}/3 -- train loss: {total_loss / len(train_loader):.4f}"
    )

# Save the fine-tuned model locally.
model.save_pretrained(str(SFT_MODEL_DIR))
tokenizer.save_pretrained(str(SFT_MODEL_DIR))

# Chapter_7_Notebook.ipynb: Applying Supervised Fine-Tuning
```

The training records in the preceding example encode the uncertainty policy, escalation behavior, and output structure directly into the dataset. This consistency is exactly what makes SFT work best. SFT stabilizes behavior that already has a definition and is repeatedly reinforced through examples, but it does not resolve conflicting product decisions about the target behavior.

LoRA answers a different question. *How can you change model behavior while limiting blast radius and enabling fast rollback?* If you expect frequent training iterations, multiple model variants, or quick rollback, LoRA provides an operational advantage. This is what you are covering in the next section.

Adapting models with LoRA

LoRA solves a different problem than SFT. The main value of LoRA is operational flexibility rather than learning capability. LoRA is a parameter-efficient fine-tuning technique that freezes the base model and inserts small trainable adapters into selected layers. This design allows teams to customize model behavior without updating all model weights. In practice, LoRA

reduces training cost and memory usage while making experimentation and rollback easier.

Teams usually choose LoRA for practical reasons. One reason is faster rollback because adapters can be disabled or replaced without retraining the base model. Another reason is supporting multiple behavioral variants while sharing the same foundation model. This flexibility becomes important in larger organizations where one team owns the base model while downstream teams customize behavior independently.

LoRA can be used in two ways. Some teams use LoRA instead of SFT while keeping the same supervised training data. Other teams apply LoRA adapters on top of existing models to support multiple customizations at the same time. This often happens in larger systems where different teams own different workflows or products.

Figure 7.2 illustrates how LoRA adapters attach to a shared base model while remaining detachable and independently deployable:

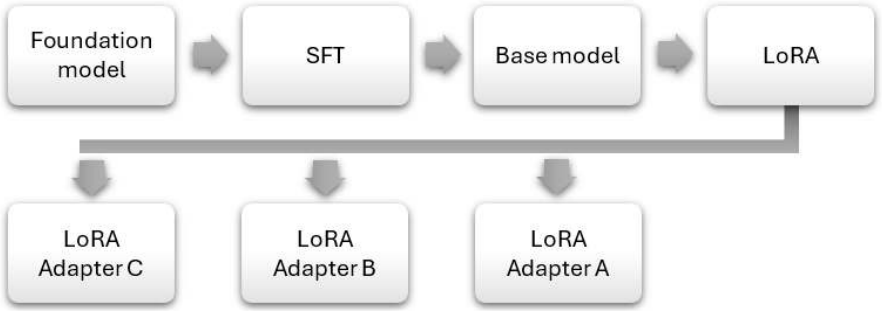


Figure 7.2: Integrating SFT and LoRA as adaptation strategies: base model plus detachable adapters

LoRA solves a deployment problem that traditional SFT does not solve well. Instead of updating the entire model, LoRA stores behavioral changes in lightweight adapters that can attach to or detach from the shared base model.

This design makes rollback much simpler. If an adapter introduces undesirable behavior, teams can disable or replace that adapter instead of retraining and redeploying the entire model.

LoRA also simplifies experimentation. Different adapters can be evaluated side by side in A/B tests, allowing teams to compare behavioral changes while keeping the underlying foundation model unchanged.

Finally, adapters help with isolating undesired changes. Because each adapter represents a localized customization, model developers can easily determine which training update introduced an undesired change, simplifying debugging and operational maintenance.

At the same time, LoRA does not reduce the importance of data quality. Poor supervision still produces unstable behavior. Small datasets still overfit. Weak behavioral definitions still create inconsistent outputs. LoRA limits how much the model changes. LoRA does not guarantee correctness.

The following example demonstrates a simple LoRA configuration and training workflow:

```
# --- Configure LoRA adapter
#     (rank, alpha, and which layers to target are explicit) ---
lora_config = LoraConfig(
    r=16,                          # low-rank dimension
    lora_alpha=32,                 # scaling factor
    target_modules=["c_attn"],     # inject into attention projection only
    lora_dropout=0.05,
    bias="none",
)

# ATTACH: wrap base model with LoRA, base weights are frozen,
# only adapter params are trained
model = get_peft_model(base_model, lora_config)
model.print_trainable_parameters()
#e.g."trainable params: 294,912 || all params: 82,206,720 || trainable%: 0.36"

# --- Train adapter to learn one fact ---
FACT = "The capital of France is Paris."
train_enc = tokenizer(FACT, return_tensors="pt")
labels = train_enc["input_ids"].clone()

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
model.train()
for step in range(50):
    optimizer.zero_grad()
    loss = model(**train_enc, labels=labels).loss
    loss.backward()
    optimizer.step()
    if (step + 1) % 10 == 0:
        print(f"Step {step + 1}/50, loss: {loss.item():.4f}")
model.eval()

prompt = tokenizer("The capital of France is", return_tensors="pt")
```

```
# Stop at "." so generation ends at the sentence boundary
stop_token_id = tokenizer.encode(".", add_special_tokens=False)[0]
gen_kwargs = dict(max_new_tokens=10, eos_token_id=stop_token_id)

# ATTACH: adapter active, model has learned the fact
print("\n=== Adapter ATTACHED ===")
with torch.no_grad():
    out = model.generate(**prompt, **gen_kwargs)
print(tokenizer.decode(out[0]))

# Chapter_7_Notebook.ipynb: Applying LoRA in practice
```

The important part of this example is not the hyperparameters themselves. The important part is the deployment model. The adapter can be attached, detached, or replaced without changing the base model itself.

LoRA improves deployment flexibility, but it also introduces additional complexity. Adapter versioning becomes important. Evaluation pipelines must track which adapter is active. Multiple adapters can also interact with each other in unexpected ways when behavioral boundaries are not clearly defined.

The following table summarizes the operational differences between SFT and LoRA:

Technique	Primary benefit	Rollback speed	Typical use case
SFT	Resets global behavior	Slow (retrain + redeploy)	Stable default behavior
LoRA as alternative	Fast iteration with smaller blast radius	Fast (disable or swap adapter)	Rapid experimentation
LoRA as overlay	Parallel variants	Very fast (per-adapter switching)	Multi-team customization

Table 7.3: Comparing SFT versus two methods of implementing LoRA

At this stage, the model's behavior is usually already correct at a high level. The remaining failures often relate to decision quality rather than formatting or workflow structure. This is where preference alignment and reinforcement learning become more useful.

Optimizing agentic behavior with RL

By the time reinforcement learning becomes useful, formatting and workflow problems are usually already solved. The remaining failures appear during execution and decision-making. The model might choose the wrong action, misuse tools, or optimize the wrong objective despite having the correct information available.

Reinforcement learning approaches this differently from SFT and LoRA. Instead of learning from target outputs directly, RL optimizes behavior using rewards attached to outcomes.

Most production RL systems for LLMs use graders. The model executes actions inside an environment, graders evaluate the result, and the resulting reward signal drives optimization. The grader can use rules-based, model-based, or hybrid techniques.

This process creates a learning loop that evaluates the quality of the model's decisions and actions, converts the grader results into rewards, and uses those rewards to update model behavior:

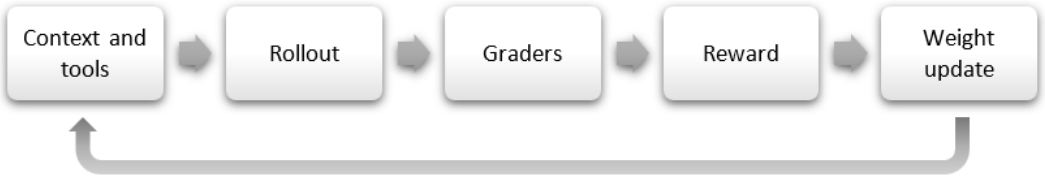


Figure 7.3: Agentic RL Implementation Loop

The grader is the most important part of the RL system. Weak graders produce unstable optimization behavior quickly. If the grader can be gamed, the model learns to optimize the grader rather than the actual task. Reward quality becomes part of the system design itself.

Strong graders usually verify final outcomes rather than surface phrasing. They check constraints explicitly, distinguish model failures from system failures, and remain stable when rerun multiple times on the same rollout.

RL training data also looks different from SFT data. SFT trains on prompt-response pairs. RL trains on rollouts that include tool calls, environment state, intermediate actions, grader outputs, and rewards.

The following example illustrates an agentic rollout with tools and reward scoring. The reward here reflects operational success rather than linguistic quality.

```
{
  "scenario": {
    "tools": ["kb.search", "ticket.create"],
    "initial_state": { "ticket_count": 0 }
  },
  "input": {
    "messages": [
      { "role": "system",
        "content": "Use tools for actions. Ask clarifying questions when needed." },
      { "role": "user",
        "content": "Create a support ticket for VPN failures affecting finance team." }
    ]
  },
  "rollout": [
    { "type": "tool_call", "name": "kb.search",
      "args": { "query": "VPN failures finance team triage" } },
    { "type": "tool_result", "name": "kb.search",
      "result": { "top_hit": "VPN outage triage checklist" } },
    { "type": "tool_call", "name": "ticket.create",
      "args": { "title": "VPN failures - Finance", "severity": "high" } },
    { "type": "tool_result", "name": "ticket.create",
      "result": { "ticket_id": "INC-12345" } },
    { "type": "final",
      "content": "Created INC-12345. Next steps: collect affected locations." }
  ],
  "graders": {
    "assertions": [
      "ticket_count_increased",
      "ticket_has_required_fields",
      "final_response_includes_next_steps"
    ]
  },
  "reward": 1
}
```

Preference methods learn from comparisons. However, you can convert preference data into scalar rewards and reuse them inside RL pipelines. Many systems start with preference comparisons, then reuse the same datasets for RL optimization. For example:

```
{
  "messages": [
    {
      "role": "system",
```

```
      "content": "You are a safety-aware assistant."
    },
    {
      "role": "user",
      "content": "Can I share customer data with a vendor?"
    }
  ],
  "completions": [
    {
      "assistant": "Only if a data processing agreement is in place.",
      "reward": 1.0
    },
    {
      "assistant": "Yes, as long as the vendor is trusted.",
      "reward": 0.0
    }
  ]
}
```

The same dataset can support both approaches. The following pseudocode illustrates a simplified RL training loop:

```
for step in range(num_training_steps):

    # 1. Sample a prompts or scenario
    scenario = sample_environment_state()

    # 2. Run the model to produce a rollout (including tool calls)
    rollout = run_model_with_tools(
        model=policy_model,
        scenario=scenario,
        max_steps=10
    )

    # 3. Grade the outcome
    grade = grader.evaluate(
        rollout=rollout,
        expected_constraints=scenario.constraints
    )

    # 4. Convert grade to reward
    reward = reward_fn(grade)

    # 5. Update the model to increase expected reward
    optimizer.step(
        model=policy_model,
        rollout=rollout,
        reward=reward
    )
```

The implementation details differ across RL frameworks, but the operational structure remains similar. The rollout must resemble production behavior, the grader must remain stable, and the reward signal must represent the intended objective.

Several RL-specific failure modes appear repeatedly in production systems. Reward misspecification causes the model to optimize the wrong behavior. Environment mismatches happen when training tools behave differently from production tools. Another common issue is uninformative rollouts where everything either passes or fails, making optimization unstable.

A safer approach to RL is to move incrementally by validating reward distributions before training, comparing results against frozen baselines, and monitoring for reward collapse throughout training cycles. These checks help identify unstable or mis-specified reward signals before they produce larger behavioral changes. RL amplifies both good reward design and bad reward design.

Managing data scarcity and overfitting

Across post-training techniques, most failures are fundamentally data problems rather than optimization problems. Some teams wrongly blame the training algorithm when the real issue is inconsistency inside the training data.

Many datasets include conflicting applications of what should be the same business rule, encode outdated policies, and surface unresolved product decisions. Adding more data amplifies these disagreements rather than fixing them and makes it even more difficult to trace them back and resolve them.

This pattern is visible in post training because the model learns behavioral patterns directly from the dataset. If data creators and reviewers disagree on the meaning of "good," the model learns to average these disagreements together. The result is usually a system that appears reasonable during evaluation while remaining unstable in production.

Hidden disagreements accumulate in post training data for a few common reasons. Different teams might label similar cases differently. Product and organization policies evolve over time while existing datasets remain static without revision. Subjective preferences are reflected in training examples

without clear and common definitions. This damages RL systems because unstable rewards lead to reward hacking behavior.

Synthetic data is used to address data sparsity, especially for edge cases and scenarios that do not appear frequently in training datasets and production logs. Synthetic data is useful when the data generation process follows stable behavioral rules and targets identified failure modes. On the other hand, synthetic data amplifies existing assumptions. If the generation process already contains hidden disagreements and inconsistent or outdated policies, synthetic generation scales those problems instead of fixing them.

Overfitting is another common post-training problem. This is where offline metrics improve and training loss decreases, but the model fails to generalize on related examples. This regression would appear later in production systems if the offline evaluation were not comprehensive enough. Typical overfitting symptoms include regressions on nearby scenarios, rigid response formatting, inconsistent handling of edge cases, and large performance gaps between examples that resemble the training data and unseen examples. Evaluation datasets should measure generalization rather than memorization.

Pay attention to these warning signs while building and running your post-training pipelines:

- Improvements appear only on training data and very similar scenarios
- Response changes are difficult to explain and connect with target behavior
- Offline evaluation metrics improve without corresponding online metric improvements

When you observe one or more of these warning signs, you should take samples from your post training data and revise them for freshness and consistency against the latest target behaviors. Teams that manage post training pipeline effectively apply repeatable operational discipline. They version the datasets explicitly and revise them frequently, keep evaluation independent of training, and trigger a dataset update cycle with every policy change. These practices resolve behavioral disagreements in the dataset and reflect positively on post-training results.

Integrating post-training into learning loops

Post training should not be treated as a one-time intervention. In production systems, models continue evolving as products, policies, workflows, and user behavior change over time.

Evaluating each training data update separately helps teams identify the specific data changes that introduced improvements or regressions. When multiple dataset changes accumulate before evaluation, it becomes harder to isolate the source of behavioral change and determine which data should be corrected or removed.

For this reason, training data should be updated and evaluated incrementally. Each step in the learning loop depends on the others. Evaluation improves deployment quality, monitoring improves data quality, and higher data quality improves evaluation results.

Continuous learning amplifies both good and bad operational disciplines. A healthy post-training learning loop contains the following stages:

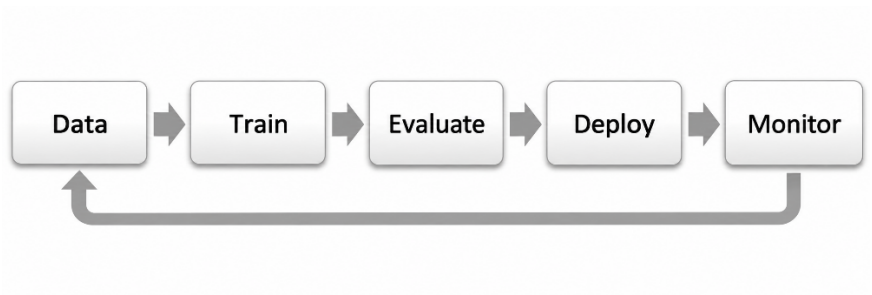


Figure 7.4: Post-training loop: Data → Training → Evaluation → Deployment → Monitoring

Data management is especially important in continuous systems. Policies evolve, workflows change, and product expectations shift over time. Training datasets must evolve with them. Mature systems usually version datasets explicitly, connect datasets to policy revisions, and track which datasets influenced each model version.

Evaluation becomes a gating mechanism rather than an advisory process. Every post-training update should be evaluated on independent datasets before deployment. Regression checks become particularly important because many failures appear in nearby behaviors rather than the target behavior itself.

Rollback is also part of the learning system. Stable systems assume that mistakes will eventually happen. Adapter versioning, staged rollouts, rollback workflows, and deployment isolation all become operational requirements.

Large organizations usually introduce additional complexity because multiple teams influence the same learning loop. Data teams, product teams, and model teams might all update different parts of the system independently. Stable organizations usually establish clear ownership for datasets, evaluation standards, deployment approvals, and rollback procedures.

The goal of an integrated learning loop is to support faster and more controlled iterations. The faster the learning loop becomes, the more important it becomes to understand why the model behavior changed. A well-designed learning loop is more than a training pipeline. It is an operational discipline that connects data, evaluation, deployment, and monitoring into a continuous feedback cycle. Teams that invest in these practices are better able to evaluate model changes, explain regressions, and safely roll back undesired changes when needed throughout model improvement cycles.

Summary

In this chapter, you learned when to use post training for improving LLM behavior and when inference-time controls, such as prompting, RAG, and tools remain the better choice. You explored the major post-training techniques including SFT, LoRA, preference alignment, and reinforcement learning. You also examined how data quality, evaluation, and operational learning loops determine their success.

There are three key takeaways from this chapter. First, post training is not the first solution. It should be considered only after inference-time controls have reached their practical limits. Second, different post-training techniques solve different problems. Choosing between SFT, LoRA, preference alignment, and reinforcement learning depends on the problems you are trying to solve, the data you have, and the operational burden you are willing to take. Third, post training introduces new responsibilities. Changing model weights requires high-quality training data, independent evaluation, well-designed reward signals, and continuous monitoring. Teams that apply these principles build

more reliable and maintainable LLM systems while avoiding unnecessary operational complexity.

The next chapter builds on these concepts by looking at LLM customization as a complete system rather than a collection of individual techniques. Instead of focusing only on post training, we will examine how prompts, retrieval, tools, synthetic and real data, and post-training methods work together to transform a general-purpose model into a reliable domain-specific assistant.

References

- Bommasani, R., et al. (2021). *On the opportunities and risks of foundation models*. Stanford Center for Research on Foundation Models. https://arxiv.org/abs/2108.07258
- Sculley, D., et al. (2015). *Hidden technical debt in machine learning systems*. NeurIPS. https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems
- Breck, E., et al. (2017). *The ML test score: A rubric for ML production readiness and technical debt reduction*. IEEE Big Data. https://doi.org/10.1109/BIGDATA.2017.8258038
- Brown, T. B., et al. (2020). *Language models are few-shot learners*. NeurIPS. https://arxiv.org/abs/2005.14165
- Rafailov, R., et al. (2023). *Direct preference optimization: Your language model is secretly a reward model*. https://arxiv.org/abs/2305.18290
- Hu, E. J., et al. (2021). *LoRA: Low-rank adaptation of large language models*. https://arxiv.org/abs/2106.09685
- Ouyang, L., et al. (2022). *Training language models to follow instructions with human feedback*. https://arxiv.org/abs/2203.02155
- Weng, L. (2023). *Reward hacking in reinforcement learning*. https://lilianweng.github.io/posts/2024-11-28-reward-hacking/
- Bai, Y., et al. (2022). *Training a helpful and harmless assistant with reinforcement learning from human feedback*. https://arxiv.org/abs/2204.05862
- Mitchell, M., et al. (2019). *Model cards for model reporting*. ACM. https://doi.org/10.1145/3287560.3287596

- Microsoft. (2026). *Customize a model with fine-tuning*. Microsoft Learn. https://learn.microsoft.com/azure/foundry/openai/how-to/fine-tuning

8

LLM Model Customization

The previous chapter focused on post-training and explored its techniques, including supervised fine-tuning, LoRA, preference optimization, and reinforcement learning. These techniques modify model weights and allow LLM development teams to adapt foundation models to their specific needs.

At this point, you might think of customization as a training or post-training problem. Training is one clear form of customization, but training is only one part of a broader customization process. Most applications should exhaust inference-time controls before modifying the model itself.

Most applications do not need a model that is good at everything. They need a model that behaves correctly within a particular set of domains and environments. For example, legal and medical assistants must abide by the authoritative references and safety requirements. An enterprise assistant must understand organizational terminology and structured knowledge. A content moderator must apply the policies consistently under a wide range of conditions.

Domain expertise takes different forms and lives at different layers. Some of it comes from knowledge, specific tools and workflows, and retrieval and recognition of authoritative policies and decisions. Most of this is data- and process-driven rather than model driven. This is why two assistants built on the same foundation model can behave very differently.

It is hard to define the need for specific customization techniques (for example, **retrieval-augmented generation** [RAG], finetuning or reinforcement learning) without first defining the behaviors you need to customize. Answering this determines what data is needed, how that data should be organized, and which customization techniques are likely to be effective.

This chapter approaches customization as a system-level process. Prompts influence behavior at inference time through instructions and few-shot examples. RAG provides access to stored domain knowledge. Tools connect the model to external systems that hold external data. Synthetic data generation expands coverage. Post-training reinforces the desired behaviors to become more consistent. Each mechanism contributes differently, but all of them consume or process some form of data.

Customization techniques are different in how they consume and use data. This view connects many of the topics we covered so far in the book. Chapters two and three focused on acquiring and processing data. Chapters four and five focused on retrieving data. *Chapter 6*, *Synthetic Data Generation* focused on generating synthetic data. *Chapter 7*, focused on training models using that data. This chapter brings those pieces together and shows how they contribute to the development of domain-specific LLM assistants. Prompting, retrieval, and post-training are not competing approaches. Each one solves a different problem and depends on different forms of data.

By the end of this chapter, you will understand the different dimensions of domain customization and the data assets that support it. You will also understand the roles of retrieval, synthetic data, evaluation, and post-training within a broader customization strategy.

In this chapter, you will cover the following topics:

- Understanding model customization as a system-level process
- Dimensions of domain specialization
- Building customization data assets for reusability
- Designing the customization workflow and its components
- Implementing the complete customization workflow for an enterprise assistant
- Customization patterns for different AI assistants

Before selecting customization techniques, you need to establish a stable definition of *customization* and what separates a domain-specific assistant from a general-purpose model.

Understanding model customization

The introduction established that customization is a system-level process supported by multiple techniques. The next step is to define model customization in the context of LLM systems. Model customization is the process of adapting a general-purpose model to behave more effectively within a specific domain, organization or application. This adaptation goes beyond expanding the model's knowledge. The adaptation includes the model response style and constraints, its reasoning and workflows, and the external tools it uses.

Throughout this chapter, customization refers to adapting the assistant's behavior. Customization mechanisms are the techniques used to produce that adaptation, such as prompting, RAG, tools, and post-training. Customization data is the knowledge, behavioral, preference, and feedback data consumed or processed by those mechanisms.

Customization happens at different layers of the LLM system using different techniques. These techniques work together as a customization stack. Each layer of the stack contributes different capabilities and consumes different forms of data. You will revisit this stack in detail later in the chapter.

Understanding customization starts with defining the behaviors you want the model to learn. When the target behaviors are clear, it becomes much easier to determine what data should be collected and which customization techniques are appropriate. The following sections build this foundation by examining the different dimensions of model customization and the role that data plays in each one.

Customization as a System-Level Process

The *model customization* term is broadly used to describe the process of adapting LLM behavior for a specific domain or application, even when the underlying model weights are not changed. In this chapter, model customization refers to customizing model behavior rather than necessarily modifying the model itself. Model customization can happen at inference time through system prompts and few-shot examples, or through post-training that modifies the model weights. The model behavior is also influenced by the surrounding components, including RAG pipelines, external tools, and application logic. These components continuously influence the information that goes to the model and the actions it can perform.

Consider an enterprise assistant that answers human resources (HR) questions. A prompt tells the model how to respond. Retrieval provides the latest HR policies. An authorization service determines which documents the employee may access. External tools retrieve information that documents do not store, such as leave balances. Even if the underlying model does not change, the assistant will behave differently when these components evolve.

Another useful distinction is between *domain knowledge* and *domain expertise*. Domain knowledge represents the information required to answer questions correctly. Domain expertise includes the workflows, decisions, policies, and operational constraints that determine the answer generation process. A legal assistant needs access to laws and regulations, but it also needs to recognize authoritative sources, follow legal reasoning, and communicate uncertainty appropriately. Likewise, an enterprise assistant must understand company policies and the business processes that govern the organization.

Most successful custom models are the result of a well-customized system. Understanding the role of each customization mechanism helps development teams decide which techniques to implement and what type of data is required to support them.

Table 8.1 summarizes the primary customization mechanisms that work together across the LLM system:

Layer	Purpose	Example
Prompting	Behavior control	System instructions
Retrieval	Knowledge access	Enterprise knowledge base
Tools	Action execution	Enterprise and public APIs
Post-training	Behavioral stabilization	**Supervised Fine-Tuning (SFT)**

Table 8.1: Customization mechanisms across the LLM system

Although customization mechanisms operate differently, they all depend on data. Prompts, retrieval, tools, and post-training consume and process different forms of data. The quality of the final assistant depends on the quality of that data and how well it represents the target domain. The next

section examines the different data assets that serve as the foundation of model customization.

Data as the foundation of customization

Data is the common foundation behind every customization technique, but different techniques consume it differently. RAG depends on data that applications can access during inference. Prompting depends on examples that demonstrate the desired behavior. Post-training depends on datasets that reinforce those behaviors during training. Improving the underlying data improves multiple customization mechanisms at the same time.

The relationship between data and customization techniques is easier to understand visually. *Figure 8.1* shows that reusable data assets support different customization mechanisms to transform a foundation model into a domain-specific assistant:

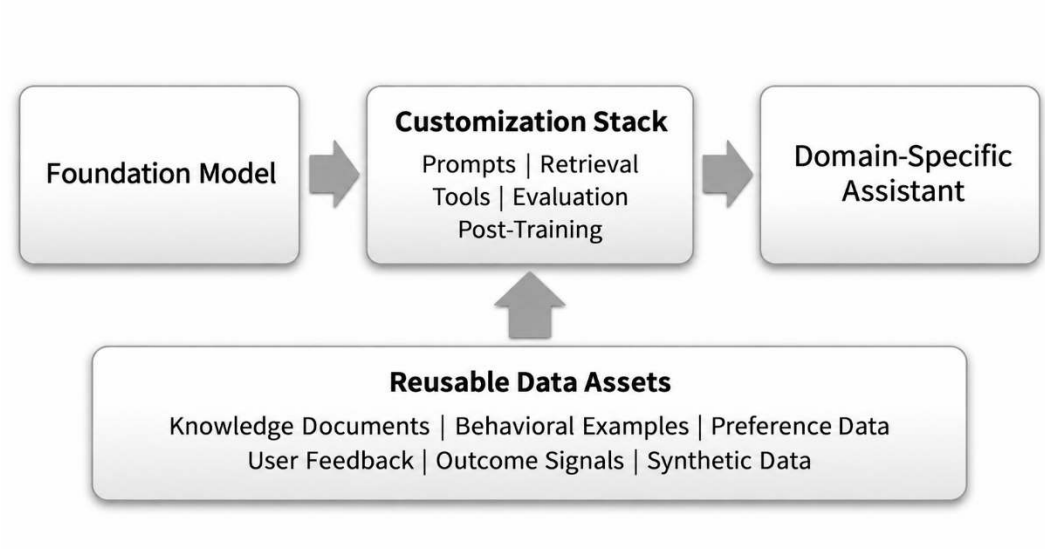


Figure 8.1: Data flow through the customization stack

Another useful way to think about customization is to follow the flow of data through the system. The same domain knowledge can appear as documents in a retrieval system, examples inside prompts, synthetic datasets for post-training, or evaluation benchmarks. The data representation in each of those is different, but the underlying knowledge stays the same.

This view has an important implication for LLM development teams. Improvements to the underlying data benefit multiple customization

techniques at the same time. Correcting an outdated policy improves retrieval quality. The same policy can later be used to generate synthetic examples, expand evaluation datasets, or produce post-training data. Instead of building isolated datasets for every stage of development, mature teams build reusable data that supports the entire customization process.

Reusing data across the customization stack is only part of the problem. It is equally important that the data represents the knowledge, workflows, and constraints of the target domain. A legal assistant, a medical assistant, and an enterprise assistant all depend on data, but not the same type of data. The next section examines the dimensions of domain specialization and how they influence the data requirements for customization.

Dimensions of Domain Specialization

This section covers three domain specialization dimensions and their impact on data requirements: knowledge and terminology, workflows and decision-making, and policies and compliance requirements. These dimensions cover different aspects of model knowledge and behavior. Knowledge defines what the assistant knows. Workflows guide how it acts. Policies control what it is permitted to do. Together, they provide a practical framework for the customization needs covered in this chapter and translate domain specialization into clear data requirements.

Every domain combines the three customization dimensions with different priorities. The most important customizations for a personal assistant are user preferences and interaction style. An enterprise assistant must understand organizational terminology, business workflows, and internal knowledge. Legal and medical assistants must focus on authoritative knowledge, expert decision-making, and safety requirements and de-prioritize less authoritative sources. A content moderator system needs to apply policies and guidelines and enforce them consistently.

Understanding the key dimensions for your application is required to both prioritize the customization techniques to be applied and the data to be collected or labeled. The following sections discuss each customization dimension individually before combining them into a complete customization strategy.

Knowledge and terminology

Knowledge is the first dimension of domain specialization because every assistant must understand the key concepts of its domain. Knowledge includes facts, relationships, common terms, and the references that users expect the assistant to know. Without this knowledge, the assistant performs poorly in understanding users and does not provide accurate responses as a result.

Terminology is a key part of domain knowledge. Every domain and organization has their own vocabulary and expressions that encode meanings specific to their fields. Even when a term looks like common knowledge, it might have different meanings across domains. AI agents must understand these specific terms because users communicate naturally using the language of their domain, and they expect the assistant to understand and respond in the same language.

For an enterprise assistant scenario, employees refer to internal systems, project names, product codenames, and organizational acronyms without additional explanation. A legal assistant must recognize legal terminology and references to known regulations for the specific country or region of operation. A medical assistant must understand clinical terms and procedures that are not frequent in general-purpose data. In general, AI assistants should interpret these terms correctly before reasoning about the problem at hand.

Table 8.2 compares the knowledge and terminology requirements across the four AI assistants discussed throughout this book:

Assistant	Knowledge Examples	Terminology Examples
Personal assistant	Calendar, contacts, preferences	Meeting names, user-defined labels
Enterprise assistant	HR policies, internal documentation	Project names, acronyms, system names
Domain-specific assistant	Medical guidelines, legal regulations	Clinical terms, legal terminology

Assistant	Knowledge Examples	Terminology Examples
Content moderator	Community standards, policy definitions	Internal policy labels, violation categories

Table 8.2: Examples of domain knowledge and terminology across different AI assistants

Knowledge and terminology are directly represented through data. Documents, manuals, policies, and knowledge bases contribute to this specialization dimension. Later in this chapter, we discuss the specific data assets and how they are organized. At this stage, it is sufficient to recognize that effective customization begins with a clear representation of the domain knowledge that the assistant is expected to know.

Knowledge alone is not enough to build a useful, specialized assistant. In addition to knowing the correct information, the assistant needs to follow the correct business process and make decisions as a domain expert. The next dimension of specialization focuses on workflows and the decision-making process that transform knowledge into consistent actions.

Workflows and decision making

Domain expertise extends beyond knowledge and terminology. Domain experts follow established workflows to complete their work, and make decisions based on their expertise and the information available at each step. Expert workflows and decision-making transform domain knowledge into domain expert behavior. AI assistants that know domain workflows and follow decision-making patterns resemble experienced professionals rather than just answering basic questions.

Workflows describe the sequence of steps required to complete a task. For example, an enterprise assistant might follow the employee onboarding process while helping with access provisioning for internal resources for a new employee. A good legal assistant reviews supporting documents before identifying the applicable regulations. A medical assistant supports clinicians by following specific workflows that organize relevant information, surface applicable clinical guidance, and support diagnosis and treatment decisions. Knowing and applying these workflows improves consistency and reduces the chance of skipping important steps.

Decision-making determines the right actions to take at each step. It encodes the choices that experts make when multiple actions are applicable. For example, a personal assistant decides whether to interrupt the user with a notification now or wait for a better time. An enterprise assistant decides which tool to call when the available information is not sufficient. A content moderator system decides whether a piece of content should be approved, rejected, or escalated to human review. Decision-making is defined by existing domain expertise, organizational practices, and the objectives of the application.

Behavioral data is as important as knowledge data for LLM customization. However, unlike domain knowledge where a document may describe a specific concept, workflows and decision-making patterns are usually much more distributed. They are distributed across historical records, organizational policies, expert demonstrations, meeting notes, and sometimes, a single workflow is distributed across multiple data sources that need to be joined together to understand the reconstruct the target pattern.

Table 8.3 shows that workflows and decision-making vary across the AI assistant applications discussed throughout this book:

Assistant	Typical Workflow	Example Decision
Personal assistant	Manage schedules, reminders, and personal tasks	Decide whether to interrupt the user or defer the notification
Enterprise assistant	Search internal knowledge and complete business processes	Decide whether to retrieve additional information or invoke an enterprise tool
Domain-specific assistant	Follow domain-specific procedures	Decide whether sufficient evidence exists to provide a recommendation
Content moderator	Review reported or generated content	Decide whether content should be approved, rejected, or escalated

Table 8.3: Examples of workflows and decisions across different AI assistants

Applying domain workflows improves consistency, but it does not define the permissions and limits that control the desired assistant behavior. Organizations also establish policies and compliance requirements that define what an assistant is allowed to do. The next section examines these policies and requirements, and how they influence model behavior.

Policies and compliance requirements

Knowledge teaches the assistant fundamental domain information. Workflows tell the assistant how the work is performed. Policies and compliance requirements determine what the assistant is allowed to do. They establish rules that govern model behavior regardless of the assistant's level of competency.

The nature of these policies varies across domains. A content moderation system applies policy guidelines or community standards before approving or rejecting content. An enterprise assistant validates data access and permissions before retrieving certain information or taking certain actions. A personal assistant operates within the boundaries of user privacy settings. Although these rules originate from different sources, all of them influence the assistant's behavior.

Policies are especially important when the assistant faces uncertainty, restricted information, or rule-violating requests. In such situations, the assistant might refuse to answer the question, request additional information, escalate to a human expert, or limit the response to authoritative sources. The model behaviors that are influenced by policies and requirements reflect healthy compliance, rather than limitations of the model.

Unlike fundamental domain knowledge, policies continue to evolve as the organization introduces new products and procedures. It is critical to refresh policies periodically and keep them current. An outdated policy can produce severely wrong actions even when the assistant understands the domain knowledge and workflows perfectly. Governance documents are important sources of customization data for applications that operate in regulated or changing environments.

Table 8.4 summarizes the dimensions of domain specialization introduced throughout this section and the types of data that support them:

Specialization Dimension	Typical Data Source	Example Target Behavior
Knowledge and terminology	Documents, manuals, domain content	Accurate facts and correct domain language
Workflows and decision making	Historical interactions, expert demonstrations	Procedural consistency and sound decisions
Policies and compliance requirements	Governance documents, policies, guidelines	Policy-compliant behavior

Table 8.4: Recap of dimensions of domain specialization and related data sources

Knowledge and terminology, workflows and decision-making, and policies and compliance requirements define the three dimensions used in this chapter to distinguish one domain from another. The next section shifts from these dimensions to the data assets required to represent them and support the customization techniques.

Building customization data

The previous section identified the dimensions that distinguish one domain from another. These dimensions define the behaviors that make an assistant specialized, but they do not tell you what data is required to build those behaviors. *Figure 8.1* introduced reusable data as the foundation of the customization stack. This section examines that foundation in detail by organizing customization data into the categories used throughout the rest of the chapter.

Customization data is grouped into four categories: knowledge data, behavioral data, preference data and feedback data. These categories reflect the primary role of the data during customization rather than the specialization dimension it supports. Synthetic data is treated differently because knowledge, behavioral, preference, and feedback data can all be expanded synthetically when additional coverage is required. These categories cover the major forms of data required to build and evaluate domain-specific assistants.

The relationship between specialization dimensions and customization data is not one-to-one. Every specialization dimension depends on multiple data categories, and each data category contributes to multiple specialization dimensions. For example, compliance documents define organizational policies, provide domain knowledge, and might describe business workflows. *Table 8.5* shows that each category of data contributes to multiple specialization dimensions introduced in the previous section. Synthetic data can expand any of these categories, so it supports all specialization dimensions wherever additional coverage is needed.

Data Category	Applicable Specialization Dimensions
Knowledge data	Knowledge, terminology, and policies
Behavioral data	Workflows and decision making
Preference data	Decision-making, policies and compliance
Feedback data	Decision-making, policies and compliance

Table 8.5: Relationship between customization data and domain specializations

Building high-quality knowledge data is the starting point for most customization efforts because it provides the key information that the assistant is expected to understand before responding or taking action. "It also serves as the foundation for other data assets throughout the customization process. Knowledge documents apply to retrieval, evaluation, and post-training. This broad reuse makes knowledge assets the first category to discuss before building the rest of the customization data.

Knowledge data

Knowledge data feeds the LLM assistant with the core information required to answer user questions within a specific domain. This information exists in company policies, product documentation, internal knowledge bases, and structured system records. During customization, you decide which documents and records are used for retrieval, evaluation, synthetic data generation, and post-training.

Chapter 2, Data Sources, focused on the collection of these data sources. This section focuses on selecting and using them as part of the customization process. Most organizations have knowledge data, but only a few of them have a reviewed data collection that is ready to use for LLM customization.

A common mistake is to retrieve and index all available knowledge data from all sources. This mistake is relatively justified for two reasons. First, more data can benefit LLM development, but additional volume does not guarantee better customization. Second, the review and selection process is very time-consuming compared to retrieval and indexing. However, without a selection process, the data index ends up containing outdated documents, draft versions, and duplicate copies. The problem does not sound too bad, but it is too bad when these documents give wrong and conflicting information. The LLM alone cannot distinguish between an approved company policy and an old document that somebody forgot to remove.

Consider an HR assistant where the travel policy is published on the HR portal. Another team copied the policy document into its own wiki and added their local guidance to their copy. In parallel, an old PDF is still shared with new employees because it is included in the onboarding material that was not updated in years. Before building the data index, you need to decide which documents must be included because they answer employee questions and add them first.

Other documents might still be useful. However as you retrieve and index additional documents, you should compare them with what is already in the index and make sure that additional information does not contradict the initial dataset that was carefully selected. For example, the HR system stores an employee's vacation balance in a structured database. The travel policy document explains how that balance is calculated. When the user asks, "*How many vacation days do I have left?*," the database is used. When the user asks, "*How is my vacation balance calculated?*," the policy document is used. The assistant needs both data sources, but each source answers a different question without conflict.

In a typical enterprise AI project, teams spend most of the data-related work on comparing document versions, identifying duplicate copies, verifying that documents are still current, and removing wrong and contradictory

information. These activities determine the quality of the knowledge available to the model. Data quality matters for LLM customization regardless of data volume.

Another important practice is to identify the source of truth for every important type of information. Enterprise applications contain several copies of the same information. The same financial information might exist as an Excel sheet used by one team and as a table in a database used by a different system. Decide which system owns the data before connecting those sources to the assistant. That decision simplifies evaluation because the expected answer comes from a known source. It also simplifies debugging when the assistant produces an incorrect answer.

Knowledge data supports several customization techniques. RAG systems use it for grounding responses", referring to knowledge data in current enterprise information. Synthetic data generation uses it as a seed to create more realistic questions, answers, and evaluation scenarios. It also provides high-quality examples for supervised fine-tuning. Reusing the same knowledge across multiple techniques reduces duplicate data preparation and keeps the assistant consistent.

On the implementation side, several libraries simplify document preparation. For example, the Unstructured library and Apache Tika extract text and metadata from enterprise documents. LlamaIndex, Haystack, and LangChain provide loaders for document repositories and structured data sources.

These libraries prepare the data for indexing, but they do not review document quality, identify the current policy, or remove duplicate documents. This code example prepares a document collection before indexing. The example applies very simple techniques that help identify duplicate policies, detect outdated versions based on document metadata, and keep only the latest versions.

```
# Load documents and the approved source for each policy.
documents = load_documents("knowledge_documents.jsonl")
sources_of_truth = load_source_registry("source_of_truth_registry.csv")

# Group related versions and compare their normalized content.
for policy_id, versions in group_by(documents, key="policy_id").items():
    for document in versions:
        document["content_hash"] = hash_text(
```

```
            normalize_text(document["content"])
        )

    duplicates = find_duplicate_hashes(versions)
    near_duplicates = find_similar_documents(versions, threshold=0.45)

    # Keep the latest approved version from the designated source of truth.
    authoritative = [
        document for document in versions
        if document["status"] == "approved"
        and document["source_system"] == sources_of_truth[policy_id]
    ]
    selected = max(authoritative, key=lambda item: item["effective_date"])
    cleaned_collection.append(selected)

    # Conflicting, copied, draft, and outdated versions require review.
    review_queue.extend(
        flag_for_review(versions, selected, duplicates, near_duplicates)
    )

write_jsonl("cleaned_knowledge_collection.jsonl", cleaned_collection)
write_csv("knowledge_review_queue.csv", review_queue)

# Chapter_8_Notebook.ipynb: Preparing Knowledge Data for Customization
```

Knowledge data provides the information required to answer questions about a business. The next section introduces behavioral data, which tells how AI agents should perform business-related tasks.

Behavioral data

The previous section covered the knowledge data used during customization. However, knowledge alone does not make expert behavior. Good documentation explains a business process, but it does not show how an experienced employee follows that process. Behavioral data fills that gap by feeding AI agents with examples of people performing these tasks. It captures the human expertise associated with performing tasks, such as asking clarifying questions, making decisions, handling exceptions and using tools to complete workflows.

Common sources of behavioral data include conversational interactions through messages, emails, calls and support tickets, and expert demonstrations. However, in most cases, this data cannot be used as is for customization for many reasons. Many conversations contain incomplete context, unnecessary back-and-forth, and responses that do not represent the

desired behavior. You might also think that every resolved support ticket represents a good example, but that is not necessarily true. A ticket may have reached the correct outcome due to undocumented reasons or have gone through unnecessary steps and escalations.

A good behavioral example starts with a clear objective, ends with a clear outcome, and consists of a well-captured set of steps. The interaction starts with the user asking a question, requesting an action, or reporting a problem. The human agent asks clarification and diagnostic questions, uses business tools to collect information or complete actions, and makes decisions based on documentation and previous cases. The interaction ends after the question is answered, the action is completed, or the correct decision is made.

Knowledge data teaches the assistant what to know. Behavioral data teaches the assistant how to apply knowledge. Think about your favorite teacher. They may teach the same material as every other teacher, but the way they explain concepts, ask questions, and guide students makes the difference. Behavioral data plays a similar role. It is also a critical role because users perceive AI behavior, not its underlying knowledge.

Behavioral data faces a different set of challenges than knowledge data. The first challenge is extracting useful conversations. Conversations contain greetings, unrelated discussions, and incomplete requests. Additionally, many conversations do not reach a successful outcome in the first place.

Applying simple filters removes many of these interactions, but filtering alone is not enough. Filtering may also remove conversations that contain valuable behavioral examples. For example, a conversation may start with several minutes of greetings before moving into a well-executed troubleshooting session. Another conversation may contain multiple independent tasks, where only one of them is useful for customization. Discarding the entire conversation means losing examples that demonstrate the behavior we want the assistant to learn.

Instead of filtering entire conversations, it is better to extract the useful interactions from them. The process starts by identifying, then either labeling or removing parts of the conversation that do not contribute to the task, such as greetings, unrelated discussions, or repeated confirmations. Then, you can split the remaining conversation into smaller task-oriented interactions,

where each interaction represents a single objective and a successful outcome. By following this process, some conversations might produce several behavioral examples instead of discarding them. Preparing behavioral data is an extraction task rather than a filtering task.

This process can expand to full-fledged conversation segmentation. The basic idea is that splitting a conversation should be based on task boundaries rather than conversation boundaries. A single conversation might contain multiple tasks, while a single task may span many conversations. The objective is to extract complete task-oriented interactions, not a time-based set of sequential messages.

Another challenge is that many workflows are only partially captured. The conversation may show the user question and the final answer, but not some of the work performed by the human agent in between. The human agent might run internal or web search, consult another team, and use multiple business tools throughout the conversation. These actions happen outside the messaging platform and get recorded in different systems, if recorded at all. Without these actions captured, the behavioral example could be deceiving rather than useful.

Identifying and handling this issue is very important when preparing behavioral data. Many missing steps can be reconstructed by joining conversations with application logs, database records and tool execution history. In other cases, the missing behavior can be recreated by connecting with expert demonstrations or through synthetic examples. If none of that works, it is better to filter out such conversations.

That said, you do not need to solve all problems at once. As a practical strategy, start with a small collection of high-quality examples rather than trying to process every available example. Then, evaluate whether the assistant follows the expected workflow successfully according to the initial examples. If these examples work well but lack coverage, expand the dataset to cover additional scenarios and edge cases that were not represented. This approach reduces the introduced noise and simplifies evaluation, debugging and rollback, often resulting in a more maintainable model than a model that is trained on every available conversation.

The following example prepares historical conversations for few-shot prompting or supervised fine-tuning. Instead of using complete chat histories, it extracts conversations with a clear objective and a successful outcome.

```
# Load historical conversations.
conversations = load_jsonl("behavioral_conversations.jsonl")
quality_examples = []

for conversation in conversations:
    # Remove non-task events and split conversations at task boundaries.
    task_interactions = segment_by_task(
        remove_non_task_events(conversation["events"])
    )
    for task in task_interactions:
        outcome = conversation["task_outcomes"][task.id]
        # Retain only complete interactions with successful outcomes.
        if outcome["status"] != "resolved":
            continue
        if not outcome["captured_steps_complete"]:
            continue
        quality_examples.append({
            "objective": task.first_user_message,
            "events": task.messages_and_tool_actions,
            "outcome": task.final_assistant_message,
        })

# Export complete quality examples and compact training pairs.
write_jsonl(
    "behavioral_quality_examples.jsonl",
    quality_examples,
)
write_jsonl(
    "behavioral_instruction_response_pairs.jsonl",
    to_instruction_response_pairs(quality_examples),
)

# Chapter_8_Notebook.ipynb: Extracting High-Quality Behavioral Examples
```

The components of the customization stack that consume behavioral data differ from those that consume knowledge data. Few-shot prompting uses behavioral examples to reinforce the expected task execution flow during inference. Supervised fine-tuning learns the same behaviors from a much larger collection of examples. Evaluation also benefits from behavioral data in two ways. Known workflows can be replayed as regression tests to verify that previously supported behaviors remain intact, while held-out variations of

those workflows test whether the assistant generalizes to requests that have not been seen before.

This section focused on teaching the assistant how tasks are performed. The next section introduces preference and feedback data, which teaches the assistant to pivot toward the preferred responses when more than one valid answer exists.

Preference and feedback data

Preference data and feedback data are separate data categories, but they are discussed together because they provide related signals in relation to the desired assistant behavior. Preference and feedback data help the LLM decide which response to produce when multiple responses are valid.

Although preference data and feedback data are closely related, they are not the same. Preference data compares multiple responses and identifies the preferred one. Feedback data measures the goodness of responses according to user reactions or business outcomes. Both improve assistant behavior, but they provide different learning signals and require different preparation processes.

Experts source preference data from expert reviewers who compare two or more responses for the same prompt. The reviewers select the preferred response according to predefined and consistent criteria and record the reason behind that decision. Users and systems mostly provide feedback data from user ratings and escalations, business outcomes, and other operational signals collected while the assistant is running in trial mode or production.

Collecting high-quality preference data requires a significant human investment. Reviewers need clear evaluation criteria, training, and enough domain knowledge to make consistent decisions. Many organizations use a second reviewer and apply a tie-breaker process when reviewers disagree. Recording the rationale behind each decision improves consistency but increases the cost of building the dataset. Preference data is one of the most expensive data assets to build.

Consider a personal assistant that helps a user plan their next day. The user asks, *"Help me organize tomorrow's schedule."* The assistant generates two possible responses for the same request. The first response creates a list of

meetings in chronological order. The second response lists the meetings, identifies scheduling conflicts, and recommends moving one meeting to avoid travel time. Both responses answer the user request correctly. However, most people would prefer the second response because it is more useful. Preference data makes response comparison explicit by identifying which response users would prefer and why.

Now, consider the same assistant after deployment to production, generating a single response. The user sees only one response and has the option to select a thumbs-up or thumbs-down button after reading it. This is not preference data because there is no comparison between multiple responses. This signal is feedback data. The feedback tells you whether the user liked the response or not, but it does not tell whether another response would have been better.

One of the biggest misconceptions is that user feedback should be used for learning directly. There are many practical issues with that. One issue is that users are not required to be objective and they might prefer different responses for no clear reasons. Some users prefer short answers, while others expect detailed explanations, and even a single user can change their feedback over time based on external conditions.

Another issue is that some users do not like follow-up questions, even when those questions help the assistant clarify the user's intent and reach the correct conclusion while avoiding mistakes and repetitions. One more frequent issue is that users might prefer responses that violate regulatory policies because they produce the desired outcome. User feedback must be validated and transformed before using it as training data.

Collecting feedback at scale introduces another challenge. Voluntary feedback, such as user surveys, is usually sparse and biased toward strong positive and strong negative reactions. Business outcomes have other limitations because they are influenced by many factors beyond the assistant's response. Sparse feedback is not low-quality feedback, but it requires careful handling.

Additional validation boosts the value of feedback data. One of the best validation steps is to check user feedback against company policies and other authoritative guidelines before using it for learning. Augment that with asking an independent LLM judge that uses a separate powerful model and

human-reviewed instructions to evaluate the assistant's response against expected behavior and compare the LLM judgment with the user feedback.

Many applications, especially enterprise applications, can rely on additional stronger signals than user ratings alone. Users might ignore surveys, but their follow-up actions reveal whether the assistant provided a useful response to them. Actions like executing the suggested next step, accepting the generated content, or avoiding escalation to a human agent are all stronger indicators than a simple thumbs-up or thumbs-down.

Reliable feedback requires combining multiple signals. User ratings, business outcomes, and system events capture complementary aspects of assistant response quality. Fusing these signals together produces a much more reliable dataset than using any one of them alone.

Once collected and prepared, preference data and feedback data function as valuable inputs to different customization techniques. This distinction determines the utility of each dataset during customization. Preference data supports supervised preference learning. Feedback data supports reinforcement learning after the raw signals are filtered, attributed, and validated, then translated into carefully designed rewards or reward models. The choice between them depends on the behavior you want to learn and the data available to you. Treating them as complementary data assets creates a continuous learning process. *Table 8.6* summarizes the main differences between preference data and feedback data.

Comparison Aspect	**Preference Data**	**Feedback Data**
Source	Expert reviewers	Users and production systems
Input	Two or more responses	One response
Primary signal	Preferred response	User or business reaction
Human supervision	High	Low

Comparison Aspect	Preference Data	Feedback Data
Typical volume	Limited	Large
Main challenge	Expensive human supervision	Noise and sparse feedback
Typical use	Preference optimization	Reward learning and feedback loops

Table 8.6: Comparing preference data and feedback data

Several tools simplify collecting and reviewing preference and feedback data. For example, **`Argilla`** and **`Label Studio`** support pairwise comparisons and human review workflows. **`LangSmith`**, **`OpenAI Evals`**, and **`Azure AI Foundry`** help collect reviewer feedback, compare responses, and organize evaluation datasets. These tools simplify the review process, but do not eliminate the need for well-defined evaluation criteria and review guidelines.

The next section revisits synthetic data in the context of model customization. Unlike the previous data categories, synthetic data is generated rather than collected. It expands the available training and evaluation data when real-world examples are limited or difficult to obtain.

Synthetic data

Synthetic data is not a separate data category. Synthetic data can augment any of the data categories you discussed earlier. Knowledge data, behavioral data, preference data, and feedback data can all be either collected from real-world systems or generated synthetically. In the context of model customization, synthetic data expands or refines existing datasets instead of introducing new domain knowledge.

Chapter 6, discussed the techniques used to generate synthetic data and the engineering considerations behind them. This section focuses on synthetic data within the model customization process. *Figure 8.2* shows the role of synthetic data across the customization process. Unlike the other data categories, synthetic data expands the existing customization assets by

generating additional knowledge, behavioral, and preference examples, and additional evaluation scenarios wherever more coverage is needed.

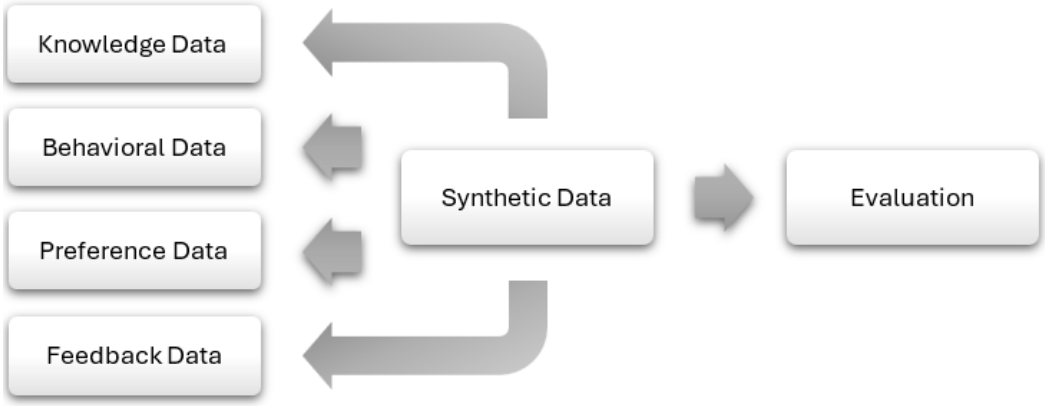


Figure 8.2: Synthetic data expands each customization data category

One of the biggest misconceptions around synthetic data is that more data automatically produces a better LLM, but that is not true in many cases. If the original data contains wrong examples or inconsistent behavior, generating more synthetic data simply amplifies these problems. Synthetic data should expand a good dataset, not recover from a bad one.

This distinction separates data quality problems from data coverage problems. Synthetic generation should not be used to compensate for low-quality data, but it is valuable for expanding a high-quality dataset that lacks sufficient coverage. Examples include missing knowledge scenarios, infrequent workflows, edge cases, or insufficient examples in the formats required by customization techniques.

On the good side, synthetic data expands every customization data category. It generates additional question-answer pairs to improve knowledge coverage, creates behavioral examples for less-frequent workflows, and reduces the cost of collecting preference examples. It is also useful for expanding evaluation datasets with edge cases that might not appear in existing data but are still important to test. Generate synthetic examples selectively to close the identified gaps. Selective generation produces focused datasets that are easier to validate, maintain, and refresh.

Synthetic data should follow the same quality standards as real-world data. Every generated example should be reviewed and validated before integrating it into the customization datasets. Synthetic data is easier to generate than

real-world data, but it should not receive less scrutiny. It fills well-defined gaps and does not replace real-world examples. The strongest customization datasets combine carefully generated synthetic data with representative real-world data.

This recommendation applies to every customization data category discussed in this chapter. Knowledge data, behavioral data, preference data, and feedback data all influence LLM assistant behavior regardless of whether they are collected or synthetically generated. Data quality determines the quality of the final product. The next section discusses the data quality requirements that apply across all customization datasets.

Data quality requirements

The previous sections discussed the main customization data categories. One important lesson applies to all of them. All customization techniques depend on high-quality data. However, the data quality requirements are different based on each customization technique. The same dataset can be used for multiple customization techniques, but the quality aspects that matter most are different across techniques.

For example, consider a well-maintained collection of company policy documents that make a good knowledge dataset. It may be fresh enough to support RAG workflows, but it might not provide enough coverage to generate evaluation scenarios. As another example, a recorded conversation may be a good behavioral example for few-shot prompting, but usually it would not be the golden example for a supervised fine-tuning dataset. Therefore, data quality should be evaluated according to how the data will be used during customization, not just based on the data source or category.

Table 8.7 summarizes the most common quality requirements for the customization data categories discussed throughout this chapter. Notice that the same data category appears more than once because the quality requirements change based on how the data is consumed during customization.

Data category	Used as	Common quality requirements	Why it matters

Data category	Used as	Common quality requirements	Why it matters
Knowledge data	Retrieval corpus	Authority and freshness	Prevents outdated or conflicting answers during retrieval
	Evaluation dataset	Coverage and realism	Verifies that the assistant answers both common and edge-case questions
Behavioral data	Few-shot examples	Representativeness	Demonstrates the expected workflow during inference
	SFT dataset	Consistency	Assures that the model learns the correct behaviors
Preference data	Preference dataset	Reviewer agreement	Produces stable and reliable preference signals
Feedback data	Reward signals	Stability and validation	Reduces noisy feedback before reinforcement learning

Table 8.7: Data quality requirements across customization assets

The quality requirements in this table represent necessary quality gates required before using the data for customization. Additional requirements exist for specific applications, domains and enterprises. These quality gates are practical starting points for most LLM customization projects.

When any of these datasets contain synthetic examples, synthetic data validation is an additional quality requirement. Synthetic data validation needs to satisfy the same requirements as real-world examples without generation errors or inconsistencies.

Reusing customization data also requires appropriate governance. Enterprise documents, historical interactions, preference data, and feedback might contain confidential or sensitive information, so access controls and data-

handling requirements continue to apply when these assets are reused for prompting, retrieval, evaluation, or post-training. *Chapter 10, Data Quality and Human Oversight,* discusses the broader data governance life cycle in more detail, including quality gates, provenance, versioning, monitoring, and retention.

This section focused on building the customization data required to build a specialized LLM assistant. The next section shifts the discussion from the data to the customization process. You will examine the connection between data categories and different stages of the customization workflow and how to answer the design questions that shape your customization workflow.

Designing the customization workflow

The previous section focused on the reusable data that forms the foundation of the customization stack as explained in *Figure 8.1*. This section moves to the customization mechanisms that consume that data and focuses on the utility of each mechanism. Building a successful specialized assistant is not about choosing one customization technique. It is about selecting and executing the right techniques for the right problem and time and knowing when to move to the next one and which one to move to.

Customization is an iterative process, not a collection of techniques. Start with the simplest set of customization techniques that satisfy the known application requirements, and introduce additional complexity only when evaluation shows the need for it. Evaluation drives every customization decision by identifying whether the current solution is sufficient and the remaining failure modes.

Understanding the customization life cycle

The customization life cycle puts this principle into practice. *Figure 8.3* shows the sequence from inference-time controls through retrieval, data scaling, and post-training, with evaluation acting as the gate after every stage. A successful evaluation can lead to deployment at any point, while remaining failures determine the next customization step:

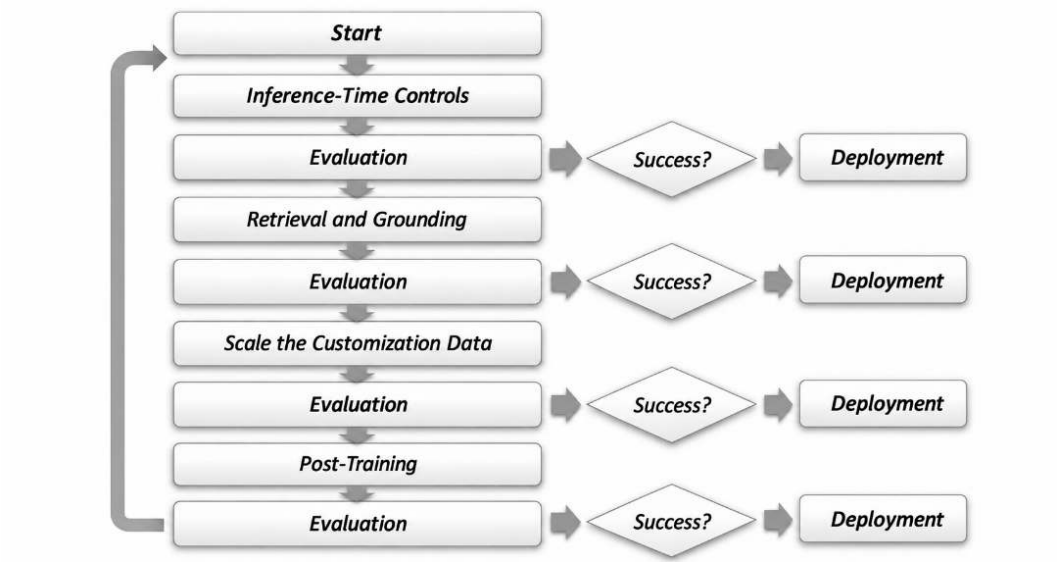


Figure 8.3: The LLM customization life cycle

Other customization mechanisms, such as tools, can be introduced at any stage when the application requires access to external systems or capabilities beyond model generation. This workflow highlights three important principles:

1. Customization is iterative, not a set of independent techniques. Every stage builds on the previous one and does not replace it.
2. Training is not the default solution. Most applications can achieve their goals through prompting, RAG, and few-shot examples without modifying the underlying model.
3. Evaluation is the gating function for every stage. Teams should evaluate at each step and move to the next one only when the current solution cannot address the remaining failure modes.

The next section discusses the connection between the customization data introduced in Section 3 and the customization techniques shown in this workflow and shows that different data categories contribute to different customization techniques.

Connecting data to the customization workflow

Figure 8.1 introduced customization data and customization mechanisms as parts of the same stack. The customization lifecycle adds the sequencing

between these mechanisms. The same customization data supports multiple stages of that life cycle.

Consider the enterprise assistant discussed throughout this chapter. The same software documentation and troubleshooting guides used to answer developer questions can be indexed for retrieval, generate evaluation scenarios, or seed synthetic question-answer pairs that expand test coverage. Refined versions of the conversational interactions between developers can provide few-shot examples for inference, act as supervised finetuning examples, and contribute to evaluation datasets. Reusing the same data across multiple customization techniques reduces data preparation effort and helps maintain consistent LLM behavior.

Preference data and production feedback are also reused throughout the customization workflow. Preference comparisons are used to train preference optimization models and validate improvements over time. Production feedback helps identify new improvement and evaluation scenarios, measure performance after deployment, and provide reward signals for reinforcement learning. *Table 8.8* shows how customization data is reused throughout the customization workflow. Presence of a check mark indicates that the data category contributes to the corresponding customization stage.

Data category	Prompting	RAG	SFT	RL	Evaluation
Knowledge data		✓	✓		✓
Behavioral data	✓		✓		✓
Preference data	✓			✓	✓
Feedback data				✓	✓

Table 8.8: Reusing customization data across the customization workflow

Synthetic examples can contribute to any of these customization stages according to the data category they expand. For example, synthetic behavioral

data can support prompting and SFT, while synthetic knowledge data can support RAG and evaluation.

Do not interpret this table as a fixed set of rules. You can reuse these data categories differently based on your requirements. The important principle is that the same data is reused throughout the customization lifecycle. Building reusable data assets reduces duplicate work, improves behavioral consistency through the customization process, and simplifies improvement over time.

The next step is to examine each stage of the customization workflow. You begin with inference-time controls, which customize assistant behavior by providing instructions, context, and examples during inference without modifying the underlying model.

Starting with inference-time controls

The first stage of customization uses inference-time controls. These techniques customize an assistant by changing the input data that goes into the model during inference rather than modifying the model itself. They are inexpensive to implement and easy to update. They provide enough customization for many applications without investing in additional training infrastructure.

Inference-time controls include four common mechanisms. Although they are often used together, each serves a different purpose.

- System prompt defines the assistant's role, responsibilities, and global constraints. It establishes behavior that should remain consistent across all interactions.
- Task instructions specify what the assistant should accomplish for anticipated requests. They guide the model to perform known tasks and follow application requirements.
- Few-shot examples demonstrate the expected behavior through representative examples. They are particularly useful in areas where teaching by example is more effective than general instructions.
- Prompt templates combine system prompt, task instructions, retrieved context, few-shot examples, and user input into a reusable structure. They ensure that every request is presented to the model in a consistent format.

The objective is to guide the model toward the desired behavior using information provided at inference time. The four mechanisms usually work together, but they do not solve the same problems. You need to decide whether the desired behavior should be expressed through instructions or through examples. A simple engineering heuristic is to ask two questions:

- Can you clearly describe the behavior?
- Can you demonstrate the behavior by example?

If you can describe the desired behavior clearly, system prompts and task instructions could be sufficient. For example, you can instruct the LLM to answer user questions using company terminology, cite internal documentation, and avoid guessing when the required information is not available to the model.

If the behavior is easier to demonstrate by example rather than description, the few-shot technique would be a better choice. For tasks such as support ticket summarization and incident report generation, add a few representative examples that show the expected structure, level of detail, and writing style.

Prompt templates complement these approaches by organizing the prompt components into reusable templates with content placeholders. Rather than constructing a new prompt for every request, the application populates the same template with the appropriate instructions, retrieved context, few-shot examples, and user input for each request. Here is an example prompt template.

```
System Prompt:
  You are an enterprise assistant that answers employee questions using company documentation.

Task Instructions:
  Provide a concise answer and cite the relevant documentation when available.

Retrieved Context:
  {retrieved_documents}

Few-Shot Examples:
  {few_shot_examples}

User Request:
  {user_query}
```

Only the retrieved context, few-shot examples, and user input change from one request to the next. The template itself remains unchanged, ensuring that every request follows a consistent structure while adapting to the current task and available context.

Always evaluate inference-time controls before implementing more advanced customization techniques. If evaluation shows that the required behavior is sufficiently covered, further customization might not be required. When the remaining failures are due to lack of knowledge or inconsistent behavior across scenarios, the customization journey should progress to the next stages.

The remaining failure modes usually indicate the next stage of the customization workflow. *Table 8.9* summarizes common failure modes and the corresponding customization approach.

Failure mode	Next step
The assistant lacks domain knowledge or uses outdated information	Add retrieval and grounding through RAG
The assistant is not consistent in performing a workflow despite clear instructions and examples	Improve the behavioral data and evaluate whether post-training is necessary
The assistant needs information of capabilities that require access to external systems	Integrate tools or function calling with the required instructions and examples
The assistant continues to fail frequently after trying all other improvements	Evaluate post-training techniques

Table 8.9: Selecting the customization technique based on failure mode

The important takeaway is to use the customization workflow as a decision framework and use evaluation as the gating mechanism between stages. Start with the simplest combination of customization techniques that satisfies the known application requirements, evaluate the results, and introduce additional techniques when the remaining failures justify them. This approach avoids unnecessary complexity while ensuring that each

customization step creates a solution for a specific limitation that will be validated through evaluation.

The next section discusses retrieval and grounding, which is the next step when the missing knowledge can be provided at inference time.

Expanding through retrieval and grounding

The previous section examined the inference-time controls that change the input to the model during inference to customize the model behavior, rather than modifying the model itself. Retrieval-augmented generation (RAG) follows the same principle but addresses a different category of failures. RAG changes the information available to the model during inference by retrieving the relevant knowledge from external sources and incorporating it into the prompt.

A practical engineering heuristic is to check whether the required knowledge can be provided at inference time. If the answer is yes, RAG is usually the best customization technique to use at this stage. RAG is most effective when the model generally understands the domain and knows how to perform the target tasks, but it does not have access to the specific information required to answer some of the user requests. This is usually the case when the required information is not part of the model training data or changes frequently and independently of the model training cycles.

For example, a personal assistant might understand how to summarize news articles, but it cannot summarize today's news without retrieving it first. A content moderation assistant may understand moderation policies in general, but it still needs access to the latest policy revisions before classifying new content. An enterprise assistant may know how to answer employee questions, but it still needs access to the latest product documentation and customer information to generate correct responses.

Chapters four and five explained building and optimizing RAG systems from preparing the retrieval corpus and indexing documents to improving retrieval quality and handling production-scale challenges. Within the context of the customization workflow, the key concerns are deciding when and how to use RAG. If the assistant generally understands the domain and tasks but fails to

provide the correct answer due to lack of the required information, RAG is usually a better solution than adding more instructions or post-training.

RAG is frequently used as a core component of model customization workflow because most of the LLM applications depend on external information that changes frequently and independently of the model development cycles. The development effort associated with RAG is spent on refining the indexed information, updating the data sources, and integrating the retrieved information into prompt templates. The quality of the retrieval process and retrieved data has a large impact on overall application quality.

Once implemented, RAG is operationally efficient compared to post-training and other customization techniques. When new software is released, company documentation is updated, or new regulations are published, you need only to update the retrieval index rather than updating the model. Even better if you use a separate search tool, you don't need to update the index since it is encapsulated by the search tool. This allows the application users to continuously benefit from the latest information without updating the underlying model.

RAG extends the customization workflow by supplying restricted and fresh knowledge. When evaluation shows that failures are caused by missing or outdated information, incorporating RAG is probably the next best step. When the required knowledge is already available, but the assistant fails frequently, the customization workflow should continue by improving the customization data before considering post-training.

Scaling the customization data

The previous section focused on providing access to the required information available during inference. This section assumes that the assistant already has access to that information. Many remaining failures would result from a mismatch between the available customization data and production-time user requests. For example, some requests are phrased differently than the incorporated examples and some edge cases are not represented at all. Scaling the customization data addresses these gaps.

Scaling the customization data is not just about adding more data. Scaling the customization data means improving the knowledge data, behavioral data

and preference data that are used for few-shot examples and RAG. Different failure cases require different improvements. Some failures benefit from additional examples, while others benefit from better examples or removal of conflicting examples. Two simple questions help you identify these cases:

- Are edge cases missing?
- Are the available examples consistent?

To answer these questions, evaluation should focus on root cause analysis of the failed examples rather than the overall evaluation score. The objective is to understand the root causes behind the failures before moving to data improvement.

Consider an enterprise assistant that answers employee questions about company policies. Assume that the required policy documents are already indexed and retrieved correctly, and that basic questions are answered correctly. However, employees ask the same question using different terminology or describe their situation in different ways. The model might fail because these variations are not represented in the existing behavioral examples. Adding more representative examples or generating synthetic variations is the best solution for this problem.

Another common root cause is the presence of inconsistencies within the customization data itself. Some examples may demonstrate different workflows for the same task or provide conflicting responses to similar requests. In such cases, the model cannot follow a reliable pattern because the examples themselves are not consistent. Improving the existing data by removing conflicting examples or replacing them with higher-quality examples produces larger improvements than just adding more data.

Scaling the customization data includes expansion of the knowledge corpus, collection of additional behavioral examples, generation of synthetic data, refining preference comparisons, and removal of low-quality examples. The appropriate solution depends on the specific failure cases identified during evaluation. The objective of data scaling is not to increase the volume of the customization data. The objective is to improve the representativeness of the customization data toward the expected behavior.

Scaling the customization data should always be something teams should consider before post-training. If evaluation still identifies the same failures

despite sufficient model information, customization data coverage, and consistent examples, continue the workflow with post-training.

Escalating to post-training

The customization steps discussed so far improve the assistant without modifying the model itself. This section introduces the final stage of the customization workflow, where the model is customized through post-training. Post-training becomes justified when evaluation shows that inference-time techniques are not sufficient to resolve the remaining failures. Three practical questions help determine whether post-training should be considered:

- Does the model ignore or fail to follow the given instructions?
- Are prompts becoming too large or difficult to maintain?
- Does the model fail to use the available information?

These situations indicate the need for post-training rather than overcomplicating the prompt with instructions and few-shot examples. By the time post training is needed, most of the required customization data should already exist. Yes, these are the knowledge data, behavioral data, preference data, feedback data and evaluation data discussed earlier in this chapter. Post-training uses these data assets rather than requiring new types of data.

Chapter 7 introduced the main post-training techniques and explained their impact on the model behavior through additional training. In the customization context, supervised fine-tuning teaches the model new knowledge and behaviors from representative examples. LoRA provides an operationally efficient implementation of supervised fine-tuning. Preference optimization teaches the model to generate more of the preferred responses. Reinforcement learning optimizes model behavior using a variety of reward signals collected during evaluation or production.

The decision to use any of these techniques depends on the failure modes identified during evaluation. A model that ignores instructions might benefit from supervised fine-tuning. A model that produces several acceptable responses but does not consistently choose the preferred one may benefit from preference optimization. Reinforcement learning might be useful when

golden responses are not practical and the target behavior can be represented reliably through reward signals. *Table 8.10* summarizes common evaluation findings and the corresponding post-training recommendations:

Evaluation Finding	Next Step
Instructions solve most of the problems	Improve inference-time controls
Retrieval provides missing information successfully	Expand RAG coverage
Better examples improve the behavior	Scale the customization data
The model still ignores instructions	Consider supervised fine-tuning
Model answers are correct but not preferred by users	Consider preference optimization
Golden answers are not applicable for failure cases	Consider reinforcement learning

Table 8.10: Selecting post-training techniques based on failure modes

Remember that post-training increases the operational complexity of your model and dependent applications. Unlike prompting and retrieval, training model weights requires additional validation, deployment, and maintenance steps whenever a training cycle is required. This additional investment is usually unnecessary if inference-time customization satisfies the application requirements. Post-training is introduced when evaluation justifies it, and it is validated through the same evaluation process before moving forward. This makes evaluation the common decision-making mechanism throughout the entire customization workflow rather than a final validation step.

Using evaluation as a gating function

The previous sections introduced the main customization mechanisms and explained when each one should be used. Evaluation connects these mechanisms into a practical workflow. Every customization stage is followed by evaluation before introducing additional complexity or deployment. Evaluation acts as the gate between customization stages.

Evaluation for LLM customization should be driven by the application requirements rather than generic benchmarks, because different applications measure success differently. A personal assistant might focus on task completion and user satisfaction. An enterprise assistant might emphasize answer correctness, citation accuracy, and grounded responses. A content moderation system might prioritize precision, recall, or false negative rates depending on the application risks. The evaluation metrics and acceptance thresholds should reflect the requirements of the target application.

The same evaluation data and metrics should be reused throughout the customization workflow. Changing more than one variable at a time makes it difficult to analyze whether the observed improvement came from the customization itself or from changes in the evaluation process. Keeping the evaluation consistent helps to compare different customization techniques objectively and identify the simplest solution that satisfies the application requirements.

Evaluation should also focus on the details of the remaining failures, not just a single overall score. An overall evaluation score is not sufficient for identifying the best next step as discussed in the previous section. On the other hand, a small improvement in the overall score might justify new deployment if the customization done resolves the failure modes that matter most to the application. *Table 8.11* summarizes the evaluation guidance at every stage of the customization workflow.

Customization Stage	Evaluation Goal	Example Success Criteria	Next Decision
Inference-time controls	Verify that prompts and examples produce the expected behavior	Instruction following, response quality	Deploy or add RAG

Customization Stage	Evaluation Goal	Example Success Criteria	Next Decision
Retrieval and grounding	Verify that retrieval provides the required information	Grounded responses, citation accuracy, retrieval quality	Deploy or scale the customizatio n data
Scaling the customization data	Verify that additional data closes the remaining gaps	Coverage, task completion, behavioral consistency	Deploy or escalate to post-training
Post-training	Verify that learned behavior satisfies the application requirements	Behavioral consistency, human agreement, task success	Deploy or continue improving the model

Table 8.11: Using evaluation to guide customization decisions

Not every application has to progress through every stage of the customization workflow. Many applications satisfy their customization requirements after improving prompts or applying RAG and can proceed to deployment when they meet the application-specific acceptance criteria according to evaluation. Others require post-training to satisfy the application requirements. The objective is not to use more customization techniques. The objective is to stop when evaluation demonstrates that the current solution is sufficient.

The previous sections established the customization workflow and the engineering decisions behind each stage. The next section implements this workflow with an end-to-end example and shows how the customization techniques work together for building a specialized assistant.

Implementing the Customization Workflow

The previous sections introduced the customization workflow and the engineering decisions behind each stage. This section applies these concepts within a prototype enterprise assistant. The prototype would not be a production-ready system, but it shows the implementation of the customization workflow and demonstrates the different customization mechanisms working together.

The implementation follows the same sequence introduced throughout the chapter. You begin by defining the target behavior of the assistant, building the required customization data, integrating RAG, and finally discussing evaluation and deployment. The result is a prototype enterprise assistant that is ready for initial deployment.

The complete code is available in `Chapter_8_Notebook.ipynb` in the companion GitHub repository. The code snippets included throughout this section highlight the key implementation steps, while the notebook provides the complete prototype implementation.

Although the implementation focuses on a prototype, we also discuss the additional engineering work required for production systems. Topics, such as scalability, monitoring, security, automation, and operational reliability are intentionally simplified in the prototype implementation, so the customization workflow remains the primary focus.

Defining target behaviors

Every customization project starts by defining the expected assistant behavior. This specification is the foundation for the remaining customization steps. The specification defines the assistant's responsibilities, the tasks supported, and the expected response characteristics. These specifications guide the data curation process, the inference-time controls, and the evaluation process later in the workflow.

For this prototype, you build an enterprise assistant that answers employee questions using enterprise documentation and structured business data. The assistant should answer factual questions, summarize enterprise information, recommend actions when appropriate, and avoid unsupported assumptions.

The first implementation artifact is a structured behavior specification that captures the assistant requirements. Here is an example.

```
# Define the enterprise assistant behavior

assistant_specification = {
    "assistant_name": "Enterprise Assistant",

    "supported_tasks": [
        "Answer enterprise questions",
        "Summarize enterprise documents",
        "Recommend next actions"
    ],

    "knowledge_sources": [
        "Enterprise documentation",
        "Structured enterprise records"
    ],

    "response_requirements": {
        "grounded": True,
        "cite_sources": True,
        "avoid_speculation": True
    },

    "evaluation_metrics": [
        "answer_correctness",
        "groundedness",
        "citation_accuracy"
    ]
}
```

The behavior specification acts as the contract for the customization workflow. Every implementation decision in the following sections traces back to this specification. The list of supported tasks determines the data to be collected. The response requirements influence the system prompt and few-shot examples. The evaluation metrics guide the evaluation process to use the business metrics that matter and control deployment decisions.

The next step is to author the system prompt that establishes the assistant's global behavior. Unlike the system prompt, task instructions describe the general steps or sub-tasks required to be performed by the assistant for a particular user request. The instructions remain consistent for similar tasks while the user request changes from one interaction to another. If the assistant is required to perform multiple different tasks, the task prompt can be replaced during inference time based on user selection or external parameters.

```
# Create the system prompt
system_prompt = (
    "You are an enterprise assistant. Answer accurately, "
    "protect confidential information, and rely on approved "
    "enterprise sources when context is provided."
)

# Create the task instructions
task_prompt = (
    "Answer the user's request directly and concisely. "
    "Follow the demonstrated behavior, use retrieved context when "
    "available, and state when the available evidence is insufficient."
)

# Chapter_8_Notebook.ipynb: Assembling Inference-Time Controls
```

For now, the prompts intentionally contain only instructions. They do not yet include retrieved information or few-shot examples. These components will be added later in the next few sections.

This prototype implementation stores the behavior specification and prompts directly in the notebook for simplicity. Production systems separate these artifacts from the application code. Many organizations manage assistant specifications, prompts, and supported tools using configuration files, prompt management systems, or centralized repositories. Incorporating these systems simplifies experimentation, versioning, and deployment across assistant versions.

With the target behavior defined, the next step is curating the enterprise data required to support the target behaviors. The implementation includes knowledge documents, behavioral examples, structured enterprise records, and any synthetic data prepared using the techniques presented in *Chapter 6*.

Curating customization data

Now that you defined the target behavior of the enterprise assistant, the next step is to curate the customization data required to support this behavior. The data includes knowledge data that provides enterprise information, behavioral data that demonstrates representative workflows, and preference data that defines the desired response properties. When additional coverage is required, these datasets are expanded using the synthetic data generation techniques presented in *Chapter 6*.

The implementation starts by loading the different data sources into a common representation. Although these datasets originate from different systems, they should follow a consistent structure to support reusability through the customization workflow. This structure unification includes assigning common identifiers, preserving source information, and attaching common metadata such as document version, authority, and timestamps whenever applicable.

```
# Curate customization data into a common representation.
customization_data = []

for document in curated_knowledge:
    customization_data.append({
        "id": document["document_id"],
        "category": "knowledge",
        "content": document["content"],
        "metadata": {
            "source": document["source_system"],
            "authority": document["owner"],
            "version": document["version"],
            "last_updated": document["last_reviewed"],
            "synthetic": False,
        },
    })

for interaction in behavioral_examples:
    customization_data.append({
        "id": interaction["example_id"],
        "category": "behavioral",
        "content": {
            "messages": to_messages(interaction["events"]),
            "workflow": extract_workflow(interaction["events"]),
            "outcome": interaction["outcome"],
        },
        "metadata": {
            "source": interaction["channel"],
            "authority": "validated successful interaction",
            "synthetic": False,
            ...
        },
    })

for comparison in preference_examples:
    customization_data.append({
        "id": create_preference_id(comparison),
        "category": "preference",
        "content": {
```

```
            "prompt": comparison["instruction"],
            "preferred_response": comparison["preferred_response"],
            "rejected_response": comparison["non_preferred_response"],
        },
        "metadata": {
            "source": "preference_pairs_dataset.csv",
            "authority": "synthetic preference generation",
            "synthetic": True,
            ...
        },
    })

# Chapter_8_Notebook.ipynb: Curating Customization Data into a Common Representation
```

The unified representation simplifies the remaining implementation. Every customization record now follows the same structure regardless of its source. The next step transforms these records into the implementation artifacts required by the enterprise assistant. Knowledge data forms the retrieval corpus used by RAG. Behavioral data is filtered, ranked, and organized into candidate few-shot examples. Preference data contributes to evaluation and future post-training if additional customization is necessary. The preferred responses can also be selected as few-shot examples when they demonstrate the desired behavior effectively.

```
# Build inference-time assets.
retrieval_corpus = []
few_shot_candidates = []

for record in customization_data:
    if record["category"] == "knowledge":
        retrieval_corpus.append({
            "id": record["id"],
            "content": record["content"],
            "metadata": {
                "source": record["metadata"]["source"],
                "authority": record["metadata"]["authority"],
                "last_updated": record["metadata"]["last_updated"],
            },
        })

    elif record["category"] == "behavioral":
        few_shot_candidates.append({
            "conversation": record["content"]["messages"],
            "workflow": record["content"]["workflow"],
            "quality_score": calculate_quality(record),
            "synthetic": False,
        })
```

```
    elif record["category"] == "preference":
        few_shot_candidates.append({
            "conversation": [
                {"role": "user", "content": record["content"]["prompt"]},
                {
                    "role": "assistant",
                    "content": record["content"]["preferred_response"],
                },
            ],
            "quality_score": 0.8,
            "synthetic": record["metadata"]["synthetic"],
        })

few_shot_candidates.sort(
    key=lambda item: (
        item["quality_score"],
        not item["synthetic"],
    ),
    reverse=True,
)

# Chapter_8_Notebook.ipynb: Building Inference-Time Assets
```

The implementation artifacts are intentionally different from the original customization data. Each artifact is optimized for a specific stage of the customization workflow. This prototype implementation performs these preparation steps directly inside the notebook. Production systems typically automate this process as part of the data ingestion pipeline. New documents, interactions, and preference data are continuously validated, enriched with metadata, and transformed into updated retrieval corpora.

The same customization data is also reused beyond inference-time customization. Evaluation and post-training consume many of the same records but organize them differently according to their purpose.

```
# Reuse customization data for evaluation and post-training.
evaluation_dataset = []
post_training_dataset = {
    "supervised": [],
    "preference": [],
}

for record in customization_data:
    if record["category"] == "knowledge":
        evaluation_dataset.extend(
            generate_evaluation_questions(record)
```

```
        )
        post_training_dataset["supervised"].extend(
            knowledge_to_supervised_examples(record)
        )

    elif record["category"] == "behavioral":
        evaluation_dataset.append(
            workflow_to_evaluation_case(record)
        )
        post_training_dataset["supervised"].append(
            interaction_to_supervised_example(record)
        )

    elif record["category"] == "preference":
        evaluation_dataset.append(
            preference_to_evaluation_case(record)
        )
        post_training_dataset["preference"].append(
            preference_pair(record)
        )

# The notebook validates and exports each resulting dataset.
...

# Chapter_8_Notebook.ipynb: Reusing Data for Evaluation and Post-Training
```

The prototype implementation uses only the inference-time assets prepared earlier. The evaluation and post-training datasets are included to illustrate data reusability across the customization workflow. The evaluation dataset is used as a deployment gate during model development cycles, while the post-training datasets could be relevant based on evaluation results.

The curated customization data is now ready for implementation. The next section selects representative behavioral data that forms the few-shot examples used by the enterprise assistant during inference.

Selecting few-shot examples

The objective of few-shot example selection is to demonstrate the expected assistant behavior using the smallest number of high-quality examples. A common mistake is to select the examples with the highest quality scores. Selecting the highest-scoring examples may produce examples that demonstrate the same workflow while leaving other important scenarios uncovered. Few-shot examples should represent a diversity of user requests

rather than a single interaction. A small collection of complementary examples usually produces better results than many similar examples.

The implementation starts by ranking the candidate examples according to the quality score prepared earlier. Then, a selection is applied to increase coverage of different workflows and user requests while avoiding duplicate examples.

```
# Select a small, diverse set of few-shot examples.
few_shot_examples = []
formatted_examples = []
coverage = set()
selected_requests = []
used_tokens = 0

for example in sort_by_quality(few_shot_candidates):
    coverage_key = get_coverage_key(example)
    request = get_user_request(example)
    formatted = format_few_shot_example(example)

    # Avoid repeated workflows and similar user requests.
    if coverage_key in coverage:
        continue
    if is_near_duplicate(request, selected_requests):
        continue

    example_tokens = count_message_tokens(formatted)
    if used_tokens + example_tokens > FEW_SHOT_TOKEN_BUDGET:
        continue

    few_shot_examples.append(example)
    formatted_examples.append(formatted)
    coverage.add(coverage_key)
    selected_requests.append(request)
    used_tokens += example_tokens

    if len(few_shot_examples) == 5:
        break

# The notebook validates and exports the selected examples.
...

# Chapter_8_Notebook.ipynb: Selecting Few-Shot Examples
```

This implementation demonstrates three practical selection criteria. The first is quality, where examples with successful outcomes and complete workflows receive higher priority. The second is diversity, where each example

introduces a different workflow or user scenario instead of repeating the same patterns. The third is the prompt budget. Because every additional example consumes tokens, only the examples that provide new behavioral information should be included. After selecting the final examples, the conversations are converted into the format that matches the prompt template. Each example contains a representative user request followed by the corresponding assistant response.

The prototype implementation selects a small number of examples directly inside the notebook. Production systems maintain prompt components separately and update them independently. They also maintain multiple few-shot collections optimized for different tasks and scenarios and select the appropriate collection dynamically during inference.

The selected few-shot examples are now ready to be used during inference. Together with the system prompt and task instructions prepared earlier, they define the expected assistant behavior during inference. The next section assembles these components into a reusable prompt template that will later be extended with retrieved enterprise knowledge.

Assembling inference-time controls

The previous sections produced the individual components required during inference. This section assembles the system prompt, task instructions, and the selected few-shot examples into a reusable prompt template that will be used for every request processed by the enterprise assistant.

The purpose of the prompt template is to keep the prompt structure consistent while allowing individual components to change independently. The system prompt and task instructions remain stable across similar requests. Task instructions and few-shot examples change for different tasks or when the behavioral targets are updated. The user request changes for every interaction. The retrieved context is intentionally left empty at this stage because the RAG pipeline has not yet been integrated. The following is an example of a reusable prompt template:

```
# Assemble the reusable inference-time structure
prompt_template = {
    "system_prompt": system_prompt,
    "task_instructions": task_prompt,
    "retrieved_context": None,
```

```
    "few_shot_examples": formatted_examples,
    "user_request": "{user_request}"
}

# Build the inference request
def build_prompt(user_request):
    prompt = prompt_template.copy()
    prompt["user_request"] = user_request
    return prompt

# Chapter_8_Notebook.ipynb: Assembling Inference-Time Controls
```

The prompt template defines the complete structure of every inference request, but one component is still missing. The **`retrieved_context`** placeholder will later contain the enterprise knowledge returned by the retrieval pipeline. Leaving this placeholder in the template simplifies the implementation because the retrieval mechanism can evolve independently from the rest of the prompt.

This implementation separates the static and dynamic components of the prompt. The system prompt, task instructions, and few-shot examples are reused across many requests, while the user request is supplied dynamically during inference. The retrieved context remains empty until the retrieval pipeline is integrated in the next section.

The prototype implementation stores the prompt template directly in the notebook for simplicity. Production systems usually maintain prompt templates separately from the application code. Prompt management systems allow teams to version prompts, perform controlled updates, and experiment with different prompt variants without modifying the application itself.

Some enterprise information is better accessed through tools than through RAG. For this prototype, enterprise documents are retrieved through RAG, while structured business records are accessed through an external tool. The following example defines a simple tool for retrieving authorized employee information:

```
# Define the external tools available to the assistant

tools = [
    {
        "name": "get_employee_record",
        "description": "Retrieve authorized employee information",
```

```
        "parameters": ["employee_id", "field"],
    }
]

# Assemble the inference request

inference_request = {
    "prompt": build_prompt(user_request),
    "tools": tools,
}
```

The prompt controls model behavior, while the tool definitions provide external enterprise capabilities when they are required by the user request.

The inference-time controls are now assembled into a reusable prompt template. The next step is connecting the retrieval pipeline to populate the retrieved context with relevant enterprise knowledge before sending the prompt to the model.

Integrating retrieval and grounding

The previous section assembled the inference-time controls into a reusable prompt template, but the **`retrieved_context`** component intentionally remained empty. This section completes the prototype enterprise assistant by integrating the retrieval pipeline and populating that placeholder before each inference request.

Chapters four and five discussed RAG in detail, including document chunking, indexing strategies, retrieval algorithms, and production-scale optimizations. This implementation focuses on connecting these components to the enterprise assistant rather than rebuilding the retrieval pipeline.

The retrieval corpus prepared earlier in this chapter is the input to the retrieval pipeline. During indexing, each document is divided into searchable chunks while preserving the metadata required during retrieval. This metadata supports filtering, ranking, citations, and access control without modifying the underlying prompt template.

The implementation details of chunking, indexing, and retrieval optimization were covered in Chapters four and five. The important point here is that the retrieval pipeline consumes the retrieval corpus and exposes a simple retrieval interface to the model. During inference, the model retrieves the most relevant enterprise knowledge for the current user request.

The retrieved documents are not returned directly to the user. Instead, they populate the **`retrieved_context`** placeholder. This keeps the retrieval pipeline independent of the prompt template. Improvements to retrieval quality, indexing strategy, or ranking algorithms do not require changes to other inference-time controls:

```
# Reuse the persistent vector store established in earlier chapters.
collection = chroma_client.get_or_create_collection(
    name="chapter8_enterprise_rag"
)

# Retrieve relevant knowledge, capped by the collection size.
retrieved_documents = search_enterprise_knowledge(
    query=user_request,
    top_k=5,
)

# Populate a request-specific copy while preserving the base template.
prompt = build_prompt(user_request)
prompt["retrieved_context"] = format_documents(retrieved_documents)

# Chapter_8_Notebook.ipynb: Integrating Retrieval and Grounding
```

With this update, every inference request follows the same structure. Production systems typically introduce additional RAG integration capabilities, such as metadata filtering, document reranking, document-level access control and content caching. These improvements increase retrieval quality and scalability while preserving the same prompt template and inference workflow.

With RAG integrated, the enterprise assistant can access unstructured enterprise knowledge through retrieval and structured business information through the tools. The grounded prompt includes the relevant retrieved knowledge and metadata before it is sent to the model. Production systems usually extend this workflow with document-level access-control filters, document reranking and content caching.

The prototype enterprise assistant is assembled now. The next section evaluates the assistant using the evaluation dataset prepared earlier. The evaluation determines whether the current customization workflow is sufficient for deployment or whether additional customization, including post-training, is required.

Deploying the customized assistant

The next step for the customized assistant is to verify that it meets the application requirements before deployment. Rather than measuring general model quality, the evaluation focuses on the target behaviors defined at the beginning of this implementation. The same evaluation dataset, metrics, and acceptance thresholds prepared earlier are reused to determine whether the current customization workflow is sufficient.

The implementation begins by evaluating the assistant using the prepared evaluation dataset:

```
# Define the minimum requirements for the inference pipeline.
acceptance_thresholds = {
    "knowledge_evidence_recall": 0.9,
    "prompt_readiness_rate": 0.95,
}

# Evaluate retrieval and prompt assembly using the prepared cases.
evaluation_results, evaluation_metrics, meets_requirements = (
    evaluate_inference_pipeline(
        evaluation_dataset,
        acceptance_thresholds,
    )
)

# Passing advances the assistant to held-out response evaluation.
if meets_requirements:
    decision = "READY FOR HELD-OUT MODEL-RESPONSE EVALUATION"
else:
    decision = "HOLD: ANALYZE REMAINING FAILURES"

print("Metrics:", evaluation_metrics)
print("Decision:", decision)

# Chapter_8_Notebook.ipynb: Evaluating the Customized Assistant
```

The evaluation results determine the deployment decision. If the assistant satisfies the predefined acceptance criteria, it is ready for deployment. Otherwise, the remaining failure modes should be analyzed before introducing additional customization.

Evaluation should guide the next engineering decision rather than simply produce a score. The failure modes determine which stage of the customization workflow should be revisited. For example, missing or

outdated information indicates the need to expand the retrieval corpus. Poor coverage of user requests or inconsistent workflows indicates the need for refining the instructions and few-shot examples. Persistent behavioral failures after improving prompts and retrieval indicate that post-training might be required.

This prototype implementation stops after assembling and evaluating the inference-time customizations. Many enterprise assistants do not need to progress beyond this point because prompting, retrieval, and high-quality customization data already satisfy the application requirements. When required, the next step is to introduce post-training using the techniques discussed in *Chapter 7*. The customization data prepared throughout this implementation is reused for supervised fine-tuning, preference optimization and reinforcement learning.

Production systems include additional engineering work beyond deployment, such as monitoring, telemetry, prompt and model versioning, retrieval index management, automated evaluation pipelines, staged rollouts, and rollback mechanisms. These capabilities improve reliability and operational efficiency without changing the customization workflow itself.

The same customization workflow discussed in this section can be applied to other AI assistants, but the required data assets and customization techniques might vary according to the application requirements. The next section revisits these customizations across multiple assistant types.

Customization patterns for AI assistants

The customization workflow introduced in this chapter applies to the four assistant types discussed throughout this book. Every assistant needs definition of the target behavior, preparing the required customization data, applying inference-time controls, and evaluating the results before considering post-training. The customization workflow is consistent between assistants, but the supporting customization data could be different.

Different assistants depend on different forms of knowledge, follow different workflows, and operate under different constraints. Thus, they prioritize different customization techniques and customization data. *Table 8.12*

summarizes the typical customization patterns for the different assistants discussed throughout this book.

Assistant	Primary Data Assets	Primary Customizations
Personal Assistant	Personal context, external knowledge, user preferences	Prompting, RAG, tools
Enterprise Assistant	Enterprise documentation, structured records, behavioral examples	Prompting, RAG, tools
Domain-Specific Assistant	Domain knowledge, expert demonstrations, preference data	Prompting, RAG, selective post-training
Content Moderation	Policies, preference data, feedback data	Prompting, preference optimization, reinforcement learning

Table 8.12: Customization patterns across common AI assistants

Personal assistants typically depend on external information that changes continuously, such as calendars, emails, weather, news, and user preferences. Their primary customization challenge is providing relevant and personalized responses. As a result, inference-time customization, RAG, and tool integration usually provide sufficient capabilities without modifying the underlying model.

Enterprise assistants focus on organizational knowledge and business workflows. The best sources for this information are internal documentation, structured enterprise system records, and conversational interactions. Retrieval and grounding are usually the dominant customization techniques because enterprise knowledge changes independently of the model development cycles. Comprehensive few-shot examples usually address most of the customization goals without post-training.

Domain-specific assistants in domains, such as healthcare, finance, and law depend on domain terminology, expert reasoning and company policies. RAG provides access to current authoritative references. Additionally, many

organizations invest in representative behavioral examples and selective post-training to improve adherence to enterprise workflows. These assistants are more likely to benefit from post-training because they perform well-defined and complex tasks repeatedly.

Content moderation systems need a different customization profile. Acceptable responses usually depend on expert judgment, and many moderation decisions balance between competing objectives. Preference data, feedback signals, and reinforcement learning play a critical role. Also, these systems rely heavily on synthetic edge cases and comprehensive evaluation datasets to verify policy compliance before deployment.

Although these assistants focus on different customization techniques, they follow the same customization process discussed in this chapter. They begin with inference-time customization, expand through RAG and better customization data, evaluate the results, and last escalate to post-training when required. The customization workflow is consistent, but the customization data assets and customization priorities adapt to the application requirements.

Summary

In this chapter, you learned that LLM customization is a system-level process rather than a training technique. You discovered the dimensions of domain specialization and the customization data required to support them. Then, you organized this data into reusable knowledge, behavioral, preference, and feedback data assets that serve multiple customization techniques. You also visited the evaluation-driven customization workflow that starts with inference-time controls, expands through retrieval and better data, and progresses to post-training only when justified through evaluation. Finally, we implemented this workflow by building a prototype enterprise assistant and showed how the same customization principles apply across different AI assistants.

These topics are important because most successful LLM applications are often customized through data and system design rather than model training alone. Understanding the relationship between customization data, inference-time controls, retrieval, evaluation, and post-training helps

development teams build assistants that are easier to maintain and improve over time. More importantly, it provides a practical engineering framework for deciding which customization technique to apply, when to apply it, and when to stop.

Evaluation is at the core of the customization workflow and acts as the gate for moving between its steps. The next chapter focuses on building representative evaluation datasets, selecting meaningful metrics, and measuring the performance of specialized assistants. These evaluation techniques validate every customization decision discussed throughout this chapter.

References

- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-rank adaptation of large language models*. **arXiv**. https://arxiv.org/abs/2106.09685
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). *Training language models to follow instructions with human feedback*. Advances in Neural Information Processing Systems. https://arxiv.org/abs/2203.02155
- Microsoft. (2026). *Augment LLMs with RAGs or fine-tuning*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/developer/ai/augment-llm-rag-fine-tuning
- Wolff, H., Rohrer, D., & Liu, J. (2023). *RAG 101: Retrieval-augmented generation questions answered*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/rag-101-retrieval-augmented-generation-questions-answered/
- Xu, X., Weytjens, H., Zhang, D., Lu, Q., Weber, I., & Zhu, L. (2025). *RAGOps: Operating and managing retrieval-augmented generation pipelines*. arXiv. https://arxiv.org/html/2506.03401v1
- Grid Dynamics. (2023). *RAG and LLM business process automation: A technical strategy*. https://www.griddynamics.com/blog/retrieval-augmented-generation-llm

- Aerospike. (2025). *RAG for enterprise AI systems*. https://aerospike.com/blog/retrieval-augmented-generation-enterprise-ai
- MDPI. (2025). *Retrieval-augmented generation and large language models (LLMs) in enterprise knowledge management: A systematic review*. Applied Sciences, 16(1), 368. https://www.mdpi.com/2076-3417/16/1/368

Part 4

Building and Maintaining Reliable LLM Systems

The final part of this book brings together the practices needed to assess and maintain system reliability. You will build representative evaluation datasets and use the results to investigate failures. You will then trace data quality problems to their sources, apply corrections, and combine automated checks with human oversight. Finally, you will explore context engineering, organizing project knowledge so that an agent starts each session with current facts, rules, and past decisions. By the end of this part, you will understand how evaluation and production feedback guide improvements throughout the data lifecycle.

This part of the book includes the following chapters:

- *Chapter 9, Task-Specific Evaluation Data*
- *Chapter 10, Data Quality and Human Oversight*
- *Chapter 11, Context Engineering: Data Strategy at Agent Runtime*

9

Task-Specific Evaluation Data

I witnessed many teams skip evaluation. I keep coming back to it because the failures are always the same pattern: you build a retrieval pipeline and fine-tune a model, deploy the whole thing, and two weeks later support tickets start rolling in. The model scored great on your 200-question internal test set. Production queries look nothing like those 200 questions. Engineers write test questions about what they think matters; users ask about what actually matters to them. Not the same thing at all. In *Chapters 4* through *8*, we built all the pieces: retrieval, synthetic data, fine-tuning, the works. When I started building AI systems, I tried multiple approaches to avoid hallucinations and improve context. Back in 2024, I wrote about these hallucination mitigation strategies [1]. I proposed RAG to ground the model in real data, fine-tuning to narrow its scope, prompt engineering to steer it, and human-in-the-loop to catch what automation misses. This chapter is where we find out if those strategies actually worked. Honestly, you do not know until you measure.

At Clearwater, we use two main ways of evaluation: LLM-as-a-judge (or multiple judges) and human review. LangChain surveyed more than 1,300 AI professionals in late 2025, and human review came out on top at 59.8 percent, with LLM-as-a-judge right behind at 53.3 percent, `https://www.langchain.com/state-of-agent-engineering`, and offline testing on a fixed set at 52.4 percent [2]. The totals go past a hundred because teams do not pick one method; they layer them. So human review is still the most common on its own, but it is slow, and you cannot put a person on every answer. That is why the judge ends up doing most of the scoring, and the human is saved for the parts that matter. This is why teams end up running both. Dan Siddall's framework at Clearwater Analytics is a good example. The judges do the bulk scoring, and a human is only pulled in when two of them

disagree by more than 0.3 on an attribute. https://medium.com/cwan-engineering/generating-diverse-questions-for-llm-instruction-fine-tuning-strategies-and-considerations-73270f2b0adb Building the golden set the same way Dan describes in his article is what keeps the judge honest [3].

In this chapter, I will focus on the LLM-as-a-judge approach, as it is the most scalable and cost-effective way to evaluate your AI system with examples. As usual, I will explain the foundation to understand the concept. New methods will come every day, most of which will be based on this simple idea. For example, at Clearwater, Dan used a group of judges, which is an enhanced LLM-as-a-judge with a rubric and an ensemble of judges. Other variants include pairwise comparison (Elo-style ranking), rubric-based scoring (G-Eval), and jury-of-judges (three independent judges, majority vote, reduces bias at 3x cost). If you understand and practice the foundational method LLM-as-a-judge, you will be able to adapt to these new methods easily.

I will also focus on evaluating what we built in the previous chapters. Along the way, we will evaluate the RAG system we built in the first half of the book (*Chapters 1* to *5*). We will also validate the synthetic data we created in *Chapter 6* and evaluate the fine-tuning we set up in *Chapter 7*.

In this chapter, we will cover the following topics:

- Using LLM-as-a-judge
- Evaluating the RAG pipeline
- Evaluating and using *Chapter 6*'s synthetic data
- Evaluating the fine-tuned model from *Chapter 7*
- Evaluating live production data
- Building task-specific evaluation datasets
- Selecting evaluation metrics by assistant type
- How to evaluate evaluation platforms

Technical requirements

This chapter builds on almost everything we built earlier, so you already have most of what you need. You need Python 3, the OpenAI library (`pip install openai`), ChromaDB (`pip install chromadb`) for the vector store, and RAGAS (Retrieval Augmented Generation Assessment, pip install ragas) [4] for the

automated RAG metrics, plus a valid API key. The fine-tuning delta near the end also needs PyTorch and Transformers (pip install torch transformers) to load the local model that *Chapter 7* saves. If you have not set up a local environment yet, follow the instructions in *Chapter 1*. However, the easiest way is to use Google Colab and skip local setup. Also, coding assistants, like Claude, can help you with the setup and save time. Using a coding assistant for this kind of setup work is common practice now.

The full code is in the `chapter_09` folder at `chapter_09/Jupyter_Notebooks/Chapter_9_Notebook.ipynb`. You can open it directly from GitHub the way we did in earlier chapters. The setup cells at the top reuse the environment from the earlier notebooks, so run those first before you jump to any section.

This chapter is different from the earlier ones in one way: its notebook depends on what a few earlier notebooks produced, so run *Chapter 6* and *Chapter 7* before this one. *Chapter 3* is optional here, needed only for the optional RAG evaluation section, where it creates the shared ChromaDB store that section reads from. I decided to make it optional for people not interested in the ChromaDB setup. However, it is very important to get the basics of RAG and its evaluation. *Chapter 6* produces the synthetic datasets we grade here: the QA pairs, preference pairs, and augmented text. *Chapter 7* produces two fine-tuned artifacts, an SFT (supervised fine-tuning) model and a **Low-Rank Adaptation** (**LoRA**) adapter. Both train locally and save to disk, and this notebook loads them from the repo folders. Training locally is a deliberate choice. When we started this book, hosted fine-tuning through the OpenAI API was the obvious route, but OpenAI announced in May 2026 that it is winding down its self-serve fine-tuning platform, with all new job creation ending in January 2027 [5]. So the training runs on your own machine instead, and nothing in this chapter depends on a hosted fine-tuning account. You still need an API key, but only for the model and judge calls. If you have not run the *Chapter 7* training, this chapter still works. The notebook evaluates the base model against the same behavioral contract, and the delta cell picks up the local fine-tuned model automatically whenever its saved folder exists. Each section tells you which cell to jump to, so you can read and run the code side by side.

Using LLM-as-a-judge

An LLM as a judge is exactly what it sounds like. You take the answer from one model and hand it to another strong model to grade. The judge gets three things: the question, the reference answer to score, and a short rubric that says what a good answer looks like. It reads them and returns a score with a reason. The following function is an example of an LLM-as-a-judge prompt in its simplest form:

```
def judge(question, answer, reference):
    prompt = f"""You are an evaluation judge. Score the answer against the reference
on a 1-5 scale, where 5 means it matches the reference and 1 means it is wrong.

Question: {question}
Reference answer: {reference}
Answer to score: {answer}

Return ONLY valid JSON: {{"score": N, "reason": "one sentence"}}"""

    response = client.chat.completions.create(
        model=JUDGE_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.0,
    )
    return response.choices[0].message.content
```

The reason that this method became popular is that the old metrics could not do this. **Recall-Oriented Understudy for Gisting Evaluation** (**ROUGE**) and **Bilingual Evaluation Understudy** (**BLEU**) only count word overlap against a reference answer. They do not read for meaning. A good answer written in different words scores low; a wrong answer that copies the reference words scores high. I have been burned both ways. The judge reads for meaning, the way a person would. We hit exactly this at Clearwater Analytics, and Dan Siddall captured it with a clean example in his article [6]. The question asked which schedule a CLO (collateralized loan obligation) should be modeled on under **NAIC** (short for **National Association of Insurance Commissioners**) rules. The right answer is Schedule D. The model said Schedule C, which is just wrong. But every other word in the two answers lined up, so ROUGE scored it 93 percent, and BLEU scored it 92 percent. Two metrics called a wrong answer almost perfect, because the one word that decided everything

was buried under a pile of words that matched. A judge that reads for meaning catches the wrong schedule right away [6].

Why it matters is cost and speed. A person is the most accurate grader we have, but people are slow and expensive, and you cannot sit one behind every answer. The judge agrees with human graders around 80 percent of the time and costs almost nothing next to a person. Good enough to trust for most of the work, cheap enough to run on thousands of answers a day. It is not perfect, though. The judge carries its own biases. It leans toward the longer answer, or the first one it saw, or an answer from its own family of models. I come back to the first-position and own-family leans later in the chapter, with a check for each. This is the reason Dan used a group of judges instead of just one.

Let's explain this through a simple example from the Notebook. The notebook for this chapter starts with an LLM-as-a-judge evaluation example you can run yourself. In the Notebook, search for `"# === The simplest LLM-as-a-judge ==="`. Make sure to run all the cells before it for setup, then run this cell and the one that follows it. We created the function explained above. Now let's use the example that Dan used in his article. We will give it two hardcoded answers, the right answer and a wrong one:

```
# One question, one reference answer.
question = "Under NAIC rules, which schedule should a CLO be modeled on?"
reference = "CLOs should be modeled on Schedule D."

# A correct answer, worded very differently from the reference.
good_answer = "For NAIC reporting, collateralized loan obligations belong on Schedule D."

# A wrong answer that reuses almost the exact same words (only the schedule letter changed).
wrong_answer = "CLOs should be modeled on Schedule C."

print("GOOD answer (different words, same meaning):")
print("  ", judge(question, good_answer, reference))

print("\nWRONG answer (almost identical words, wrong schedule):")
print("  ", judge(question, wrong_answer, reference))
```

As expected, the LLM scored the good answer high and the wrong one very low:

```
GOOD answer (different words, same meaning):
   {"score":5,"reason":"The answer correctly states that CLOs should be reported on Schedule D under
```

```
NAIC rules."}

WRONG answer (almost identical words, wrong schedule):
   {"score":1,"reason":"The answer is incorrect because CLOs should be modeled on Schedule D, not
Schedule C."}
```

This is the LLM-as-a-judge in action. It understands that the good answer is correct even though it uses different words, and it penalizes the wrong answer for being factually incorrect. In this example, I used a hardcoded answer to focus on explaining how the judge function works. In the following example, I will test LLM answers for real. I will create five test cases, fixed questions with known correct answers, and run them through two different prompts. Prompt A gives the model the policy text as context. Prompt B drops the context and just asks the question. Same five queries, same model, same judge. The only thing that changed was the prompt. Here is the process I used in this example. I kept each step in its own cell in the Notebook so you can change things and try it yourself to see the difference.

Step 1: building the gold dataset

I start with the golden dataset, the fixed set of questions I already trust the answers to. Each entry is a small dictionary holding the question, its reference answer, and a couple of tags. The notebook has all five; here is the first one so you can see the shape the rest follow:

```
golden_dataset = [
    {
        "query": "What is the refund policy for digital products?",
        "reference_answer": ("Digital products can be refunded "
                             "within 14 days if not downloaded."),
        "category": "policy",
        "difficulty": "easy"
    },
    # ... 4 more test cases
]
```

This is your unit test suite. Five fixed questions with known correct answers, and we add them to the `golden_dataset` list. You tag each one with category and difficulty so you can slice the results later—how did we do on edge cases, did we regress on hard questions? You can build those golden sets using the techniques we explained in *Chapter 6*. I am just hardcoding simple examples to focus on explaining the LLM-as-a-judge.

Step 2: defining two prompts to compare

Have a look at the following code. Prompt A gives the model the reference answer as policy context in the `reference` field. This is the in-context learning we explained in *Chapter 1*. In production, RAG will insert its context into the prompts, as we explained in *Chapters 1* to *5*.

```
PROMPT_A = """You are a customer support assistant. Answer the question using ONLY the provided
policy.
If the answer is not in the policy, say "I don't have that information."

Policy: {reference}

Question: {query}"""
```

Prompt B gives the model nothing, just the question:

```
PROMPT_B = """Answer this customer question.
Question: {query}"""
```

Same model, same judge, same questions. The only variable is the prompt. So far, those are the calls to the LLM. Next, we will evaluate the answers they returned.

Step 3: running both prompts through the evaluation function

For each test case, the function does three things: formats the prompt with the question and policy if using Prompt A, calls the LLM to get a (the notebook discovers and tests a low-cost model your key can use, so it does not go stale), and calls the judge to score the response against the reference on a 1 to 5 scale. So, for 5 questions times 2 prompts, that is 10 model calls plus 10 judge calls, 20 API calls total.

Let me walk through one test case end to end so you can see exactly what happens at each step. We will use the first question: "What is the refund policy for digital products?" The reference answer in our golden dataset is "Digital products can be refunded within 14 days if not downloaded."

Prompt A tests context adherence, sometimes called faithfulness, groundedness, or attribution depending on the framework. The idea is the same across all of them: did the model use what you gave it instead of making

things up? This context can come from in-context learning methods such as RAG. The policy text is right there in the prompt, and the model does not need to know anything from its training data. It just needs to read the policy and answer based on it. This is the same pattern as RAG: a retrieval system fetches relevant documents and stuffs them into the prompt, and the model's job is to read what it was given and answer faithfully. Prompt A simulates that. So, what we are actually measuring is: can the model follow instructions, can it extract the right information from the context, and does it hallucinate or refuse when it should not? Prompt B is the control, same questions with no context. The delta between A and B shows you the value of providing context.

First, the function takes Prompt A and fills in the placeholders. The {reference} becomes the reference answer, and the {query} becomes the question. This is what actually gets sent to the model:

```
You are a customer support assistant. Answer the question using ONLY the provided policy.
If the answer is not in the policy, say "I don't have that information."

Policy: Digital products can be refunded within 14 days if not downloaded.

Question: What is the refund policy for digital products?
```

The reference answer is right there in the policy field. The model should be able to read it and give it back. The model reads this and generates a response. Here is the main call to the LLM where we send the previous prompt:

```
resp = client.chat.completions.create(
    model=BASE_MODEL,
    messages=[{"role": "user", "content": prompt}],
    temperature=0.0
)
```

In my run, the model read the policy and handed it straight back, word for word, as it turned out:

```
=== Prompt A: With policy context ===
[A] Query: What is the refund policy for digital products?
Model answer: Digital products can be refunded within 14 days if not downloaded.
```

Now the judge steps in. The function builds a second prompt that gives the judge three things: the original question, the reference answer we know is

correct, and the response the model just produced. Here is what the judge actually receives:

```
judge_prompt = f"""Score this response against the reference on a 1-5 scale.
1 = completely wrong, 5 = matches reference perfectly.

Question: {item['query']}
Reference: {item['reference_answer']}
Response: {model_answer}

Return ONLY a JSON object: {{"score": N, "reason": "one sentence"}}"""
```

During my run, the judge prompt was as follows:

```
[A] Judge prompt: Score this response against the reference on a 1-5 scale.
1 = completely wrong, 5 = matches reference perfectly.

Question: What is the refund policy for digital products?
Reference: Digital products can be refunded within 14 days if not downloaded.
Response: Digital products can be refunded within 14 days if not downloaded.
```

The judge compares the response to the reference and returns a JSON object. For this question, it came back with:

```
{ "score": 5, "reason": "The response matches the reference perfectly." }
```

Score of 5 out of 5. The model had the policy, read it, and gave it back correctly. That is the happy path.

Now the function does the same thing with Prompt B for the same question, but Prompt B has no policy field. The model only sees:

```
Answer this customer question.

Question: What is the refund policy for digital products?
```

Without the policy, the model has to guess from its training data. It gave a generic answer about refund policies that missed the 14-day window and the download condition. The judge scored it 2 out of 5.

Here are the outputs from my run:

```
=== Prompt B: Without policy context ===
[B] Query: What is the refund policy for digital products?
 Model answer: The refund policy for digital products can vary depending on the company or platform
from which you made the purchase. Generally, digital products are often non-refundable due to their
```

```
intangible nature and the ease of duplication. However, some companies may offer refunds under
specific circumstances, such as if the product is defective, not as described, or if there are
technical issues that prevent its use.

To get the most accurate information, it's best to check the specific refund policy on the website
where you purchased the digital product. Look for sections labeled "Refund Policy," "Returns," or
"Terms and Conditions." If the information is not clear, consider reaching out to the company's
customer service for clarification.
[B] Judge prompt: Score this response against the reference on a 1-5 scale.
1 = completely wrong, 5 = matches reference perfectly.

Question: What is the refund policy for digital products?
Reference: Digital products can be refunded within 14 days if not downloaded.
Response: The refund policy for digital products can vary depending on the company or platform from
which you made the purchase. Generally, digital products are often non-refundable due to their
intangible nature and the ease of duplication. However, some companies may offer refunds under
specific circumstances, such as if the product is defective, not as described, or if there are
technical issues that prevent its use.

To get the most accurate information, it's best to check the specific refund policy on the website
where you purchased the digital product. Look for sections labeled "Refund Policy," "Returns," or
"Terms and Conditions." If the information is not clear, consider reaching out to the company's
customer service for clarification.

Return ONLY a JSON object: {"score": N, "reason": "one sentence"}
```

The judge gave 2 out of 5 for this answer:

```
{
  "score": 2,
  "reason": "The response provides general information about refund policies for digital products but
does not accurately reflect the specific 14-day refund condition mentioned in the reference."
}
```

Prompt B is the control group; you cannot price what the context adds without seeing the run where it is absent. Scores dropped without it here, so retrieval is earning something. Flat scores would have sent me back to *Chapters 3* through *5* instead of the prompt.

That is one test case, two prompt variants, two model calls, two judge calls. The function loops through all five questions and collects every score into a dataframe.

Step 4: reading the regression report

Step 4 compares the two prompts question by question. The averages came out to 4.20 for Prompt A and 3.40 for Prompt B, so context wins overall. The

per-question table is where it gets interesting, though. Prompt A won four of the five questions, but one went the other direction. The physical return question scored 1 with the policy sitting right there in the prompt. The model refused to answer anyway, then scored 5 without context because its general knowledge happened to be correct. No average was ever going to surface that. Have a look here:

```
============================================================
PROMPT COMPARISON -- SIDE BY SIDE
============================================================
  A=5  B=2  (A wins)  What is the refund policy for digital products?
  A=1  B=5  (B wins)  Can I return a physical item after 30 days?
  A=5  B=3  (A wins)  What happens if my subscription renews and I want
  A=5  B=3  (A wins)  I bought a gift card and the recipient lost it. Ca
  A=5  B=4  (A wins)  My order arrived damaged. Who pays for return ship

  Average:  A=4.20  B=3.40
  Delta: +0.80 (positive = A is better)
```

The two failures map two different parts of this book. Prompt B scoring 2 out of 5 on the refund question is a missing context problem; that is what *Chapters 4* and *5* solve with RAG and retrieval. Prompt A scoring 1 out of 5 on the physical return question is a behavioral problem; the model had the policy right there and still refused to answer. That is what *Chapter 7* solves with fine-tuning. Evaluation does not just tell you something is wrong; it tells you which tool to reach for.

A big improvement from adding context was the gift card edge case, where the no-context model invented a policy that does not exist. Exactly the kind of fabrication you want to catch before deployment. Change the prompt, run the suite, read the breakdown. About 30 seconds for five questions, under five minutes for a hundred.

../../EPUB/TIP

One note before you compare your numbers to mine. The notebook picks a low-cost model your key can use, so your model may answer differently from mine; the refusal I just described might not happen on yours at all. That is fine. The point is the check, not my exact scores.

This was the simplest prompt, but it shows how the LLM-as-a-judge works. You can keep tweaking the prompt and the aggregation until you achieve better accuracy. At Clearwater Analytics, we have been using Dan Siddall's framework for offline and online evaluation. The trick has always been creating the golden set, as Dan explained in his other article [3].

Test sets go stale. Users in 2024 asked different questions than users in 2026, and if you only test against the original 200 questions, you are blind to the distribution shift. The model can memorize your test set too, especially after a few rounds of fine-tuning against the same data. One team reported 95 percent on their eval set; production was closer to 70. Too small, too old.

So far, the checks graded the prompt and the model. Retrieval needs its own check, because the pipeline can fail even when the model is fine, and the next part shows how I grade the retrieval side.

Evaluating the RAG pipeline

This section is optional. It is the only part of the chapter that needs the ChromaDB store from *Chapter 3*, and it is here to show how I run retrieval evaluation in practice on my own systems. If you only want the data-evaluation methodology, you can skip this section entirely.

The prompt comparison in the LLM-as-a-judge walkthrough only tested generation; the model had the answer, or it did not. RAG evaluation is two problems, not one. Retrieval can miss the right chunk entirely, and no prompt fix will save you because the model never saw the information. Or retrieval nails it and the model fabricates anyway, ignoring the context sitting right there in the prompt. I have seen both in production more times than I want to admit. That comparison tested one half. Now we test both halves together on the same ChromaDB collection from *Chapter 3*.

Step 1: loading the ChromaDB from Chapter 3

The *Chapter 3* notebook created the shared ChromaDB store at `data/chroma_db/`, and we load that same collection here: four company documents. If you skipped *Chapter 3*, the cell does not stop you; it builds a small fallback collection from the same four company documents, so the evaluation still runs.

```
import chromadb

SHARED_DB = repo_root / 'data' / 'chroma_db'
rag_client = chromadb.PersistentClient(path=str(SHARED_DB))

book_collection = rag_client.get_or_create_collection(
    name="book_collection")

print(f"Loaded collection: {book_collection.name} "
      f"({book_collection.count()} documents)")
```

The output lists every document the retriever can search over, and you should recognize them: the AI policy, the Q2 financial results, the Aurora Project announcement, and the expense report instructions.

Your count will probably not be exactly 24. The four company documents are the ones that matter here; the rest are stray test entries my earlier *Chapter 3* runs left behind, a manual test sentence, and a couple of demo lines. Shared vector stores accumulate junk unless somebody clears them, which is a small preview of the data hygiene problems *Chapter 10* deals with.

```
Loaded collection: book_collection (24 documents)
  [manual_1] This is a quick test sentence to index....
  [demo_ce405c63b181a6ab] First line...
  [demo_b456a020d1eecda5] Second line...
  [doc_e703d12f46044bda] The company's new AI policy, effective June 1st, requires all employees to
compl...
  [doc_3881aea93a79aa81] Our Q2 financial results show a 15% increase in revenue, driven by strong
sales ...
  [doc_8a91f8af1bf32f5e] The Aurora Project, our next-generation AI platform, is scheduled for a beta
rel...
  [doc_7da6c838a91b28c5] All travel and expense reports must be submitted through the new online
portal b...
  [id_0] The company's new AI policy, effective June 1st, requires all employees to compl...
  [id_1] Our Q2 financial results show a 15% increase in revenue, driven by strong sales ...
  [id_2] The Aurora Project, our next-generation AI platform, is scheduled for a beta rel...
```

```
[id_3] All travel and expense reports must be submitted through the new online portal b...
... (13 more entries)
```

Look at the IDs and you will see each company document twice. In *Chapter 3*, we indexed them once through `vector_indexer`, which produced the `doc_` hash IDs, and once by hand with the fixed IDs `id_0` to `id_3`. The golden dataset in the next step checks for the fixed IDs. Both copies have the same text, so they come back as a pair, and the top-two check here behaves like a top-one check: the right document has to rank first, and its copy takes the second slot.

Four documents sound too small to mean anything. That is the point of it. With four documents, you can trace every retrieval result by hand: retrieval returns `id_1` for a finance question, you open `id_1` and check whether the answer is in there.

Step 2: building the RAG golden dataset

Same idea as the offline golden dataset, with one addition. Each test case carries the document ID we expect the retriever to return. Four questions map to real documents, and the fifth asks about a remote work policy the collection has never contained, on purpose.

```
rag_golden = [
    {
        "query": "What is the new AI policy?",
        "expected_doc_id": "id_0",
        "reference_answer": ("The new AI policy requires "
                             "all employees to complete ..."),
        "category": "policy",
    },
    # ... 3 more with expected doc IDs
    {
        "query": "What is our remote work policy?",
        "expected_doc_id": None,   # intentionally out of scope
        "reference_answer": ("No information available in the "
                             "knowledge base."),
        "category": "out_of_scope",
    },
]
```

The `expected_doc_id` field lets retrieval get graded on its own. The retriever hands back `id_2` where we expected `id_0`; retrieval failed, and the fluency of whatever got generated afterward is beside the point.

Step 3: running the full pipeline and scoring

For each test case, the evaluation function makes four moves. It queries ChromaDB for the top two chunks and checks whether the expected document made the cut; that is the retrieval hit. Then it stuffs whatever was retrieved into a prompt, generates an answer, and sends the result to the judge, which scores faithfulness, whether the model stayed inside the retrieved context, and correctness, whether the answer matches the known truth. Have a look here:

```
# Loop body of evaluate_rag(collection, golden_data); item is one test case.
# The notebook calls it as evaluate_rag(book_collection, rag_golden).
# 1. Retrieve
retrieval = collection.query(query_texts=[item["query"]], n_results=2)
retrieved_ids = retrieval["ids"][0]
expected = item["expected_doc_id"]
retrieval_hit = expected in retrieved_ids if expected else True

# 2. Generate with retrieved context
context = "\n---\n".join(retrieval["documents"][0])
prompt = f"Use ONLY this context to answer.\n<context>{context}</context>\n<question>{item['query']}
</question>"
answer = client.chat.completions.create(model=BASE_MODEL, messages=[...])

# 3. Judge on two dimensions
judge_prompt = f"""Score this RAG response on two dimensions (1-5 each).
1. Faithfulness: answer uses ONLY retrieved context (5=grounded, 1=hallucinated)
2. Correctness: answer matches the reference (5=perfect, 1=wrong)
..."""
```

Before looking at the output, notice that faithfulness and accuracy are closely related in this example because both are evaluated against the same source document. In larger production systems, they often become separate checks. Faithfulness asks whether the answer stayed within the provided evidence, while accuracy asks whether the answer itself is correct. Most failures affect both scores, but not always.

Here is what the output looks like for each question. The `[HIT]` or `[MISS]` tag tells you whether retrieval found the right document. The out-of-scope question has no document to find, so the code counts it as a hit by construction, and what we actually grade there is the answer. The Faith and Corr scores come from the judge:

```
=== Running RAG Pipeline Evaluation ===

  [HIT]  What is the new AI policy?                        | Faith=5 Corr=5 | The new AI policy
requires all employees to complete a man...
  [HIT]  What were the Q2 financial results?               | Faith=5 Corr=5 | Q2 results show a 15%
increase in revenue, driven by str...
  [HIT]  When is the Aurora Project beta release?          | Faith=5 Corr=5 | The Aurora Project is
scheduled for a beta release in th...
  [HIT]  How do I submit expense reports?                  | Faith=5 Corr=5 | Expense reports must
be submitted through the new online ...
  [HIT]  What is our remote work policy?                   | Faith=5 Corr=4 | I cannot find this
information in the provided context...
```

The first four questions were all hit. The retriever found the right document and the model generated an accurate answer, 5/5 on both scores. The fifth question is the interesting one; there is no remote work policy in the collection, and the model correctly said, "I cannot find this information." The judge scored faithfulness at 5 because the model did not fabricate, and correctness at 4 because the phrasing was slightly different from the reference, but the intent matched.

Running the pipeline is only half the story. The real value comes from interpreting the scores, which is what the next step summarizes.

Step 4: reading the summary

The summary rolls up the per-question scores into three numbers:

```
============================================================
HAND-BUILT RAG EVALUATION SUMMARY
============================================================
  Retrieval accuracy:  100% (5/5)
  Avg faithfulness:    5.00 / 5
  Avg correctness:     4.80 / 5
```

Retrieval accuracy is the percentage of questions where the expected document appeared in the top two results. This shows that we built a good retriever in *Chapter 4*. Faithfulness measures whether the model stuck to what it was given. Correctness measures whether the final answer matches the known truth.

When retrieval is correct but correctness drops, the generation prompt needs work. When retrieval misses, no prompt fix will help; the data strategy needs work. That is the whole point of evaluating both stages separately.

Step 5: running RAGAS automated metrics

After the hand-built evaluation, the same golden dataset goes through RAGAS, an open-source framework that automates this kind of checking at scale. The standardized metrics it ships include faithfulness, context precision, context recall, and answer relevancy. We run the first three. Answer relevancy got dropped from this run; its embedding dependency clashed with the notebook's package versions, and the three remaining metrics already tell the retrieval and grounding story. The hand-built version walked you through what each metric measures line by line, so none of these should feel new. RAGAS just packages them so a **continuous integration** (**CI**) pipeline can run the checks on every change and track the trend over time.

```
from ragas import evaluate as ragas_evaluate
from ragas.metrics import faithfulness, context_precision, context_recall

# answer_relevancy skipped: needs embed_query, breaks on newer langchain-openai
safe_metrics = [faithfulness, context_precision, context_recall]
ragas_result = ragas_evaluate(dataset=dataset, metrics=safe_metrics)
```

Each metric prints on a 0 to 1 scale with a bar next to it:

```
Stubbed missing module for RAGAS compatibility: langchain_community.chat_models.vertexai
RAGAS loaded successfully
Prepared 4 samples for RAGAS

Evaluating: 100%|█████████| 12/12 [00:41<00:00,  3.45s/it]
=== RAGAS Evaluation Results ===
  faithfulness              1.000  ####################
  context_precision         1.000  ####################
  context_recall            1.000  ####################

Metric guide:
  faithfulness       -- 1.0 = answer uses only retrieved context
  context_precision  -- 1.0 = every retrieved chunk is relevant
  context_recall     -- 1.0 = all needed information was retrieved
```

All three came back at 1.0, the expected result on four documents; anything less means our setup has a bug. Production corpora in the thousands move these numbers, context recall first usually. When it slips from 1.0 to 0.85, the retriever lost part of an answer somewhere, and CI is where you want that news, not a production incident a month out.

The challenge with RAGAS is the dependencies [7]. It requires specific version matching between LangChain, Pydantic, and several Google packages, and those versions break every few months. I have hit this myself more than once; it is frustrating. The notebook's fix does not pin versions; it heals the break instead. When RAGAS asks for a LangChain module that no longer exists, the cell stubs the missing module and retries the import, printing which module it stubbed so you can see the fix work. A future release can still break in some new way the stub cannot cover, and when that happens the cell says so and falls back to the hand-built metrics instead of stopping the notebook. On two projects I ended up dropping RAGAS entirely and replacing it with the hand-built judges from the previous section. Five functions, no external dependencies, same signal.

Step 6: reading the combined report

The last cell puts both sets side by side, hand-built next to RAGAS. Numbers land close on four documents. The point is the workflow; on a production corpus, the framework runs these checks in minutes where the manual version would eat your afternoon.

```
============================================================
RAG PIPELINE EVALUATION -- COMBINED REPORT
============================================================

--- Hand-Built Metrics (our code) ---
  Retrieval accuracy:  100%
  Avg faithfulness:    5.00 / 5  (LLM-judge, 5-point scale)
  Avg correctness:     4.80 / 5  (LLM-judge, 5-point scale)

--- RAGAS Metrics (automated framework) ---
  faithfulness              1.000  (0-1 scale)
  context_precision         1.000  (0-1 scale)
  context_recall            1.000  (0-1 scale)

--- Per-Question Breakdown ---
  [PASS] What is the new AI policy?                    Faith=5 Corr=5
  [PASS] What were the Q2 financial results?           Faith=5 Corr=5
  [PASS] When is the Aurora Project beta release?      Faith=5 Corr=5
  [PASS] How do I submit expense reports?              Faith=5 Corr=5
  [PASS] What is our remote work policy?               Faith=5 Corr=4
```

So far, I have explained the evaluation concept and applied it to a small validation set I built by hand. If you followed the optional RAG walkthrough,

you saw the same idea applied to the pipeline. I kept that data simple on purpose to show the concept clearly. In production, we create synthetic datasets similar to the ones we built in *Chapter 6* and use them for training and evaluation. Before we trust them for either purpose, we have to validate the synthetic data itself.

Evaluating and using Chapter 6's synthetic data

In *Chapter 6*, we created the golden synthetic data and verified it with a human in the loop. Human review is especially important for a small golden set, but reviewing every generated example does not scale when the dataset grows or changes frequently. This section will automate the validation that *Chapter 6* did by hand, across three pillars: faithfulness, answerability, and diversity, plus one more check for the preference pairs at the end. A synthetic dataset is more than just a collection of generated questions and answers. Each QA pair has several properties that can fail independently. The answer may not be faithful to the source, the question may not be answerable from the source, or the dataset may contain too many similar questions. Instead of treating the dataset as one thing to validate, I evaluate each of these properties separately.

Faithfulness

One of the challenges I faced in evaluating LLM answers is that they are very fluent and confident, carrying facts that the source never had. It is always very convincing, and the LLMs are very good at making it convincing. I have been trapped by this so many times. The scars and trauma taught me to be very careful and validate the faithfulness of the answers carefully.

Faithfulness compares the answer to the source and checks whether every claim in the answer is supported by it. Sometimes the LLM adds correct details that only a human subject matter expert (SME) can safely accept. In this synthetic-data check, the provided source is the evidence available to the judge. Any additional claim that cannot be inferred from the source will be treated as unsupported, even if the claim happens to be factually correct outside that source. When I was debugging the failed cases in production, I found the LLM actually gave a very good answer, but it was not supported by the source.

I would like to start with a hands-on example to show how to validate the faithfulness of the answers. I will start by injecting some failures to explain how the test helps catch them.

In the Notebook, search for *"Part 3: Evaluating Synthetic Data Faithfulness"* to find the section where I will demonstrate this.

The first cell is the normal judge prompt that we explained before but focused on faithfulness. Make sure to run this cell first because the rest of the cells will use it:

```
def judge_qa_faithfulness(question, answer, source):
    prompt = f"""You are an evaluation judge. Score this QA pair on three dimensions.
Each score is 1-5 where 5 is best. Give a one-sentence reason for EACH dimension.

Source policy: {source}
Question: {question}
Answer: {answer}

Score these three dimensions:
1. Faithfulness: Is every claim in the answer supported by or reasonably inferable from the source?
(5 = fully grounded, 1 = mostly unsupported)
2. Relevance: Does the answer directly address the question? (5 = directly answers, 1 = off topic)
3. Correctness: Does the answer agree with the facts stated in the source? (5 = fully correct, 1 =
factually wrong)

Return ONLY valid JSON:
{{"faithfulness": N, "faithfulness_reason": "one sentence",
  "relevance": N, "relevance_reason": "one sentence",
  "correctness": N, "correctness_reason": "one sentence"}}"""
```

In this small example, faithfulness and correctness are closely related because both are evaluated against the same source. They are not the same, though. Faithfulness asks whether the answer stayed within the evidence provided, while correctness asks whether what the answer said agrees with the facts in that evidence. In a larger production evaluation, I usually compare correctness against a separate reference answer while checking faithfulness against the retrieved context, the same split you saw in the RAG section earlier.

The second cell is the one where we will introduce the error (fault injection):

```
verdict = judge_qa_faithfulness(vac["question"], corrupted_answer, source_text)
```

When you run it, you will get the following output:

```
STEP 2: corrupt ONE real dataset row, then judge it
============================================================
INPUT TO THE JUDGE
  Question:        How many days of paid vacation do employees receive per year?
  Source of truth: Employees get 15 days paid vacation/year. Request 2 weeks in advance. Up to 5 days
```

```
carry over.
  Original answer: Employees receive 15 days of paid vacation per year.
  Corrupted answer (what we score): Employees receive 25 days of paid vacation per year, and unused
days roll over indefinitely.

JUDGE VERDICT (a score and a reason for each dimension)
  Faithfulness = 1  ->  The answer includes unsupported claims that employees receive 25 days and
that unused days roll over indefinitely.
  Relevance    = 5  ->  The answer directly addresses the question about annual paid vacation days.
  Correctness  = 1  ->  The source states employees receive 15 days per year and may carry over up to
5 days, not 25 days with indefinite rollover.

Read the three reasons together: relevance stays high because the answer is
on topic and reads well, but faithfulness and correctness collapse because it
invents numbers the policy never had. The check catches the bad row.
```

As you can see, the judge correctly identified that the answer was not faithful to the source and that it was not correct according to the source. While this is a fault injection example, it demonstrates how the judge can catch errors in the data. We usually introduce faults in the data to test the data quality checks.

Now, let's try to get the correct answer. Search for *"STEP 3: score every real HR QA pair"* in the Notebook and run the cell. Of course, all of them will run successfully.

```
STEP 3: score every real HR QA pair against its source
============================================================

  [1/6] How many days of paid vacation do employees receive per year?
      Source: Employees get 15 days paid vacation/year. Request 2 weeks in advance. Up to 5 days
carry over.
      Answer: Employees receive 15 days of paid vacation per year.
      Faithfulness = 5  ->  The answer uses only the stated source information that employees receive
15 days of paid vacation per year.
      Relevance    = 5  ->  The answer directly addresses how many paid vacation days employees
receive per year.
      Correctness  = 5  ->  The answer is factually correct because the source states employees get
15 days paid vacation per year.

...

Average correctness:  5.00 / 5.00
Rows below 4 on faithfulness: 0 / 6

All six real pairs are faithful to their source. The dataset passes.
```

Now, let's try some real-life errors like a sloppy prompt, long reference documents with ambiguous policies, and smaller models. In the Notebook, search for `STEP 4: Compare generators` and run the cell:

```
THE SLOPPY GENERATION PROMPT (same for every model; remote question shown)
--------------------------------------------------------------------------
hi can u help me anser some HR questons for our new employe handbook.
we got lots of polcies pasted below from diffrent places. jus write a nice
helpfull anser about the questoin, make it sound complete and profesional.
dont worry to much about which exact company it is, jus give the genral anser.

HANDBOOK:
### Acme Corp - Time Off
Acme employees receive 25 days of paid vacation per year, and unused days roll over indefinitely.

### Globex - PTO
Globex offers unlimited paid time off. No carryover limits apply.

### Initech - Leave
Initech grants 20 vacation days plus 12 sick days. Vacation carries over up to 10 days per year.

### POL-002 - Remote Work Policy
Work remotely up to 3 days/week with approval. Available during core hours 10 AM-3 PM.

### Umbrella - Remote
Umbrella staff may work remotely 5 days per week with no approval required.

### Hooli - Flexible Work
Hooli allows fully remote work indefinitely; core hours are not enforced.
...



==========================================================================
Comparing 5 generators on 2 questions...

  gpt-3.5-turbo                    vacation_faith=1  remote_faith=1  drifts=2
  gpt-3.5-turbo-0125               vacation_faith=1  remote_faith=1  drifts=2
  gpt-3.5-turbo-1106               vacation_faith=2  remote_faith=1  drifts=2
  gpt-3.5-turbo-16k                vacation_faith=1  remote_faith=1  drifts=2
  gpt-5.5                          vacation_faith=4  remote_faith=3  drifts=1  (control, strong)

==========================================================================
MODEL                              vacation  remote  drifts
--------------------------------------------------------------------------
gpt-3.5-turbo                             1       1       2
gpt-3.5-turbo-0125                        1       1       2
gpt-3.5-turbo-1106                        2       1       2
gpt-3.5-turbo-16k                         1       1       2
gpt-5.5                                   4       3       1  <- control
```

```
======================================================================
Faithfulness is 1-5 (5 = fully grounded). drifts = questions scoring below 4.
Smaller / older models tend to pull in the distractor companies' numbers and
drift; the strong control holds far better, though a sloppy prompt can still
nick it once. The same check catches it either way.
Generation is stochastic, so exact scores vary from run to run.
```

Here I have to answer a question that always comes up. The judge is also an LLM, so why do we trust it to catch the mistakes of another LLM? The reason is that judging is much easier than writing. The judge does not create the answer from nothing; it only reads the answer and the source together and checks if they match. This is a much smaller job than generating the answer, and this is why it can catch what the generator missed. However, the judge is still an LLM, and I do not trust it blindly. I always use a judge that is as strong as the model that created the data, or stronger. When I can, I use a judge from a different model family, so they do not make the same mistakes together. This is exactly why in Step 4 I used the strong model to judge the small older models. And I still read some of the verdicts myself from time to time, to make sure the judge stays honest.

There is another reason I do not let a model grade its own work. Models tend to like their own answers; they give their own output a higher score than they give to an answer of the same quality from another model. Dan Siddall measured this in his evaluation article and found that GPT scored its own work about 10 percent too high. This is called self-enhancement bias. Because of this, I do not use the same model as the writer and the judge at the same time. If GPT wrote the answer, I let a model from another company judge it, like Claude or Gemini. It does not remove the bias, but it stops a model from protecting its own answers.

I usually run those tests if I change config, prompts, or model versions to ensure the changes didn't introduce regressions before going to production. It caught so many issues that I now consider it a mandatory step. Sometimes I send the failed ones back to the SME, because a failed answer is not always a wrong answer. The answer can be correct even when the source does not say it clearly, and the check marks it as unfaithful anyway. The SME looks at these cases, confirms the ones that are actually right, and adds them to the good answers so the next run does not flag them again.

Faithfulness measures how close the answer is to the source. Answerability measures something different: whether the source even has the information needed to answer the question. In other words, is it a good question?

Notice that we're still validating the synthetic dataset, not the model. We're asking whether each generated question-answer pair is good enough to become training or evaluation data.

Answerability

Faithfulness is not the only way a generated pair can go wrong. The question itself can be broken. If the synthetic-data generation process from *Chapter 6* produced a question that its own source cannot answer, the pair is useless no matter how clean the answer looks. This second check is answerability, and it looks at the question against the source before I judge the answer at all. A pair can be perfectly faithful and still fail it.

Let me use the vacation policy from *Chapter 6* again. It says, "employees get 15 days a year, they should ask two weeks in advance, and up to 5 days carry over." A question like "how many vacation days do employees get" is answerable; the number is right there in the policy. A question about paternity leave is not, because the policy never mentions paternity leave anywhere. That second question is already a bad pair before we even evaluate the answer.

This matters because of how the data was made. In *Chapter 6*, the model generated the questions from each policy. Most of the time it stays on topic, but sometimes it drifts and writes a question the policy cannot answer. If that pair goes into the training data, it teaches the model to guess when it has nothing to go on, or it forces a made-up answer to fill the gap. Either way, the model learns to answer when it should have said it does not know.

I check this the same way I checked faithfulness. In the Notebook, search for *"Part 3b: Evaluating Answerability"*. The first step proves the check has teeth; it asks the paternity leave question against the vacation policy, and the judge scores it low because the source does not cover it. The second step runs the check over all six real HR questions, and they all pass because *Chapter 6* built each question from a policy that actually holds the answer.

```
STEP 1: ask a question the source cannot answer
============================================================
Source of truth: Employees get 15 days paid vacation/year. Request 2 weeks in advance. Up to 5 days
carry over.
Question (not covered by this policy): How many days of paid paternity leave do new fathers receive?

JUDGE VERDICT (a score and a reason)
Answerable = 1 -> The source policy only describes paid vacation and carryover rules and does not
mention paternity leave or benefits for new fathers.

The policy says nothing about paternity leave, so the question cannot be
answered from the source. A low score here means the check works.
```

For those questions, I set the reference answer to "Cannot answer, not covered by this policy," or something similar, so I have something to compare against. In this case, if the LLM fabricated a policy, it could be caught, but if it correctly refuses to answer, it matches the expected reference answer and passes the check. This is a good way to catch fabrication with questions that do not have answers in the source.

Unlike faithfulness and answerability, diversity cannot be evaluated one QA pair at a time because it is a property of the dataset as a whole. A dataset can be full of pairs that are each faithful and each answerable and still be weak, because the questions all ask the same thing in slightly different words. If every question is a small rewording of how many vacation days do I get, the model only really learns one question. Diversity is how I check that the set actually covers different things.

Diversity

Diversity measures how different the questions in the set are from each other [8]. The simplest version does not need an LLM at all; it is just counting. I break each question into words and count how many are unique across the whole set, and I also look at how many questions start with the same first few words. When a set reuses the same words and the same openers over and over, these numbers drop, so this is a fast and cheap way to catch a set that is repeating itself.

Here is where counting falls apart. One question asks how many vacation days I get. Another asks what the annual paid time off allowance is. Same question, barely a shared word, and the counts call them diverse. They are not.

So counting is only my first pass. It is instant; it catches the lazy repeats, and it is where I always start. However, the paraphrases matter, and the coverage has to be right. Therefore, I switch to embedding similarity or a quick LLM judge that compares what the questions mean instead of the words they use. It is always a tradeoff between cost and benefit.

In the Notebook, search for *"Part 4: Evaluating QA Diversity,"* then run the cell after it:

```
=== HR QA Diversity ===
  total_questions: 6
  unique_unigrams: 50
  unigram_ttr: 0.746
  unique_bigrams: 55
  bigram_ttr: 0.902
  starter_diversity: 0.667
  unique_starters: 4

=== Structured QA Diversity ===
  total_questions: 12
  unique_unigrams: 28
  unigram_ttr: 0.333
  unique_bigrams: 36
  bigram_ttr: 0.5
  starter_diversity: 0.417
  unique_starters: 5

Question starters are reasonably diverse. Good.
Vocabulary diversity is acceptable.
```

The output has two blocks, one for each set. Let me read the HR set first. It has six questions. The unigram type token ratio is 0.746, which means about three out of every four words in the set are different words. The bigram ratio is 0.902, so almost every two-word pair is different too. The starter diversity is 0.667, which means the six questions use four distinct openers between them, so a couple of them start the same way. This looks like a healthy lexical distribution. The questions are worded in different ways, and they do not repeat each other.

Now the structured set. It has twelve questions, and the numbers are much lower. The unigram ratio drops to 0.333, so only one word in three is unique and the rest repeat. The starter diversity is 0.417, so only five of the twelve questions start in a different way, and the rest start the same.

But do not panic when you see a low number here. The structured questions come from a template. Many of them ask for an employee's job title or tenure, or for a quarter's revenue or profit margin, so the words repeat. For this kind of data, that is normal; it comes up on every structured set I check. The count shows me lexical repetition, but it cannot tell whether the questions are semantically different or whether important topics are missing.

There is one thing to watch in the output. The two lines at the bottom (the one that says the starters are diverse and the one that says the vocabulary is acceptable) only look at the HR set.look at the HR set. The code checks the HR numbers and prints the verdict for the HR set only. The structured set is not checked there, and if I ran the same rule on it, the starter diversity of 0.417 would trip the warning. So, I always review the raw numbers for both sets instead of relying only on the printed verdicts.

Faithfulness and answerability graded individual QA pairs, while diversity evaluated the dataset as a whole. *Chapter 6* produced one more dataset type, and evaluating it brings back a bias I mentioned at the beginning of the chapter.

Preference pairs and the position bias check

A **preference pair** is a chosen response and a rejected response for the same instruction. Grading them with a judge sounds trivial: show both responses and ask which one is better. There is a trap in it, though. Judges tend to favor whichever response they read first, a well-known phenomenon called position bias. This is why every preference dataset should be randomized before evaluation. So the notebook randomizes which response goes first on every call and tracks two numbers instead of one: how often the judge picked the genuinely better response, and how often it picked whatever came first. In the Notebook, search for *Part 5: Evaluating Preference Pairs* and run the cell.

```
=== Preference Evaluation Report ===
Judge accuracy: 100.0% (4/4 correct)
Position bias:  25.0% picked Response A (first position)
  (50% = no bias, >70% = strong position bias)
```

The judge picked the preferred response on all four pairs, and the first position only a quarter of the time; no lean. Do not skip the randomization on the strength of a clean result like this one. A judge can score perfectly on accuracy

while quietly favoring whatever appears first, and if the preferred response happens to be listed first in your dataset, the accuracy number will look perfect for exactly the wrong reason. Randomize the order, or you cannot tell the difference.

The synthetic data from *Chapter 6* was one of the two data improvements I need to check. The other one is the fine-tuned model from *Chapter 7*. It lives in the same offline corner, fixed data scored by a judge, run before anything ships. So, I check it the same way, and this time the question is whether the fine-tuning was needed at all.

Evaluating the fine-tuned model from Chapter 7

Chapter 7 produced a fine-tuned model. Before I trust it, I check three things: the training data it learned from, whether the base model already meets the contract without any fine-tuning, and the quality of the content the model actually writes. I go through each one in turn.

In the Notebook, search for *"Part 6: Loading and Inspecting Chapter 7 SFT*."

Chapter 7 fine-tuned a support assistant on a strict contract: every answer has to carry three sections—a summary, a next steps list, and a risks note. Before judging any model against that contract, the data itself has to be consistent, so the first cell just inspects the training set.

```
SFT training examples:   10
SFT validation examples: 3
Unique system prompts: 1
Output structure consistency: 100%   (Summary, Next Steps, Risks on all 10)
```

Ten training examples, three held back for validation, and every one of them shares a single system prompt with all three sections present. A hundred percent structural consistency. This matters more than it looks; if half the training examples skipped the risks section, the model would learn that skipping it is sometimes fine, and the contract would leak from the first epoch.

Search for *"Part 7: Evaluating Base Model vs the Behavioral Contract"* in the notebook.

Here is the question most fine-tuning projects skip. Not whether the fine-tuned model improves, but whether you need to fine-tune at all. The honest way to answer it is to score the base model against the same contract first, and only spend on training if there is a real gap to close.

```
=== Base Model Evaluation ===
Average contract compliance: 100%
  Summary present:    100%
  Next Steps present: 100%
  Risks present:      100%
```

The base model produced all three sections on every validation example. Contract compliance: one hundred percent. If format were the whole contract,

you would stop here, save the GPU budget, and ship the base model with a good system prompt.

Format is not the whole contract, though. Checking that the three headings exist says nothing about whether what sits under them is any good. So, the last judge scores the content itself on four attributes, borrowed from the evaluation framework Dan Siddall built at Clearwater, with a written justification required for every score (search for *"Part 8: LLM-as-Judge Scoring"* in the notebook).

../../EPUB/NOTE

One naming note before you read the scores. Dan's rubric calls its first attribute accuracy, and I kept his name when reporting his framework. It is the same check I called correctness earlier in the chapter: does the answer agree with the expected facts? Current evaluation frameworks mostly use correctness for that answer-level judgment; RAGAS names its metrics answer correctness and factual correctness, while accuracy in classic ML usage means an aggregate rate, the share of predictions a model got right. So read accuracy in Dan's tables as correctness, and read accuracy everywhere else in this chapter, retrieval accuracy, judge accuracy, as the aggregate rate.

```
=== Multi-Attribute Evaluation Report ===
  accuracy    : 3.00 / 5.00
  relevance   : 4.00 / 5.00
  coherence   : 3.00 / 5.00
  reasoning   : 2.33 / 5.00
  overall     : 3.00 / 5.00
```

Now the picture changes. The same answers that scored a perfect hundred on structure came back at 3 out of 5 on content, with reasoning weakest of all at 2.33. The base model has learned the shape of a good support answer without the substance: the right headings over thin reasoning. That gap, full marks on format and a mediocre three on content, is the exact space fine-tuning from *Chapter 7* is meant to fill.

The delta cell closes the loop (search for "Part 8b: The Fine-Tuning Delta" in the notebook). It looks for the fine-tuned model that the *Chapter 7* notebook saved to disk, loads it with Transformers, and reruns the same two evaluations on it: the contract check and the multi-attribute judge. A fair delta needs the same model on both sides, so the cell loads the untrained base

model too and scores both, rather than borrowing the hosted model's numbers from the cells above. Base score, fine-tuned score, subtract. That delta is your fine-tuning return, the one number worth taking to whoever is paying for the training run.

On the machine I used while writing this chapter, the saved model folder was not there yet, and the cell said so:

```
No local fine-tuned model available -- skipping the delta comparison.

Run the Chapter 7 notebook first and re-run this cell.
Chapter 7 writes the local SFT model to:
  chapter_07/datasets/sft_model
This cell intentionally does not list remote jobs or pull a hosted model.
```

That skip message has a story behind it, and it is the technology-decay lesson of this book playing out in real time. The first version of this notebook pulled the fine-tuned model straight from OpenAI's fine-tuning API. Then OpenAI announced in May 2026 that it is retiring self-serve fine-tuning and pointing developers at prompting, retrieval, and orchestration instead, so we moved the training to a small open-source model that runs locally, and this chapter now loads it from disk. If anything, the wind-down strengthens the lesson of this section. Measure whether the base model already meets the contract before you reach for training, because the training option itself can disappear underneath you. The evaluation method survives either way. Point the same two judges at any model that serves, hosted, or local, and the delta math does not change.

Where do you go when the hosted option closes? I have lived this move already. Back in 2024, we fine-tuned a lot of models at Clearwater Analytics with a company called Predibase. We used them so much that fine-tuning open-source models like Llama became our normal way of building small task-specific models, and I wrote about that work at the time [9] https://www.linkedin.com/pulse/revolutionizing-ai-predibase-future-serverless-llms-rany-l7nnc. Predibase also came up in the Galileo webinar where Darrel Cherry talked about how we are building agentic systems with small language models at Clearwater Analytics [10] https://www.youtube.com/watch?v=6utcRSblxFI. Predibase became famous for LoRAX (LoRA eXchange). We used it extensively because it could dynamically

load lightweight LoRA adapters on top of one shared base model, instead of keeping separate copies of the model running. Predibase's published claim for LoRAX was serving hundreds of fine-tuned models for the cost of one GPU [11], and that math is what made small task-specific models practical for us. I wrote about LoRAX in [12] https://levelup.gitconnected.com/fine-tune-mistral-7b-instruct-on-predibase-with-your-own-data-and-lorax-b12b89c64874?sharedUserId=ranyel. That production work is also public: Darrel Cherry, Dan Siddall, and I co-authored a post with Amazon on the AWS Machine Learning blog about how Clearwater Analytics runs customized LLMs on SageMaker [13] https://aws.amazon.com/blogs/machine-learning/how-clearwater-analytics-is-revolutionizing-investment-management-with-generative-ai-and-amazon-sagemaker-jumpstart/. Rubrik acquired Predibase in mid-2025 [14], the same consolidation story as Galileo and Cisco, so check where the product stands when you read this. What does not change is the property that made us pick it in the first place: the models are open source, so your data and your fine-tuned weights stay with you, and no vendor decision can wind that down. Predibase is not the only path; other providers and the local LoRA route from *Chapter 7* cover the same ground, but it is the one I have run in production myself. Fine-tuning will continue to evolve and change. I have been writing about it for a while, and *Chapter 7* covers the foundation you can build on, which you can find in my article [15].

If you did run *Chapter 7* first, the cell prints the comparison instead of the skip message, and the numbers deserve a warning before you read them. Our technical reviewer ran this cell with the *Chapter 7* model in place and got a result that looks broken at first: contract compliance fell from 100 percent on the untrained base to 33 percent on the fine-tuned copy, and all five judge scores came back at exactly 1.00 for both models, zero delta anywhere. Nothing in the evaluation broke. Both models are producing text below the judge's floor, and below the floor the judge cannot rank one weak answer against another, so the flat 1.00 rows mean both models are too weak for the judge to separate. The compliance drop has its own explanation: the keyword check rewards echo. An untrained model that parrots the prompt picks up the section names for free, and ten training examples were enough to stop the parroting but nowhere near enough to teach the real format. A flat or negative delta at this scale tells you the training run is too small to measure, not that

fine-tuning hurt. *Chapter 7* keeps the example tiny on purpose so it trains on a laptop. Rerun the delta with hundreds of examples on a base model big enough to shift, and the number starts meaning something. The cell prints this reading guide on its own whenever it sees the pattern in a live run.

The report cell rolls everything into one verdict (search for *"Part 9: The Complete Evaluation Report"* in the notebook).

```
============================================================
VERDICT
============================================================
Base model partially meets the contract.
Fine-tuning should improve consistency on the gaps.

Part 8b above runs the same evaluation automatically when the Chapter 7
fine-tuned model exists on disk. The delta there is your fine-tuning ROI.
```

The base model partially meets the contract, so fine-tuning is justified, but only on the gaps the content scores exposed, not the format it already handles.

Every check up to here, the RAG pipeline, the synthetic data, the fine-tuning contract, lived in the same corner, fixed datasets scored by machine before anything deploys. This is called the offline evaluation. That corner is where you build. It is not where you learn what real users actually do with the thing once it is live, and that is the next section.

Evaluating live production data

With online evaluation, the test set is whatever production sent you today. The signals are thumbs ratings, session abandonment, escalations to a human, and they arrive without reference answers; you read trends instead of scores. Users hit failures before you do; that is the cost of evaluating on live traffic.

Latency belongs on the production dashboard too; a slow correct answer is sometimes worse than a fast wrong one. The catch with all the online signals is statistical significance. You need thousands of samples before any of them mean something, which in practice is days or weeks of waiting after every change. Offline gates the CI/CD (continuous integration/continuous delivery) pipeline so regressions never ship. Online tells you what to improve next.

For the online side, I do not build the monitoring myself. Building an evaluation platform from scratch is a lot of work, and there are good tools that already do this well. So, in production we usually use a ready-made platform like Galileo, instead of writing our own.

Applying a production evaluation platform

Rather than survey thirty platforms that will look different by the time you read this, this section goes deep on one, the platform I have run in production for years, as a worked example of what a leader in this space looks like. The framework in "How to evaluate evaluation platforms" later in this chapter is how you evaluate everything else, including whatever replaces the names you know today.

The Galileo decision came out of a long bake-off. We lined up everything on the market back when we first needed an evaluation platform and scored it all, and nobody else was close at the time. What won it was hallucination detection that actually worked and latency that held up under production load. Years later, they have not slowed down, which is rare; most vendors lead for a quarter and then coast.

Luna is still the edge. Small evaluation models that score outputs in real time without a frontier-model bill attached are a big part of why Cisco is buying Galileo and not one of the others.

I partnered with Galileo at Clearwater Analytics starting in 2024, one of their early enterprise customers. We used Galileo Observe to monitor our LLM applications in production, and it cut our response time to issues from days to minutes. Our Distinguished Engineer, Darrel Cherry, went on record in Galileo's official Protect launch, saying it let us provide guardrails around our AI products and bring them to customers faster. We gave them feedback, they shipped fixes fast, and the product got better because of the back and forth [16].

Here is Galileo scored against the same dimensions I would apply to any evaluation platform on a shortlist, from core capabilities to vendor lock-in and the sweet spot where it earns its keep. Read it as a worked example of the scoring grid, not a buy recommendation:

Dimension	Details
Core capabilities	Hallucination Index (standardized hallucination measurement); Luna-2 runtime guardrails (< 200ms latency), plus Galileo Protect for real-time PII (personally identifiable information) and hallucination interception [17]; 20+ LLM-based evaluation metrics; Agentic evaluations for multi-step workflows; Evaluate, Observe, and Protect product suite; Multi-method hallucination detection (embedding similarity, Chain-of-Thought via ChainPoll [18], G-Eval)
Integration and developer experience (DX)	Integrates with OpenAI, Anthropic, Azure, Bedrock, custom models; Python SDK; Good documentation; Setup: 1-2 hours
Production readiness	Sub-200ms latency (validated in testing); Cloud SaaS and on-premise deployment; Fortune 100 deployments; Insights Engine (automated root cause analysis)
Cost	The one column I leave unscored in print: enterprise pricing varies by volume, and the Cisco acquisition makes any printed number stale, so score this column fresh at decision time
Differentiation	Purpose-built hallucination detection (Luna-2, Hallucination Index); multiple detection methods (no single-approach); automated root cause analysis (40 hours to 5 hours/week); early enterprise customers (Magid, Clearwater Analytics)
Vendor lock-in	Can export data; provider-agnostic; proprietary evaluation algorithms
Sweet spot	Teams where hallucination is the critical concern: medical, legal, financial, at enterprise scale

Table 9.1: Galileo scored against the platform-selection dimensions.

I wrote about Luna in detail back in 2024 [19] when it first came out. What impressed me then was that it is built on DeBERTa-large with disentangled attention, fine-tuned specifically for RAG inputs. It handles long-context retrieval well and generalizes across domains without retraining, which is

why Galileo was able to evolve it into Luna-2 for production guardrails. I also wrote about ChainPoll [18], which breaks a model's response into individual facts, verifies each one through chain-of-thought prompting, and flags the specific claim that is wrong. I also spent time hands-on with Galileo's Correctness Metric and published a walkthrough with screenshots from the Observe dashboard [20]. The Correctness Metric uses ChainPoll under the hood, breaks a response into individual claims, and scores each one. I ran it on real production queries and watched it catch a math error where the model used 31 days instead of 91 in a compound interest calculation, scored it 33 percent correct, and pointed to the exact claim that failed. That level of per-claim traceability is what sold me on Galileo for hallucination-critical use cases. I also published a hands-on coding tutorial on Galileo Protect's real-time firewall with LangChain integration [17]. For more on how we think about agentic systems built on small language models, you can watch the webinar we did with Galileo: "How Clearwater Analytics Creates Agentic Systems with SLMs" [10].

Cisco announced in April 2026 that it was acquiring Galileo [21], with Luna-2 named as a centerpiece of the deal, the plan being to extend Splunk Observability Cloud's AI agent monitoring. The technology case I made above does not change.

What changes is the framework math. Integration roadmap and lock-in deserve a fresh look once the Splunk integration ships, and so does standalone pricing.

One platform as a worked example does not mean one platform exists. Confident AI, Maxim AI, Braintrust [24], LangSmith, Langfuse [23], and Arize Phoenix compete in the same space; RAGAS, DeepEval [22], and TruLens cover the open-source side, and new entrants show up most months.

Whatever platform you land on, it still needs a dataset to score against, and that dataset is the one piece no vendor builds for you.

Building task-specific evaluation datasets

Five hundred high-quality examples that match your actual production queries will tell you more than ten thousand generic benchmark questions. The best process I know for building these sets is the one Dan Siddall ran at

Clearwater Analytics; I watched it from the inside, and the shape barely changes between domains. Start with real data. Production logs if you have them, support tickets if you are building a support bot. Real user questions, the kind nobody dreams up in a meeting room.

Understanding why task-specific evaluation matters

Generic benchmarks test general capabilities. Your production task has specific requirements:

- Customer support bot must handle product-specific questions, not general knowledge
- Medical diagnosis assistant must cite specific clinical guidelines, not generic health advice
- Legal research tool must reference actual case law, not hypothetical legal scenarios
- Content moderation system must catch domain-specific toxicity, not just profanity

That gap showed up for real on one of my projects. Two models with almost identical MMLU (Massive Multitask Language Understanding) scores landed 35 points apart on the task-specific evaluation set we built for the system. The one with the lower benchmark number kept getting filing dates and regulatory numbers right, while the higher scorer invented plausible details that did not exist. Three or four enterprise systems later, same story every time, I stopped expecting the leaderboard to predict any of it.

That same benchmark-to-production gap is the whole reason task-specific evaluation sets exist. Without one, you are guessing which model to deploy from numbers that never once looked at your users.

Sourcing data for evaluation sets

The best source is production logs if you have a system already running. Sample real user queries, and make sure to include failures and edge cases, not just the successes. Stratify by query type, user segment, and time period so your evaluation set is not all one flavor of question. Anonymize PII before using anything for evaluation, obviously.

At Clearwater Analytics, Dan Siddall developed a process for sourcing evaluation questions. The one thing I took from it was that no single source works by itself. He pulled from help desk tickets, had experts write the tricky edge cases, generated a batch with Claude and threw out most of them, and scraped format ideas from SQuAD (the Stanford Question Answering Dataset). Hybrid. Only way to get enough coverage.

No production logs yet? Expert annotation is the next best option. Domain experts write realistic queries with reference answers, maybe 10 to 20 examples per hour, and they know which edge cases matter because they have lived them. As of 2026, this runs $50 to $200 per hour depending on the domain, and for medical, legal, and financial applications it is worth every dollar.

Synthetic generation with human validation scales well once you have an initial set. Generate diverse queries with GPT or Claude at about $0.10 per example, then have human experts validate and edit them for realism at $1 to $2 per example. The gotcha is that synthetic queries often miss how real users actually talk, which is the whole reason the human editing step exists.

Customer support tickets work too if the product has them: real questions from real users with full conversation context, messy typos and all. Crowdsourced annotation at $0.50 to $2 per example covers bulk labeling when the task does not need deep expertise, with clear instructions, and at least three workers per example or the quality falls apart.

Defining the evaluation dataset schema

Settle on a standard schema before collecting anything. Once you have a format that RAGAS, DeepEval, and TruLens can all read, you do not have to reformat every time you switch evaluation tools.

The *Chapter 6* HR dataset already has this structure. Each row carries a `source_policy` field tracing back to the original document. When the notebook ran the faithfulness judge, that field was the ground truth. Without it the judge has nothing to compare against and faithfulness is unjudgeable. One column makes or breaks the evaluation.

For a simple question-and-answer set, the schema can stay minimal, just the query and a reference answer:

```
{
  "query": "What is the refund policy for digital products?",
  "reference_answer": "Digital products can be refunded within 14 days if not downloaded.",
  "context": "https://example.com/refund-policy",
  "expected_score": 1.0
}
```

For evaluating a full RAG pipeline, the schema grows to add the retrieved contexts and a ground-truth answer:

```
{
  "query": "What is the refund policy for digital products?",
  "retrieved_contexts": [
    "Digital products: 14-day refund if not downloaded",
    "Physical products: 30-day return policy"
  ],
  "reference_answer": "Digital products can be refunded within 14 days if not downloaded.",
  "model_output": "You can get a refund within 14 days if you haven't downloaded it.",
  "metadata": {
    "query_type": "policy",
    "difficulty": "easy",
    "domain": "customer_support"
  },
  "ground_truth_scores": {
    "faithfulness": 1.0,
    "answer_relevance": 1.0,
    "context_precision": 1.0
  }
}
```

The extra metadata fields are worth the effort because they let you slice results in ways a flat dataset cannot:

- Slice performance by query type (policy vs technical vs billing)
- Track difficulty distribution (don't evaluate only on easy examples)
- Filter by domain for targeted improvement

Creating the golden dataset

Building a golden dataset is a staged process as shown in *Figure 9.1*, and each stage has a rough duration you can plan around:

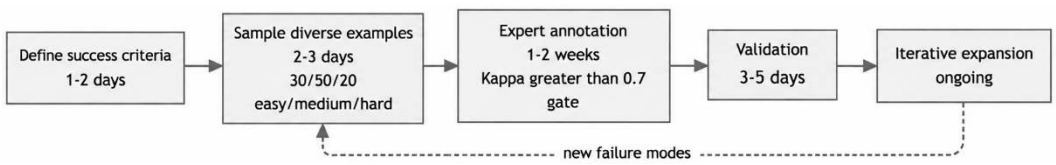


Figure 9.1: The golden dataset creation process

It starts with a day or two spent deciding what "good" even means, writing five to ten example outputs rated from perfect to unacceptable, and getting product, legal, and compliance to agree before anyone annotates a thing. Then two or three days sampling a diverse set, every query type, the ambiguous and multi-hop and conflicting-info edge cases, roughly 30 percent easy, 50 percent medium, 20 percent hard, aiming for an initial 50 to 100 examples. Expert annotation is the long pole, one to two weeks of domain experts writing reference answers with their reasoning, marking which documents should have been retrieved, scoring against the rubric, and confirming their agreement holds above 0.7. Validation takes another three to five days, running a baseline model across the set and reading the score distribution, because if everything comes back 1.0 or everything comes back 0.0, the set is too easy or too hard, and the examples where experts disagreed are where the rubric needs another pass. After that, it never really stops. You start at 50 to 100, watch what fails in production, add examples that cover those failures, and grow toward 500 to 1000, refreshing quarterly as the product moves.

Prioritizing quality over quantity

Fifty high-quality examples will teach you more than five hundred mediocre ones, and the contrast tends to show up in concrete ways:

High-quality example	Mediocre red flag
Representative of real user queries (actual language patterns)	Synthetic query that no real user would ask
Unambiguous correct answer (experts agree 90%+)	Multiple correct answers (ambiguous)
Clear rubric (anyone can score consistently)	Vague rubric (subjective scoring)
Covers edge cases (not just the happy path)	Only easy cases (doesn't stress-test the model)
Includes reasoning (explains why the answer is correct)	Just the answer (no reasoning)

Table 9.2: High-quality golden examples versus mediocre red flags

The HR QA pairs from *Chapter 6* all passed faithfulness, six out of six, with every answer grounded in the source policy. I expected at least one hallucination. The structured QA dataset was the one with the problem, not faithfulness but diversity. Starter diversity scored 0.417, meaning over half the questions opened the same way. Train on that, and the model learns the opener, not the question.

Determining evaluation set size

How big the set needs to be depends on what you are using it for; catching obvious bugs takes fewer examples than gating a production release. The following are the ranges I work from:

Use Case	Minimum Size	Recommended Size	Notes
Initial validation	50–100	200–300	Catch obvious failures
Model selection	200-300	500-1000	Statistical significance for A/B tests
Production monitoring	500-1000	2000-5000	Slice by query type, detect regressions
Compliance/safety	1000-2000	5000-10000	Cover all edge cases, adversarial inputs

Table 9.3: Recommended evaluation set sizes by use case

The takeaway is that set size scales with the stakes, not with ambition. A few dozen examples catch obvious bugs, model selection needs enough for statistical significance, and compliance or safety work needs thousands, because that is where the rare adversarial cases hide.

Example: building starter sets for the four assistant types

The notebook's final part builds four of these sets in front of you (search for *Part 10: Build Evaluation Sets* in the notebook). It takes the four assistant types from *Chapter 1* and runs the full loop this chapter taught on each one: a small source context as the ground truth, LLM-generated candidate questions using the *Chapter 6* pattern, the answerability judge from earlier filtering every pair, and the diversity check scoring what survived.

```
Personal assistant
  generated: 6  passed answerability: 6  starter_diversity: 1.00  unigram_ttr: 0.67
  example: What meeting times does the user prefer?

Medical assistant (fictional clinic policy)
  generated: 6  passed answerability: 6  starter_diversity: 1.00  unigram_ttr: 0.75
  example: According to the triage guideline, how quickly should a patient reporting
  chest pain receive an ECG after arrival?

Enterprise knowledge assistant
  generated: 6  passed answerability: 6  starter_diversity: 1.00  unigram_ttr: 0.68
  example: When does the new AI policy take effect?

Content moderation system
  generated: 6  passed answerability: 6  starter_diversity: 1.00  unigram_ttr: 0.76
  example: What action is taken when spam is identified under the moderation policy?
```

All six pairs passed for every type on this run, which is what should happen when the source contexts are two hundred words each; there is not much room for the generator to drift. The value is the loop, not the pass rate. These are seeds, fifty to a hundred times too small to ship, and you grow them the hybrid way Dan describes: real tickets and logs for realism, expert-written edge cases for coverage, and a quarterly refresh as production traffic drifts. But every method in that loop is one you have already run in this chapter, which means building a set for your own assistant is not a new skill. It is the same cells pointed at your data.

Selecting evaluation metrics by assistant type

You've built your evaluation dataset. Now you need metrics to score it. But which metrics?

The same metrics do not work across assistant types. A personal assistant lives and dies on personalization and privacy, a medical diagnosis system needs clinical accuracy and strict guideline adherence before you even look at

latency, and content moderation is all about precision and recall balanced against fairness because a false positive on moderation can be worse than a miss.

The metrics depend on what kind of assistant you build. *Chapter 1* defined four types, and each one needs different evaluation criteria.

Applying general metrics across all assistants

A handful of baseline metrics apply no matter what you build. Accuracy, which current evaluation frameworks mostly call correctness, asks whether the answer matches the reference, measured by exact match, F1 score, or semantic similarity depending on how strict you need to be. Relevance asks whether it addressed the question that was actually asked, usually a 1 to 5 judge score. Faithfulness, sometimes called groundedness, is the hallucination check: is every claim in the answer supported by the provided context? Completeness asks whether all parts of a multi-part question got answered, and fluency whether the text reads naturally, though with current models fluency is rarely the problem. Latency rounds out the set. It is a production requirement rather than a quality metric, but I track it next to the quality numbers because a perfect answer that arrives too late still loses the user.

At Clearwater Analytics, to improve the evaluation framework, Dan Siddall built an evaluation rubric with eight attributes. Four of them go beyond the basics above:

- **Coherence**: Is the answer internally consistent and logically structured? A model can get every fact right and still confuse the reader because the pieces are in the wrong order.
- **Reasoning**: Does the model show sound logical steps, not just a correct final answer? Getting the right answer through broken logic is a time bomb; it will fail on the next similar question.
- **Professionalism**: Is the tone appropriate for the domain? An enterprise assistant that sounds like a casual chatbot loses user trust fast, even when the information is correct.
- **Ethical Considerations**: Does the output avoid bias, harmful content, and privacy violations? Nobody tests for these until something

embarrassing shows up in production.

None of these dimensions show up in an accuracy score, which is exactly why a rubric that names them catches the failures a single number hides.

Seeing the difference in the notebook

Part 1b of the notebook (search for *"Part 1b: Beyond Accuracy"* in the notebook) runs the same Prompt A and Prompt B responses from the earlier prompt comparison through an eight-attribute judge instead of the single-score judge. The point is to show you what accuracy alone misses.

Step 1: defining the multi-metric judge

The multi-metric judge sends each response to the LLM with a rubric covering all eight attributes. Here is what it asks:

```
prompt = f"""Score this response on eight attributes (1-5 each).
1. Accuracy: Does the response match the reference answer?
2. Relevance: Does the response directly address the question asked?
3. Coherence: Is the response logically structured and internally consistent?
4. Context adherence: Does the response stick to provided information without fabrication?
5. Reasoning: Does the model support its statements with sound logic?
6. Professionalism: Is the tone appropriate for a customer support context?
7. Ethical considerations: Is the response free from bias, privacy issues, or harmful content?
8. Creativity: Does the response go beyond template answers when appropriate?

Return ONLY valid JSON with scores and one-sentence justifications for each."""
```

Step 2: scoring prompt B responses

The notebook scores Prompt B responses first. Remember, the basic judge from the earlier prompt comparison gave Prompt B an average of about 3.4 out of 5. That sounds bad but not terrible. Run it through the multi-metric judge, and the picture gets worse. Context adherence drops because the model fabricated generic policies instead of using the provided ones. Coherence drops because, without context, the model padded its answers with filler paragraphs about "checking the website" and "contacting customer service." The accuracy score alone hid both problems.

Step 3: scoring Prompt A responses

Then it scores Prompt A responses on the same eight attributes. Prompt A scored well on accuracy earlier, and it scored well across the board here too,

because the model had the policy text and used it. The side-by-side comparison table shows the gap per metric:

```
MULTI-METRIC COMPARISON: Prompt A (with context) vs Prompt B (no context)

Metric                  Prompt A   Prompt B      Delta    Winner
-----------------------------------------------------------------
accuracy                    4.80       3.00      +1.80         A
relevance                   5.00       3.60      +1.40         A
coherence                   4.80       3.20      +1.60         A <<<
context_adherence           5.00       2.40      +2.60         A <<<
reasoning                   4.60       3.40      +1.20         A
professionalism             5.00       4.20      +0.80         A
ethical                     5.00       4.80      +0.20       tie
creativity                  3.40       3.60      -0.20       tie
```

Step 4: comparing basic score vs. multi-metric score

The biggest gap is context adherence at +2.60. That is the fabrication problem. Prompt B invented policies that do not exist, and the basic accuracy judge gave it partial credit because the fabricated details sounded plausible. The multi-metric judge caught it. Coherence gap at +1.60 is the padding problem; the model rambled without context, and the single accuracy score did not penalize that.

Step 5: interpreting what the extra metrics caught

The summary cell at the end shows the per-question breakdown. For each Prompt B response, it finds the worst-scoring metric and prints the judge's justification. That tells you exactly which attribute failed and why, not just that something scored low.

This is why evaluation rubrics need more than one dimension. A single accuracy score hides fabrication, incoherence, and unprofessional tone. Run Part 1b in the notebook, and you will see it in your own data.

Now let's add metrics specific to each assistant type.

Personal assistant metrics (case study 1)

A personal assistant, the calendar-and-email kind from *Chapter 1*, lives on four things the general metrics never touch:

- Personalization: does the answer respect what the assistant knows about this user? The assistant that suggests a 2 pm meeting to someone

who only takes morning meetings is failing even when the suggestion is fluent, so I score preference alignment on a 0 to 1 scale.

- Privacy: an assistant that mentions your salary in an email draft has done real damage, so privacy violations get counted directly and the only acceptable count is zero.
- Proactivity: the share of interactions where the assistant usefully anticipates the next step; you add a meeting, and it offers to send the invite.
- Context retention across turns: "cancel the meeting I just scheduled" only works if the assistant remembers which meeting that was, so I measure how often its references to earlier turns are actually correct.

Domain-specific assistant metrics (case study 2: medical diagnosis)

A clinical decision-support assistant gets a stricter set, and the important parts cannot be automated. Clinical accuracy is scored 1 to 5 by a medical expert; board-certified physician review is the gold standard here, and there is no shortcut around the cost. Expert agreement gets measured with an inter-rater agreement metric, such as Cohen's Kappa, against a panel of at least three specialists—above 0.7, or the outputs are not trustworthy enough to ship. Guideline adherence checks that every recommendation cites the clinical protocol it came from; a medication suggestion should carry the FDA approval for that specific indication. Safety is a violation count with a target of zero; suggesting aspirin to a patient with a bleeding disorder is the kind of miss that ends the project. The last one is confidence calibration, measured as Expected Calibration Error (ECE) below 0.05: bin the model's predictions by how confident it claims to be, then check the accuracy inside each bin. A model that sounds equally sure when it is right and when it is wrong is dangerous in a clinic, and calibration is the number that catches it.

Enterprise knowledge assistant metrics (case study 3)

An enterprise knowledge assistant (internal documentation search and policy Q&A) is mostly a retrieval problem, so its metrics live on the retrieval side. Precision at 5 above 0.8 and recall at 10 above 0.9 are my production floors: of

the top five documents pulled, at least four should be relevant, and for critical queries the answer had better be somewhere in the top ten. Access control compliance is the one metric with zero tolerance. An assistant that surfaces a document the user has no permission to see is a security incident, not a quality issue, and no accuracy score buys that back. Freshness matters because policies change; retrieving last year's version of a policy that was updated last month is an error even when the text reads perfectly. And source attribution closes the set: every claim should cite a document that actually contains it, because a citation pointing at a document that never mentions the claim is worse than no citation at all; it manufactures false confidence.

Content moderation system metrics (case study 4)

Content moderation flips the frame entirely. It is a classifier, so precision and recall carry the weight, and the balance between them is a product decision rather than a technical one. I want precision above 0.95 because every false positive is a legitimate user getting censored, and recall above 0.90 because every false negative is harm reaching the community. I also track the false positive rate and false negative rate directly because they expose different operational risks than precision and recall alone. A false positive means legitimate content was incorrectly blocked, while a false negative means harmful content slipped through. Those are often the numbers that end up on the production dashboard, with the tolerance for misses tightening as the harm severity of the category goes up. Fairness gets measured directly, not assumed: slice the error rates by demographic group, race, gender, age, language, and hold the gap under 0.05, because a moderation system that flags one group's speech more often at the same content severity is a disparate-impact problem whether anyone intended it or not. And break precision and recall down per violation category. A system can run 0.95 on hate speech and 0.85 on spam, and the aggregate number hides exactly which one needs the work.

Metric selection framework

Pick 5 to 8 metrics and stop there: two or three general ones (accuracy, relevance, faithfulness), three or four task-specific ones from the case studies

above, and one or two business metrics like task completion or user satisfaction. With twenty metrics, you drown and miss the one that matters. And validate the set against reality. Track human satisfaction next to the automated scores, and any metric that does not correlate with it (or is below 0.5) gets dropped, whatever it claims to measure.

How to evaluate evaluation platforms

You know which metrics to track. But who calculates them? You could build your own evaluation pipeline. Or use a platform. As of June 2026, there are 30 or more to choose from. I surveyed the major ones in my 2024 article on LLM evaluation and hallucination detection [25], and the market has only gotten more crowded since. Some are all-in-one, some specialize in hallucination detection, some focus on datasets, some bolt evaluation onto observability. The names go stale fast—the Galileo acquisition above proved it mid-chapter—so do not memorize names. Score whatever exists when you arrive with the six questions as discussed in the sub-sections.

I run every platform through the same six questions, the one I use today and whichever one replaces it after the next acquisition. Each gets a 1 to 5, and I weigh them by what the project actually needs, because a startup shipping a prototype and a regulated enterprise are not scoring the same columns the same way.

Core capabilities

The first question is whether it actually evaluates the thing I am building. I want coverage across all four kinds of evaluation: offline runs on fixed datasets, online scoring on production traffic, human-in-the-loop review, and automated LLM-as-a-judge. Then I look at the metrics it ships with: RAG faithfulness and context relevance, safety checks for toxicity, PII, and jailbreaks, agent metrics like tool-selection accuracy if I am building agents, code-correctness metrics if I am generating code. A 5 covers every evaluation type and metric my use case needs. A 3 covers most but misses one or two that matter. A 1 covers less than half, and I stop looking.

Integration and developer experience

The second question is how much of my week it takes to get running. Does it plug into the stack I already run, LangChain or LlamaIndex, raw OpenAI or Anthropic or Azure, or my own framework? Is setup a pip install and two lines of code, or a multi-day infrastructure project? Are the docs real, with code examples and a quickstart, or a marketing page pretending to be documentation? I want a Python SDK at minimum, a REST API so I am not language-locked, and the option to run locally without shipping my data to someone else's cloud. A 5 drops into my stack in under half an hour with docs I never have to fight. A 3 integrates with workarounds and a few hours of setup. A 1 does not fit my stack, or it eats days before it runs once.

Production readiness

Third, will it survive contact with production? I ask what volume it handles, because a thousand evaluations a day is fine for a prototype, but I need to know it will not fall over at a hundred thousand or a million once the system is live. Latency matters if I want scoring in the inference path rather than after the fact, real-time meaning under about a hundred milliseconds of overhead; anything slower is monitoring or batch only. And for anything touching regulated data, I look hard at privacy: a self-hosted option so data never leaves my infrastructure, certifications like SOC 2, ISO 27001, and HIPAA (Health Insurance Portability and Accountability Act) where healthcare is in play, PII redaction built in. A 5 scales to my volume with low latency and a self-hosted path. A 3 scales but is cloud-only. A 1 cannot handle my volume, or it is missing a compliance box I cannot ship without.

Evaluating costs at scale

Fourth is money, and the sticker price is the least of it. I want the pricing model, per evaluation or per seat or flat, the free tier and exactly where it cuts off, and the hidden costs that only show up at volume. The judge-model API bill sitting underneath a per-eval charge is the one that surprises people. Then I run the math at my actual scale, not the vendor's tidy example, and weigh it against what evaluation saves, the bad releases it catches, and the incidents that never reach users. A 5 is affordable at my volume with no nasty surprises. A 3 works but the bill climbs faster than I would like. A 1 prices me out the moment I am past the toy stage.

Differentiation

Fifth, what does this one do that the others do not, and can I trust it to still be here next year? At the time of writing, the strongest hallucination detection belonged to Galileo, which I partnered with at Clearwater Analytics starting in 2024. Others lead on cross-functional workflows, or dataset management, or bolting evaluation onto observability, and the open-source options trade polish for control. I weigh maturity too, since a platform that launched years ago and is still shipping is a safer bet than a feature-rich one that appeared last quarter. A 5 solves my specific problem better than anything else and has the track record to back it. A 3 is competent with no clear edge. A 1 is either undifferentiated or too new to trust in production.

Lock-in and portability

Last, how hard is it to leave? I ask whether I can export everything (raw data and scores and metadata, not just the aggregated numbers) and whether that export is a standard format or a proprietary one I would have to convert. I check whether I am tied to their choice of LLM providers or free to bring my own. And I note where it sits on the line from fully open-source, which I can fork and audit, through open-core, basics free and the useful parts paid, to fully proprietary, a SaaS I cannot see inside. A 5 gives me full export in a standard format, provider-agnostic, open where it counts. A 3 is partial export and convertible formats. A 1 locks my data in with no clean way out, and that is the one answer I will not accept for anything I plan to run for years.

From criteria to decision

The mechanics from here are short. Weigh the six criteria by what your project actually needs; a healthcare startup puts its heaviest weight on production readiness for the HIPAA box alone, a prototype team weights integration and cost. Score each candidate 1 to 5 per criterion, multiply, and the weighted totals will usually narrow a field of thirty platforms to two or three. Do not decide from the spreadsheet. Pilot the finalists for a week or two on your own data, 50 to 100 evaluation examples on your real stack, and pay attention to how the documentation and the support hold up while you work, because that is what living with the platform feels like. The build-versus-buy call follows the same logic it always has: open-source frameworks like RAGAS and

DeepEval when engineers are the only users and the budget is tight, a commercial platform when product, QA, and domain experts all need the workflows, and building your own only when compliance forces everything in-house, and you can staff the maintenance forever.

Summary

In this chapter, I walked through how I evaluate the data behind an LLM system, before it ships and after. We started with a golden dataset and used it like a unit test suite, running two prompts against the same questions and reading the regression report. Then we evaluated the RAG pipeline, with retrieval and generation scored separately, so we know which half broke. After that, we validated the synthetic data from *Chapter 6* across four checks: faithfulness, answerability, diversity, and preference-pair judging with a position bias check, and we scored the base model against *Chapter 7's* behavioral contract, asking the harder question of whether the fine-tuning was needed at all, with the delta cell ready to grade the fine-tuned model wherever one exists. The notebook closes by building starter evaluation sets for all four assistant types with the same tools, so the loop you take to your own project is one you have already run. All of that is offline. Then we moved to online evaluation on live production traffic, and we used a ready-made platform, Galileo, as the worked example.

If there is one thing I want you to take from this chapter, it is to start evaluation before you think you are ready. Every team I have worked with wished they had built the evaluation dataset earlier, and by the time the user complaints arrive, you have already lost weeks.

The scores also point to the fix. A low faithfulness score usually traces back to noisy or contradictory retrieval documents, a low relevance score to poor question-context pairing, a low accuracy score to incorrect reference answers in the training data, and bias flags to skewed training data. In the next chapter, we move from finding these problems to fixing them, with data quality, root cause analysis, human review workflows, data correction pipelines, and the quality assurance gates that keep bad data out in the first place.

References

These are the sources I used in this chapter in the order they are cited:

- [1] Elhousieny, R. (2024, August 22). *Understanding LLM Hallucination and Strategies to Mitigate It*. LinkedIn Pulse / AI Solutions Architect Newsletter. https://www.linkedin.com/pulse/understanding-llm-hallucination-strategies-mitigate-rany-m4uyc/
- [2] LangChain. (2025, December). *State of Agent Engineering*. https://www.langchain.com/state-of-agent-engineering (Survey of 1,340 AI professionals, November 18 to December 2, 2025. Human review 59.8%, LLM-as-a-judge 53.3% among evaluation methods in use.)
- [3] Siddall, D. (2025, January 30). *Generating Diverse Questions for LLM Instruction Fine-Tuning*. *Medium/cwan-engineering*: https://medium.com/cwan-engineering/generating-diverse-questions-for-llm-instruction-fine-tuning-strategies-and-considerations-73270f2b0adb
- [4] Es, S., James, J., Espinosa-Anke, L., and Schockaert, S. (2024). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. EACL 2024 Demo. arXiv:2309.15217. https://arxiv.org/abs/2309.15217
- [5] OpenAI Developer Community. (2026, May*)*. *OpenAI is winding down the fine-tuning API and platform*. https://community.openai.com/t/openai-is-winding-down-the-fine-tuning-api-and-platform-discussion-thread/1380522 (Organizations without prior fine-tuning history blocked from new jobs as of May 7, 2026; all new job creation ends January 6, 2027; existing fine-tuned models keep serving until base model deprecation.)
- [6] Siddall, D. (2024, August 20). *A Cutting-Edge Framework for Evaluating LLM Output*. Medium/cwan-engineering.https://medium.com/cwan-engineering/a-cutting-edge-framework-for-evaluating-llm-output-edab53373514
- [7] *RAGAS documentation*. (2026). https://docs.ragas.io/en/stable/
- [8] *Measuring Diversity in Synthetic Datasets*. (2025). arXiv:2502.08512. https://arxiv.org/abs/2502.08512
- [9] Elhousieny, R. (2024). *Revolutionizing AI with Predibase: The Future of Serverless, Fine-Tuned LLMs*. LinkedIn Pulse / AI Solutions Architect

Newsletter. https://www.linkedin.com/pulse/revolutionizing-ai-predibase-future-serverless-llms-rany-l7nnc

- [10] Galileo, Clearwater Analytics, and Predibase (2024). *Webinar: How Clearwater Analytics Creates Agentic Systems with SLMs*. YouTube / Galileo AI. https://www.youtube.com/watch?v=6utcRSblxFI
- [11] Predibase. (2023). L*oRAX: Multi-LoRA inference server that scales to thousands of fine-tuned LLMs*. GitHub. https://github.com/predibase/lorax
- [12] Elhousieny, R. (2024, February 29*)*. *Fine Tune mistral-7b-instruct on Predibase with Your Own Data and LoRAX*. Medium/Level Up Coding.https://levelup.gitconnected.com/fine-tune-mistral-7b-instruct-on-predibase-with-your-own-data-and-lorax-b12b89c64874
- [13] Cherry, D., Siddall, D., ElHousieny, R., Ganapathy, P., and Redondo Sanchez, P. (2024, December 13). *How Clearwater Analytics is revolutionizing investment management with generative AI and Amazon SageMaker JumpStart*. AWS Machine Learning Blog. https://aws.amazon.com/blogs/machine-learning/how-clearwater-analytics-is-revolutionizing-investment-management-with-generative-ai-and-amazon-sagemaker-jumpstart/
- [14] Rubrik. (2025, June 25). *Rubrik to Acquire Predibase to Accelerate Agentic AI Adoption*. Press release. https://www.rubrik.com/company/newsroom/press-releases/25/rubrik-to-acquire-predibase-to-accelerate-agentic-ai-adoption
- [15] Elhousieny, R. (2023, November 15). *The Art of Fine-Tuning Large Language Models (LLMs)*. Medium/Level Up Coding.https://levelup.gitconnected.com/fine-tuning-llm-enhancing-ais-understanding-with-specificity-a5f62c51bea4
- [16] Cherry, D. (2024, May 2). *Quote in Galileo Introduces Protect, a Real-Time Hallucination Firewall to Safeguard Enterprise Generative AI*. BigDATAwire. https://www.hpcwire.com/bigdatawire/this-just-in/galileo-introduces-protect-a-real-time-hallucination-firewall-to-safeguard-enterprise-generative-ai/
- [17] Elhousieny, R. (2024, September 26). *Galileo Protect with LangChain - Real-Time AI Hallucination Firewall*. LinkedIn Pulse / AI Solutions

Architect Newsletter. https://www.linkedin.com/pulse/galileo-protect-langchain-real-time-ai-hallucination-rany-yocdc/

- [18] Elhousieny, R. (2024, July 29). *ChainPoll: A Revolutionary Method for Detecting LLM Hallucinations*. LinkedIn Pulse / AI Solutions Architect Newsletter. https://www.linkedin.com/pulse/chainpoll-revolutionary-method-detecting-llm-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-fjh5c/
- [19] Elhousieny, R. (2024, July 16). *Luna: A Breakthrough in Hallucination Detection for Language Models*. LinkedIn Pulse / AI Solutions Architect Newsletter. https://www.linkedin.com/pulse/luna-breakthrough-hallucination-detection-language-rany-j5tef/
- [20] Elhousieny, R. (2024, July 29). *Ensuring Factual Accuracy with Galileo Correctness*. LinkedIn Pulse / AI Solutions Architect Newsletter. https://www.linkedin.com/pulse/ensuring-factual-accuracy-galileo-correctness-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-jspec/
- [21] Network World. (2026, April). *Cisco to acquire Galileo for AI observability*. https://www.networkworld.com/article/4156855/cisco-to-acquire-galileo-for-ai-observability.html
- [22] Confident AI / DeepEval. (2026). *The LLM Evaluation Framework*. https://github.com/confident-ai/deepeval
- [23] Langfuse. (2026). *Open Source LLM Engineering Platform*. https://langfuse.com/
- [24] Braintrust. (2026). *DeepEval Alternatives 2026*. https://www.braintrust.dev/articles/deepeval-alternatives-2026
- [25] Elhousieny, R. (2024, July 29). *LLM Evaluation and Hallucination Detection*. Medium/Level Up Coding.https://levelup.gitconnected.com/llm-evaluation-and-hallucination-detection-01ba8a30c417
- [26] Clearwater Analytics. (2026, April). *1,000+ AI agents. 94% accuracy on unstructured documents. Financial close cycles cut in half*. LinkedIn (corporate post). https://www.linkedin.com/posts/clearwateranalytics_1000-ai-agents-94-accuracy-on-unstructured-activity-7450513883238338561-eeUu

10

Data Quality and Human Oversight

The previous chapter focused on evaluation and showed how to identify failures in LLM applications. Low faithfulness indicates problems with the information used to ground the assistant's answers. Low relevance exposes poor relationships between questions and the provided context. Low correctness signals wrong answers and can also expose incorrect reference data. Low coverage reveals gaps or imbalances in the underlying datasets. Evaluation acts as the warning system and deployment gate, but it does not solve the problems. The next step is addressing these problems.

At the application layer, data quality usually matters more than data volume. Adding another thousand documents to a retrieval corpus does not help if they introduce outdated or contradictory information. Similarly, adding thousands of training examples does not help if they encode inconsistent behavior. More data helps when it adds useful coverage without introducing noise.

Data quality is not an absolute property of the dataset itself. Dataset quality is defined by the application and how it consumes the data. For example, an authoritative policy document might be correct but does not represent the realistic user questions to be used for evaluation. A production conversation might contain spelling mistakes, incomplete sentences, and unnecessary exchanges, but this is exactly what makes it valuable for representing real user behavior. Clean data is not necessarily high-quality data, and imperfect data is not necessarily low quality.

This distinction matters because it changes your approach to data quality. Instead of seeking clean datasets, you should seek data that fits its intended use. For retrieval data, authority, freshness, and relevance are the most important properties. For evaluation data, correctness, coverage, and representativeness matter more. The quality criteria should follow the application requirements rather than a universal definition of good data.

When you detect a problem, root-cause analysis is usually harder than detecting the problem itself. Many problems originate from data issues, but even within data, there is a spectrum of possible causes. An incorrect answer may originate from an outdated source document, a parsing error, a bad transformation, or a stale vector index. Therefore, fixing data quality issues requires tracing failures to their sources, correcting the affected data, and rebuilding downstream artifacts as needed.

Human oversight is a key part of this process. Automated checks and LLM judges allow you to evaluate data at a scale that manual review cannot match, but automation does not remove the need for human judgment. Multiple automated judges can identify disagreements and ambiguous cases that deserve closer human review. Random human sampling provides another important quality control by detecting systematic errors that automated judges may skip. **The goal of human oversight is to improve data quality in a scalable way by minimizing human effort**.

Data quality is not a one-time cleanup activity. It is an ongoing production process with multiple pillars. Data changes continuously as documents are updated, business processes evolve, and production distributions shift. Quality gates prevent known problems from entering the datasets and indexes. Metadata, provenance, and versioning make the changes traceable and reproducible. Monitoring identifies when previously acceptable data is no longer valid.

In this chapter, you will learn to manage data quality by working through the following topics:

- Identifying data quality problems: use evaluation signals to identify potential problems in retrieval, training, and evaluation data, and define data quality based on the application layer

- Diagnosing and correcting data failures: Trace failures back to their sources, identify root causes, correct the affected data, and rebuild downstream artifacts when necessary
- Scaling human oversight: Combine multiple automated judges, disagreement analysis, random sampling, and human review to maintain quality in a scalable way
- Storing LLM data: Organize authoritative source data and derived artifacts across data stores, data lakes, and vector indexes, and select infrastructure based on the role of each data asset
- Governing and monitoring data quality: Apply quality gates, metadata, provenance, and versioning to control data changes, and monitor data health and distribution shifts in production
- Closing the data quality loop: Connect evaluation, diagnosis, human oversight, correction, governance, deployment, and monitoring into a continuous production workflow

These topics take you from detecting problems to correcting them and preventing them from returning. The first section discusses the signals produced by evaluation and how to use them to identify the data quality problems that deserve investigation.

Identifying data quality problems

You have established evaluation as the mechanism for detecting failures in LLM applications. Evaluation scores tell you that one or more components of the system are not performing as expected, but they do not tell you exactly why. For example, when the faithfulness score drops, the retrieved data might be wrong, or the model might be ignoring correct context. When correctness drops, the model might be producing wrong answers, or the reference answers used to evaluate them might themselves be wrong. Evaluation provides the signal, and data quality analysis determines whether data is the root cause.

This distinction is important because changing the data is not always the right response to an evaluation failure. If retrieval returns correct and authoritative documents but the model ignores them, improving the retrieval

corpus will not solve the problem. Before changing a dataset, you need evidence that the data contributes to the observed failure.

Chapter 9, introduced several evaluation signals that provide useful starting points for this investigation. Low faithfulness, relevance, correctness, and coverage can all indicate problems in the underlying data. *Table 10.1* summarizes these relationships:

Evaluation signal	Potential data problem	Example	First diagnostic check
Low faithfulness	Noisy, stale, or contradictory retrieval data	Retrieved chunks contain different versions of the same information	Inspect the retrieved context and compare it with the authoritative sources
Low relevance	Poor question-context pairing or irrelevant context	An evaluation question is paired with a document that mentions the topic but does not answer the question	Verify that the expected answer can be derived from the provided context
Low correctness	Incorrect reference or ground-truth data	The model gives one answer, but the reference answer is different	Validate the reference and model answer against authoritative sources
Low coverage	Missing or unrepresentative data	Evaluation failures concentrate in a domain or user category with limited representation	Compare the affected slice with the overall distribution and coverage of the underlying data

Table 10.1: Connecting evaluation signals to potential data quality problems

These mappings are starting points for diagnosis rather than final root-cause conclusions. Take low faithfulness as an example. The first step is to check the

context supplied to the model. If the retrieved chunks contain outdated and conflicting policy versions, that is clear evidence of a data problem. However, if the retrieved context is current, authoritative, and sufficient to answer the question, the investigation should move away from data to the generation layer, including prompt instructions and few-shot examples. In the second case, modifying the retrieval corpus is just additional work that does not address the failure.

The same idea applies to training data. *Chapters seven* and *eight* discussed supervised, preference, and conversational data and their impact on model behavior. In this section, you find evidence of behavioral problems in the data when evaluation identifies behavioral regression. For example, consider an assistant that does not make the correct decision on human approval escalations. Before going back to model training, inspect the failure examples related to this decision boundary. If similar situations are paired with different decisions in the training data, the dataset is probably contributing to the failure. However, if the examples consistently encode the intended behavior, the investigation should move somewhere else.

Applying the same methodology to preference data, evaluation may show that the model is making the wrong trade-offs between candidate behaviors. More preference data is not automatically needed. Take a sample of labeled preference data and inspect whether reviewers make consistent choices for similar scenarios and whether the criteria behind these choices reflect the application's current requirements. When reviewers disagree, adding more preference pairs does not help and makes the problem worse because it complicates the debugging process. *Chapter 8* established reviewer agreement as an important quality requirement for preference data. Here, we emphasize that preference disagreement is a diagnostic signal that tells you where to investigate.

Finally, moving to production feedback. User ratings and business outcomes can reveal unique problem areas that nothing else covers, including problems in the evaluation data itself. However, the production feedback may be sparse or unrepresentative. *Chapter 8* established that raw user feedback should be treated as a signal rather than training data. When negative feedback is high, you need to determine whether the feedback represents broad failures or failures for specific user slices or scenarios. The feedback could also be noise in

the feedback mechanism or users wanting a behavior against the application policy.

These examples support the main idea that data quality is dependent on its application and cannot exist independently of the role it plays. Retrieval data requires authority, freshness, and relevance for the retrieved information to be trusted. Training data requires consistency in encoding the target behavior across relevant scenarios. Evaluation data requires correct references and sufficient coverage of the scenarios that matter. Production feedback must represent actual application outcomes to support decision-making.

These criteria are not theoretical. **Data quality criteria must be measurable and actionable**. For example, to measure and fix data freshness, you must version and date the corpus, and you need the ability to remove outdated and replaced documents. To guarantee correctness of evaluation data, you must be able to trace reference answers back to authoritative sources. To achieve good coverage, the important categories observed in production traffic must be recorded, and you need the ability to compare your data against these categories. The exact criteria are different between applications, but each criterion should connect an application requirement to something that can be measured and inspected.

To respond to an evaluation failure, start with the failed examples rather than the aggregate score alone. Identify the data that contributed to each failure and the quality properties that matter for this dataset in its current role. Test those properties against trusted sources and application requirements. When the evidence points to a specific dataset, use the failed examples and quality criteria to determine what needs to change.

Once a data problem is identified, the next step is to trace the failures through the data pipeline, identify their root causes, and determine what needs to be corrected.

Diagnosing and correcting data failures

This section focuses on identifying the data that caused evaluation failures, how the problem happened in the first place, and how to fix it. Root-cause analysis starts with the failed examples, tracing backward from there. This approach is more effective than auditing entire datasets; LLM applications

might depend on millions of documents and training examples. Most of them have nothing to do with any specific failure. Starting with the failed example narrows the investigation to specific information that influenced the model behavior.

The diagnostic path depends on the data consumption dependency chain within the application. For a typical retrieval-augmented generation (RAG) application, start with inspecting the generated answer and the exact chunks retrieved for the request. Trace suspicious chunks to the indexed documents, then to the parsed documents and original data sources. For a training regression, identify the scenarios that regressed and inspect the training examples that represent similar and related scenarios and the examples that represent similar behavior. For an evaluation problem, inspect the reference answer and trace it through the labeling or generation process to the data source used to establish the expected result.

Regardless of the data type, the approach is to start with the failed example and trace backward until finding the earliest point where the data became wrong or incomplete. The following sections apply this approach to tracing failures, correcting their root causes, and rebuilding the artifacts affected by the correction.

Tracing failures to their sources

Root-cause analysis follows the data dependency chain in reverse order. *Figure 10.1* shows the diagnostic path moving backward from an observed application failure through the model input, derived data, and transformation back to the original data source. This root-cause analysis narrows the investigation to the specific data that contributed to the failure rather than auditing the entire dataset:



Figure 10.1: Tracing a data failure to its source

Consider a RAG-based enterprise assistant that provides an outdated answer to an employee question. The first thing to investigate is the retrieved context. Assume one of the retrieved chunks contains an old version of the company

policy. Deleting this chunk from the vector index fixes this specific example, but it does not explain why the old policy entered the index. It also does not tell whether other outdated chunks from the same document exist.

Instead of deleting the chunk, trace it backward. Identify the document it came from, the parser that was used to process the document, and the source system that contained the original document. You might discover that a fork of the document was placed in a side folder before the latest version was published and remained there. The ingestion pipeline ingested both documents, parsed both, and added chunks from both versions to the index. In this case, the vector database exposed the problem but did not create it.

Chapter 9, showed a simpler version of this problem when the shared vector store accumulated test and demo entries along with the intended documents. Removing those entries cleans the current index, but root-cause analysis goes further into understanding why the ingestion process added them to the production index in the first place.

The same method applies outside the RAG pipeline. Assume that a fine-tuned assistant makes a wrong approval decision for a particular workflow. In this case, start by identifying the training examples that represent the failed scenario. Then, trace these training examples back to their source conversations, documents, or generation process. If several examples encode an outdated business rule, the training data needs to be fixed rather than patching the model with a few additional examples.

Evaluation data can cause similar failures. A low correctness score is probably because the model response contradicts the reference answer. Do not assume that evaluation data is always perfect. Trace the reference answer to its source or validate it with human subject matter experts. If the reference was generated from an old document or labeled according to outdated guidelines, the evaluation is measuring the model against the wrong target. In that case, modifying the model to increase the correctness score would make the application worse.

Traceability is important when multiple sources disagree. *Chapter 8*, established the importance of identifying sources of truth rather than treating all documents as equal. During root-cause analysis, this identification is part of the process if it is not already known. If a policy document conflicts with a

wiki page, the investigation should find which source contains the right information and why the lower-authority copy was allowed to influence the application.

Some authority conflicts can be resolved automatically when the precedence rules are explicitly defined. Other conflicts require human judgment. When the available metadata and rules cannot judge the winning source, the conflict should be reviewed by subject matter experts rather than arbitrary technical rules.

Correcting the root cause

Different data failures require different corrections. Some records need to be removed, while others need to be replaced, re-annotated, or regenerated. If some information is completely missing, the relevant dataset needs to be expanded to cover the missing information. **The appropriate correction depends on the root cause identified during diagnosis**. *Table 10.2* summarizes common data quality failures and the most common corrective actions that address them:

Data problem	Typical corrective action
Stale information	Remove the outdated data and replace it with the current version
Conflicting sources	Determine source authority and resolve the conflict or explicitly represent the conflict with relevant metadata
Incorrect label or reference	Validate against trusted evidence or a human expert and correct the label
Duplicate data	Deduplicate while preserving the authoritative record
Parsing or extraction failure	Correct the processing logic and reprocess from the original source
Transformation error	Correct the transformation logic and regenerate the affected data

Data problem	Typical corrective action
Missing coverage	Collect, label, or generate data for the missing scenarios
Distribution imbalance	Expand or rebalance the affected slices based on application needs through up-sampling, down-sampling or generation of new data
Incorrect source authority	Correct the authority metadata and downstream source-selection rules
Poor question-context pairing	Regenerate the affected pairs

Table 10.2: Common data quality failures and corrective actions

The corrective action should preserve the history of the data while producing a new state that is consumable by downstream systems. Depending on the root cause, the faulty data can be corrected, replaced, or retired.

Correct an artifact when its identity remains the same after correction. For example, an evaluation sample might contain a correct employee travel question and source policy, but the reference answer lists the wrong reimbursement limit. Updating the reference answer does not change the question and source policy, so correct the reference answer and create a new version of the evaluation dataset.

Replace an artifact when a newer version deprecates the old one. For example, when the company publishes a new travel policy, the new document replaces the previous policy as the authoritative source. Keep the old version for history and reproducibility but make sure that downstream retrieval uses only the new version.

Retire an artifact when it should no longer be processed but should remain available for traceability. For example, assume an internal travel FAQ was previously approved for the assistant's use, but later, an official travel policy is published and becomes the only supported source. This is not a direct replacement. Retain the FAQ and its provenance for reproducibility of previous application behavior, but mark it as retired so the system excludes it from future retrieval and index rebuilds.

Before propagating a correction, validate it against the evidence used to identify the failure. A corrected reference answer should be checked against its authoritative source. A replacement policy should be verified for authority and effective date. Only validated corrections should propagate to downstream artifacts.

Corrective actions range from fully automated to primarily human-driven. For example, duplicates with identical content hashes can usually be handled automatically. A good LLM judge can also validate relatively complex cases when it has sufficient evidence and clear decision criteria. An LLM judge might compare an answer against an authoritative policy, identify contradictions between documents, and determine whether a generated question can be answered from a source document.

There is no fixed boundary between automation and human review. The practical boundary depends on the available evidence, the reliability of the automated judgment, and the impact of a wrong judgment. As automated judges improve, more quality decisions can be handled without reviewing every example manually. Human oversight remains necessary for cases where the available evidence does not support a reliable automated decision, and when domain expert judgment is required.

Human review is also useful when the correction itself is automated. Sampling and reviewing the corrected examples provide a quantifiable measure of the autocorrection quality and increase confidence in the overall automation process. An automated process that performs well today can regress when document formats and data distributions change. **Automation reduces manual work but does not eliminate the need for human oversight**. This chapter develops the human oversight model.

Correcting the root cause alone does not fully resolve the problem. In LLM applications, data produces other data. When the source data changes, some of the derived artifacts must be updated as well.

Rebuilding affected data

Data pipelines create intermediate dependencies. For example, a single source document can produce parsed text, chunks, embeddings, and vector-index entries. Another document can seed synthetic question-answer pairs included

in training or evaluation datasets. **When upstream data is corrected, rebuild all the derived artifacts along the dependency path**.

Back to the outdated policy example. If the source document is replaced, the parsed representation of the old document is no longer valid. Neither are the chunks created from it, their embeddings, or their entries in the vector index. The resolution path is to correct the source document, re-parse, re-chunk, re-embed chunks, update the index, and finally re-evaluate.

By contrast, suppose the source document and generated evaluation question are correct, but the wrong reference answer was assigned to the question. In this case, the correction starts much later in the dependency chain. Correct the reference, update the affected evaluation dataset, and rerun the evaluations. There is no reason to re-parse source documents or rebuild the retrieval index.

This approach also prevents the same data problems from happening again. Consider an engineer who removed an outdated chunk directly from the vector database but left the source document and ingestion pipeline unchanged. The same chunk may return to the index during the next index rebuild. Similarly, if an evaluation reference is corrected in the evaluation report but not in the underlying dataset, the next evaluation run will reproduce the same error. **Corrections must be made at the earliest faulty point in the dependency chain and propagated forward**.

Dependency-aware correction requires awareness of the relationship between data artifacts. At small scale, simple techniques, such as filename patterns and dataset manifests may be sufficient. At production scale, metadata, provenance, and versioning are required to explicitly capture these relationships. You will return to these mechanisms later in the chapter while discussing governance of data quality.

The diagnostic and correction process can therefore be summarized in these steps. Identify the failed examples, trace the data backward to the earliest faulty artifact, correct the problem in this artifact using automation or human judgment as appropriate, rebuild the downstream artifacts affected by the correction, and finally rerun the relevant evaluation. The complete diagnostic and correction workflow is illustrated in the following diagram:

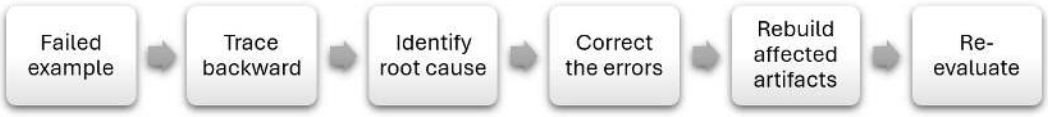


Figure 10.2: Tracing and correcting data failures

At this stage, you have a correction pipeline that can operate on large datasets at scale. The next challenge is maintaining human oversight while minimizing manual effort. Scaling human oversight requires combining automated judges with targeted human review and independent sampling.

Scaling human oversight

As data volumes increase, reviewing every record manually and accurately becomes impractical. This does not reduce the importance of human oversight, but it changes how you want to apply it. Human oversight is required for data quality, while minimizing manual effort.

Chapters six and *nine* introduced human-in-the-loop workflows from two different perspectives. *Chapter 6,* used human review to validate generated data, while *Chapter 9,* combined human evaluation with LLM judges to evaluate application responses. The same principles apply to the broader data quality process. Automated judges achieve scale, while humans validate the automatic judgments, resolve cases that automation cannot handle, and continuously improve the quality process itself.

A scalable human oversight strategy uses several mechanisms together. Multiple independent automated judges evaluate large volumes of data and escalate disagreements to human review. Human reviewers resolve ambiguous and disagreement cases. Human reviewers also take an independent random sample and audit the automated process, including cases where automated judges agreed. Targeted sampling provides additional coverage for known quality problems until full resolution.

These mechanisms serve different purposes, and they are not alternatives to each other. Together, they make a comprehensive production-scale human oversight process. You start with automated judges for large-scale quality assessment, then discuss how to prioritize human review and use human decisions to continuously calibrate the quality process.

Using automated judges for scale

LLM judges enable semantic quality checking on datasets that are too large for human review to complete. *Chapter 9,* introduced LLM-as-a-judge for evaluating model responses. For data quality, the same approach can apply to multiple quality tasks. For example, an LLM judge can evaluate whether a reference answer in the evaluation dataset agrees with authoritative sources. An LLM judge can validate whether a question can be answered from the associated context. An LLM judge can identify contradictions between records. It can also validate whether a generated example satisfies the labeling guidelines.

Automated judgment works best when the judge receives enough information to make the decision. Consider validating a reference answer from the evaluation dataset against company regulations. When the judge receives the question, the reference answer, current authoritative regulations, and a clear correctness metric, it has enough evidence to make a useful judgment. Asking the same judge to do the same task without supplying the regulations forces it to depend on its own knowledge, which is clearly not the desired decision.

This distinction is especially important for domain-specific data. A judge should not infer any of the authority hierarchy, business rules, or labeling guidelines, because all of these are properties of the domain rather than common knowledge. *Automated judgment is reliable when the decisions are grounded in explicit evidence and criteria*.

Using multiple judges provides an additional layer of judgment quality. Run the same quality check with multiple judges and compare their results. However, using identical judges does not provide any additional value, as they tend to share the same behavior and make the same mistakes.

Judge diversity is a must and can be introduced in several ways, such as using different models or different prompts or rubrics. Judges can also focus on separate dimensions, such as correctness, groundedness, and response style. Considering the additional cost of using multiple LLM judges, the best balance is to make enough judgments that expose uncertainty and disagreements with a minimal number of LLM calls.

For example, assume two judges evaluate whether a generated answer is grounded in source data. One of the judges claims the answer is fully grounded, and the other one identifies an unsupported numerical claim. The disagreement provides more information than just an average score. An example should be inspected because at least one judge raised a concern about groundedness.

Even full judge agreement is not proof of correctness. Multiple judges can agree because they share the same blind spot. This is why using multiple judges does not eliminate the need for human sampling. Multiple judges expose disagreements, while human sampling verifies the automated process. The next section focuses on applying human attention.

Prioritizing human review

Human review is expensive in comparison to automated evaluation, so human effort should be allocated intentionally. This section focuses on two complementary mechanisms: routing cases to human review based on known signals, such as disagreements and known problems, and independently sampling random cases.

Disagreement between automated judges is a strong routing signal. When judges give different conclusions about the same example, the system has enough evidence to doubt the decision. Instead of averaging the scores and hiding the disagreement, route the example to human review.

A higher disagreement rate may indicate that the evaluation rubrics or the evidence supplied to the judges need improvement. A low disagreement rate does not always mean high-quality because multiple judges can share the same blind spots. As a practical starting point, a disagreement rate of 5-15% can be used to trigger investigation and calibration of the automated judges. This range is consistent with public production reports. For example, Dropbox reported an initial 8% disagreement rate between its LLM judge and human evaluators (Dropbox, 2026). The appropriate range should be calibrated against human agreement rates and the acceptable risk level for each application.

Consider two documents that provide different versions of the same information. One judge selects the newer document because of its date from

the metadata, while another selects the older document because it comes from a more authoritative source. You may consider using a third judge to make the final decision, but the available metadata alone is not sufficient to resolve the conflict. A human reviewer would examine each piece of evidence and determine the more authoritative source. The reviewer would then correct the source metadata or precedence rules that created the ambiguity so that the problem does not happen again.

Some human reviews require domain expertise. A general reviewer can handle cases where the decision criteria are clear, and necessary evidence is available. For example, checking whether a numeric claim is grounded does not require specialized expertise. Domain experts are required when the decision depends on deep domain knowledge or expert judgment that cannot be completely represented to the reviewer. Some example cases that require domain expertise include resolving conflicting financial interpretations, reviewing legal rules, and examining medical evidence.

Another routing approach is targeted sampling for known quality problems. Once a quality problem is known, sample data from the affected slice while iterating on the solution. For example, if an optical character recognition pipeline is producing errors in scanned financial documents, increase review coverage for those documents. Similarly, if evaluation shows weak coverage for a particular language or intent, sample that slice more heavily while correcting it.

Targeted sampling should normally be more aggressive than steady-state random sampling because it focuses on a known failure. Reviewing approximately 10-30% of the target slice is a useful starting range during correction. However, for small and high-risk slices, a human may review all affected examples.

Routing-based reviews are limited in the sense that they find problems only if they are already detected prior to the human review. Automated judges may agree and still be wrong. Random human sampling detects failures that automated processes do not surface.

For large production datasets, reviewing approximately 1-5% is a good practice, with higher coverage during initial development and deployment, after significant pipeline or judge changes, and for higher-risk applications. At

a larger scale, statistical tests are required to define the review sample size. The sample should be large enough to detect the error rates that are acceptable for the application with acceptable confidence.

Random sampling should be totally independent, meaning it may include both high-confidence and low-confidence examples. If human reviewers repeatedly find errors among high-confidence records that all judges approved, then the automated quality process has a systematic blind spot. The issue may be related to the judge model, the quality rubric, or the evidence provided to the judges.

Random review also gives a quantified measure of the overall accuracy of the automated quality process. Routing-based approaches cannot achieve that because they are intentionally biased toward difficult cases, as opposed to random sampling, which provides an independent view.

These approaches complement each other. Disagreement-based routing addresses low-certainty automated decisions. Targeted sampling accelerates iteration on known failures. Random sampling audits the overall system. Human decisions do more than resolve the individual examples sent for review. They audit and guide the entire quality process.

Calibrating the quality process

The goals of human review go beyond fixing individual examples to calibrating and improving the automated quality process. When a reviewer corrects an example and moves to the next example, the automated process that produced the error remains unchanged and would produce it again.

Human oversight is more valuable when reviewer decisions are used to calibrate the automated quality process. The agreement rate between automated judges and human reviewers is one key metric. Track this metric over time and break it down by quality dimensions, data sources, scenarios, and other relevant slices.

Aggregate agreement rate alone is not enough. Consider the disagreement patterns as well. If disagreements are concentrated around specific issues, such as document authority or groundedness judgment, it may indicate problems with the provided metadata or judgment rubrics. These findings should result in concrete changes.

For example, assume human reviewers agree with the automated judges on 95% of the sampled data overall, but the agreement rate drops to 75% for questions impacted by document authority ranking. Inspecting the disagreements clarifies that the automated judges prefer the newest document, while human reviewers consider the source hierarchy. The overall agreement rate may hide this failure if authority conflicts represent a small portion of the whole dataset.

In this case, a useful calibration is to add source-authority metadata to the retrieval corpus and update the rubrics to cover authority prioritization rules. After this, rerun the previous disagreement examples and use targeted sampling on potential authority-conflict cases. If the agreement rate goes back to the required level and remains stable, deploy the updated judge configuration while continuing independent random sampling to detect other blind spots.

The calibration process is a repeatable workflow. The calibration process starts by measuring agreement by slice. It then involves inspecting disagreement patterns, identifying the missing evidence or wrong decision, updating the judge or rubric, and finally retesting the affected slice.

Human reviewers can improve judgment rubrics, refine disagreement thresholds, and update target sampling rules. They can also expose missing information that should be provided to the automated judges during future quality cycles. After making the changes, use targeted sampling to test the impacted failure modes, and random sampling to find any potential negative impact on the broader automated process. This cycle repeats as data, policies, judges, and application requirements evolve over time.

Therefore, the relationship between automation and human oversight is continuous rather than sequential. Automated judges provide scalable judgments. Human reviewers resolve targeted cases and audit automated decisions. These findings improve the automated process and change the criteria for human review in the next iterations. The following diagram shows these mechanisms working together:

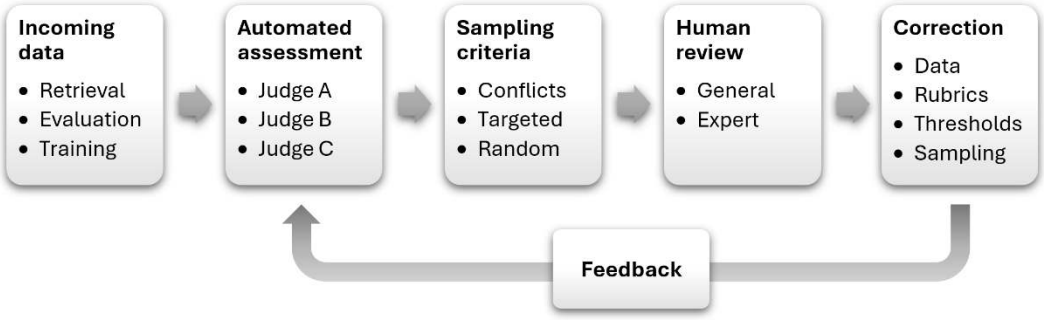


Figure 10.3: Scaling human oversight with automated judges, criteria-based routing and random sampling

Human oversight is a required component of data quality that scales through automated judges, disagreement routing, targeted routing, and random sampling. When the quality process operates on large datasets, the underlying data and its derived artifacts need a storage architecture that keeps data sources separate and accessible. The architecture must also make intermediate data reproducible. The next section focuses on storing LLM data.

Storing LLM data

You went over identification, correction, and review of data quality problems. These processes depend on fundamental capabilities to access the original data and reproduce the derived artifacts. For example, to fully correct a bad chunk from retrieval data, you need to trace it back to its original document and reprocess the document into a new index. Otherwise, it would be just a one-time patch.

LLM applications consume multiple forms of data, including documents, conversations, labeled examples, and production feedback. Data-processing pipelines transform these assets into normalized documents, chunks, embeddings, datasets, and indexes. These artifacts also have different storage requirements.

Most importantly, separate source data from derived data. Source data represents the core information you want to maintain. Derived data represents a particular transformation of this information for a specific use. Derived data such as parsed documents, embeddings, and vector indexes can be regenerated from original data or other derived data that traces back to the

original data. This distinction shows that original data needs to be preserved and maintained while derived data needs to be reproduced.

Some derived data is augmented with manual annotations and is therefore not fully reproducible from the original source data. Human-created annotations should be preserved as source data when they add information that cannot be reproduced automatically from the original data. For example, consider a production conversation that is transformed into a training example and then reviewed by a domain expert who labels the correct escalation decision. The formatted conversation remains derived because it can be regenerated from the original conversation, but the expert label is new information that cannot be regenerated automatically.

This gives us a practical boundary between the two categories. Preserve collected, acquired, and human-created information as source data and treat automatically reproducible transformations as derived data. The next section applies this distinction to the storage architecture.

Separating source and derived data

A useful storage architecture should preserve authoritative source data in a form that allows its derived artifacts to be regenerated. This storage approach is especially important for the correction workflows discussed earlier. For example, when document parsers or the chunking strategy change, you need the ability to reprocess the original documents again.

At small scale, local storage can be sufficient. Source documents, processed artifacts, training datasets, and evaluation datasets can live in files on a local or shared filesystem. A relational database such as SQLite or PostgreSQL stores structured metadata and processing state. This setup works well for experiments and applications with limited data volume and a small number of data-processing pipelines.

At a larger scale, cloud storage isolates the storage layer from the computation machines that process the data and make the data accessible from a wide variety of compute alternatives. Services such as Amazon S3, Azure Blob Storage, and Google Cloud Storage can store large collections of documents, JSON records, Parquet files, images, audio, and other artifacts. Processing jobs

read and write these objects without requiring the data to live on the same processing machines.

A data lake is a more advanced architecture built on top of cloud storage for organizing and processing large collections of heterogeneous data. Data lakes typically provide conventions for raw and processed data, dataset organization, metadata, access control, retention, and processing. A data lake is more valuable when multiple pipelines and applications consume the same collection of documents, conversations, structured records, audio, images, and other data.

A data lake is not a single storage technology. It is usually assembled from several components. Cloud storage provides the storage foundation. Formats such as Parquet provide efficient representations for large structured and semi-structured datasets. Processing engines such as Apache Spark transform and analyze the data. Data catalogs help teams organize, discover, and govern the available datasets. Cloud platforms provide catalog services such as AWS Glue Data Catalog, Microsoft Purview, and Google Dataplex, while platform-independent alternatives include DataHub and OpenMetadata. The exact stack depends on the existing data platform and operational requirements.

As datasets become larger and more structured, file-level management inside a data lake becomes more difficult. Consider a large collection of production interactions stored as Parquet files. Correcting records or applying incremental updates requires more than simply storing and reading separate files. The data pipeline must ensure that old and new data are not mixed and maintain a consistent dataset state. A data lakehouse adds a table-management layer over the files stored in the data lake. Technologies such as Apache Iceberg and Delta Lake maintain metadata and snapshots that support atomic updates, schema evolution, historical versions, and reproducible dataset states.

Table 10.3 compares these storage approaches and shows how each one fits in LLM data architecture:

Approach	What it is	Typical technologies	Best fit	Main limitation

Approach	What it is	Typical technologies	Best fit	Main limitation
Local storage	Files stored on local or shared disks	Local filesystem, network storage	Development, experiments and smaller datasets	Difficult to scale and share across distributed pipelines
Cloud storage	Durable and scalable storage for files and objects	Amazon S3, Azure Blob Storage, Google Cloud Storage	Large unstructured datasets and durable source storage	Does not provide dataset or table management
Data lake	Architecture for organizing and processing large datasets	Parquet, Spark, Glue, Purview, Dataplex, DataHub, OpenMetadata	Multiple large data sources and processing pipelines	File management is more difficult for frequently changing datasets
Data lakehouse	Data lake architecture with database-like table management	Apache Iceberg, Delta Lake	Large datasets requiring snapshots, schema updates, and reproducibility	Adds infrastructure and operational complexity
Relational database	Structured database for queries and transactions	PostgreSQL, SQLite, managed SQL databases	Metadata, application records, quality results, and moderate structured datasets	Non-ideal storage for large unstructured data
Vector database	Specialized storage and indexing for semantic retrieval	pgvector, Milvus, Weaviate, Pinecone	RAG and semantic search	Derived representation is not authoritative

Table 10.3: Comparing storage approaches for LLM data by scale and functionality

These storage approaches are complementary. A production LLM application might store authoritative documents in cloud storage and manage derived datasets using a data lake or a data lakehouse. The application might store operational metadata in a relational database. The important principle for data quality is for authoritative source data to be independently accessible, and for derived artifacts to be reproducible from them.

This distinction is especially important for semantic retrieval. Embeddings are derived from source documents through parsing, transformation, chunking, and embedding processes, and they are usually stored in vector databases. The next section visits the topic of managing vector indexes without mistakenly treating them as sources of truth.

Managing vector indexes

Vector indexes are one of the most important derived data stores in LLM applications. Documents are parsed, chunked, and converted into embeddings before they go into the vector index. Each of these steps can change independently from the authoritative source data. **A vector database is an index, not a source of truth**.

Chapters four and *five* covered vectorization, indexing, and retrieval in detail. Here, you focus on how to make the vector indexes consistent with the source data they were created from. When source data changes, the corresponding derived records in the vector index must change as well. Otherwise, the application may continue retrieving information that has already been corrected, replaced, or removed at the source.

Consider a human resources policy that has been replaced with a new version. Updating the document in source storage is not sufficient because chunks and embeddings from the previous version would still exist in the vector index. The assistant would therefore continue to answer based on the outdated policy. The update should propagate through the affected dependencies as follows: update source data, identify affected chunks, remove or supersede old vectors, regenerate chunks, generate new embeddings, and finally update the index.

Deleting source data requires a similar process. All chunks and embeddings derived from the deleted source must be identified and removed from the active index. Changing the embedding model might require generating new embeddings for the complete corpus and creating a new index, then evaluating and deploying it.

This process applies the dependency-aware correction approach introduced earlier in this chapter. A single document correction should require rebuilding only the related vectors, while a change to the embedding model may affect the whole index.

To maintain this relationship, you need stable identifiers between source and derived data. Each indexed record should retain enough information to identify the source document that produced it. At minimum, the metadata should include a source identifier, source version, and chunk identifier. It can also include effective dates, authority status, document type, region, tenant, and access-control information. For example, here is an indexed chunk with example metadata:

```
source_id: hr_travel_policy
source_version: 7
chunk_id: 12
effective_date: 2026-07-01
authority: approved
```

When version 8 of the travel policy is published, the pipeline can locate the chunks derived from version 7, remove or deactivate them, and generate the replacement records. Without this relationship, correcting a document requires a costly search inside the vector database.

Metadata also serves the retrieval requirements because enterprise applications usually need to restrict semantic search by properties such as tenant, document type, authorization scope, and region. For example, the HR policy of another country may be semantically similar to the policy of the correct country, but it should not be retrieved at all. Metadata filtering allows the application to combine semantic similarity with application-specific constraints.

Another core requirement is reproducibility. If a vector index becomes corrupt or needs migration to a different technology, the development team should be

able to rebuild it from the source data and known processing configuration. This requires preserving the source data as well as the relevant parsing, chunking, and embedding versions and configurations. Rebuilding does not mean that every update requires a complete index rebuild. The important requirement is that the complete index can be reconstructed if needed, while routine changes update only the affected records.

From a data quality perspective, vector storage infrastructure needs to support the following capabilities:

- Metadata filtering to restrict retrieval using application-specific data properties
- Updates and deletions to correct or delete derived records when their sources change
- Source traceability to connect indexed records to the source data that produced them
- Incremental update to propagate source changes without rebuilding unrelated data
- Full rebuild to fully reconstruct the index from its sources and processing configurations
- Access-control metadata for authorization to decide which information to retrieve

These requirements describe the intended behavior of the vector index, but they do not dictate the technology to use. The appropriate technology depends on data scale, update frequency, filtering requirements, existing infrastructure, and operational constraints. The next section uses these requirements to select an appropriate data infrastructure.

Selecting data infrastructure

No single data stack serves all LLM applications. Infrastructure decisions should consider the existing data environment and introduce new technologies only when they solve concrete problems. **The right infrastructure is the simplest one** that satisfies the application requirements in terms of scale, data management, retrieval, and operational cost.

For experiments and small applications, local or shared files are usually sufficient for source data. SQLite or PostgreSQL can manage metadata and

structured records. A local vector index or PostgreSQL with pgvector can support semantic retrieval. This setup keeps the number of infrastructure components small and makes debugging straightforward.

As the application moves into production, durable storage for source data becomes more important. Cloud storage such as Amazon S3, Azure Blob Storage, and Google Cloud Storage provides a common location for source documents and large derived artifacts independent of the machines that process them. A managed relational database can store metadata, quality results, and application state. For semantic retrieval, pgvector remains a practical option if the application already uses PostgreSQL and the vector workload fits comfortably within the same infrastructure.

Dedicated vector infrastructure such as Pinecone, Weaviate, and Milvus becomes useful for large vector collections, high query volume, complex metadata filtering, distributed deployments, or the need to isolate retrieval workloads. The decision should follow concrete retrieval requirements rather than uninformed assumptions.

In big organizations, the storage architecture could be driven by the existing data platform. A data lake is useful when multiple pipelines and applications consume large collections of heterogeneous data. Processing engines such as Spark transform data collections, and catalog technologies help with managing the available datasets. If the organization already runs this infrastructure, the LLM pipelines should normally reuse it.

Data lakehouse technologies such as Apache Iceberg or Delta Lake are useful when large structured or semi-structured datasets require stronger table management. Frequent corrections, incremental updates, schema changes, and the need for historical snapshots are good signals to move into data lakehouses. A lakehouse should be used when these data-management requirements exist, not simply because the dataset is large.

The same principle applies to lineage platforms. A small team may use a combination of PostgreSQL, dataset manifests, and naming conventions to track dataset versions and relationships. As the number of datasets and data-processing pipelines increases, machine learning tools such as MLflow and data version control (DVC) provide additional tracking for associating datasets with experiments, evaluations, and model artifacts. These systems

are valuable when manual conventions are not providing sufficient traceability.

The data catalog choice may follow the existing data platform. Organizations centered on a major cloud platform may already use AWS Glue Data Catalog, Microsoft Purview, or Google Dataplex. DataHub and OpenMetadata provide platform-independent alternatives if the organization uses multiple platforms or prefers a separate catalog technology.

As discussed, infrastructure decisions should be driven by the requirements rather than the application size alone. *Table 10.4* shows the additional requirements that motivate additional specialized infrastructure:

Capability	Basic technology	Additional requirements	Specialized infrastructure
Source data storage	Local or shared filesystem	Durable remote access, multiple compute systems, larger data volumes	Amazon S3, Azure Blob Storage, Google Cloud Storage
Large-scale data organization and processing	Cloud storage	Multiple pipelines need shared organization, distributed processing, cataloging, and governance	Data lake using Parquet, Spark, and data catalog technologies
Table management for large datasets	Data lake	Record updates and deletes, schema evolution, snapshots, and incremental processing	Lakehouse using Apache Iceberg or Delta Lake
Structured metadata	SQLite or PostgreSQL	Higher availability, concurrency, or scale	Managed or distributed relational database

Capability	Basic technology	Additional requirements	Specialized infrastructure
Semantic retrieval	pgvector or existing retrieval infrastructure	Larger vector collections, higher query load, advanced filtering, workload isolation, or distributed operation	Dedicated vector infrastructure such as Pinecone, Weaviate, or Milvus
Dataset and artifact tracking	Dataset manifests and naming conventions	Dataset, evaluation, and model relationships are difficult to manage	MLflow, DVC, or equivalent version management tooling
Data discovery and lineage	Database metadata and documented conventions	Many datasets, pipelines, and consumers complicate manual discovery and lineage tracking	Glue, Purview, Dataplex, DataHub, OpenMetadata, or existing enterprise data catalog

Table 10.4: Scaling LLM data infrastructure based on application requirements

These choices are independent rather than stages of maturity. An application may need cloud storage without a data lake, or a dedicated vector infrastructure while its metadata is stored in PostgreSQL. Start with the basic technologies that satisfy the core requirements and add specialized infrastructure only for concrete limitations.

The technology decisions should also be influenced by the cost and operational ownership. Every additional storage system introduces deployment, security, access-control, monitoring, backup, and maintenance requirements. Managed services reduce some of this operational burden but introduce service cost and platform dependency. Self-managed systems provide more control but add more responsibility to the development or infrastructure teams. A technology that improves performance may still be the wrong choice if the team cannot operate it.

Finally, if an organization already has a well-governed data infrastructure, reusing it usually provides more value than introducing another technology

with overlapping capabilities. Existing access controls, monitoring, ownership, and operational experience are part of the infrastructure decision. The best data technology is not necessarily the most capable one. It is the one that satisfies the requirements without adding unnecessary operational complexity.

Regardless of the implementation, the architecture should preserve the principles established throughout this section. Authoritative source data must be independently accessible. Derived data must be traceable and reproducible. Vector indexes must be synchronized with their sources. Storage systems are integrated because their capabilities address concrete application requirements.

Choosing the right infrastructure gives you the foundational ability to store and rebuild data, but infrastructure alone does not ensure quality. You still need controls that filter the data, track its source and how it changed, and detect quality degradation over time. The next section focuses on governing and monitoring data quality.

Governing and monitoring data quality

The previous section established the storage foundation for source and derived data. Storage alone does not guarantee that data that is valid today remains valid for the future, especially when data sources and application requirements change.

To achieve good data quality, you need additional controls across the complete data life cycle. In this section, you discuss four distinct quality controls:

- Quality gates prevent known problems from propagating through data storage and pipelines
- Metadata and provenance enable data failure tracing back to its sources
- Versioning identifies the state of the data used by the application and processing components
- Monitoring detects when data quality assumptions or criteria change over time

Together, these mechanisms keep data quality operational after the initial data preparation is complete. You can start by discussing the quality gates.

Enforcing data quality gates

You have already learned that data quality criteria must be measurable and actionable. This measurability is exactly what makes a good quality gate that controls whether a piece of data is allowed to go to downstream consumers. Consider an evaluation dataset where you need every reference answer to trace back to an authoritative source. One good quality gate is to check whether **`source_id`** exists for every answer and prevent records without a valid source from being added to the published evaluation dataset.

A similar quality gate applies to retrieval data. Deprecated policy documents should not be part of the retrieval index. Identify documents that are marked as deprecated through metadata or parsing and prevent them from getting added to the active vector index.

Quality gates operate at several control points in a typical data pipeline. They filter newly ingested source data before processing, confirm valid derived data after transformations, or validate a complete dataset before publishing it for retrieval, training, or evaluation. The right placement of a quality gate is simply as early as possible in the pipeline once the information required for validation is available. It is also okay to have some overlapping logic between the quality gates if it helps with detecting the problems and isolating them as quickly as possible.

Some quality gates are deterministic and can be computed directly from the data or metadata. Common examples include required fields (for example, source identifiers, effective dates, labels), schema validation, field validation, exact duplicates, and explicit deprecation signals.

Other quality requirements are semantically oriented and require judgment rather than simple validation rules. Examples include checking whether a reference answer is supported by authoritative evidence, checking whether a question can be answered from the associated context, and detecting contradictory or near-duplicate documents.

The semantic checks are directly connected to the LLM judges and human oversight workflows discussed earlier. Automated judges perform semantic

quality checks at scale when sufficient evidence and clear criteria are available. Human reviewers resolve ambiguous and low-confidence cases that automated judges cannot process reliably and audit automated decisions through sampling. This introduces new decisions a quality gate can make in addition to passing and failing. A quality gate can escalate to human review or quarantine malformed data if it is recoverable from its source.

Quality gates are not a universal checklist that applies to all datasets. A good gate should enforce the measurable criteria established for the application and the known failure modes that matter for that dataset. A quality gate is operational when passing or failing it changes what happens to the data. Passing the gate should indicate that the data satisfies the required checks for the next processing step. Failing the gate should take you to diagnosing the failure. To diagnose a failure, you need to know where that data came from and what happened to it.

Tracking metadata and provenance

Diagnosing a failure depends on your ability to move backward from a failed example to the data that influenced it. This process would be difficult if you did not have the relationship between the derived artifact and its source data. Provenance records the source of data artifacts and how they were produced. Metadata is a collection of attributes that represent these relationships.

Consider a document used by an enterprise RAG application. The data-processing starts from the source document and produces a parsed document, a normalized document, multiple chunks, and finally embeddings that go into the vector index. The source would carry metadata such as its identifier and owner. The parsed artifact would record the parser that produced it. Chunks would retain their parent document identifiers. The vector record would retain the chunk identifier and embedding configuration. This information records the dependency chain in an explicit way that the system can inspect.

The exact metadata schema differs between applications. Common fields include the source identifier, source system, source authority, source version, ingestion timestamp, parent artifact, transformation configuration, and review status. Artifacts that are late in the dependency chain do not need to include every metadata field. The requirement that matters is that the

relationships can be reconstructed whether directly or indirectly through multiple hops or a central metadata database.

The same provenance requirements apply to training and evaluation data. An evaluation question may be generated from a source document, then edited by a reviewer and assigned a reference answer. A training example may originate from a production failure, then get corrected by a domain expert and added to a curated training dataset. Preserving these relationships helps the development team understand the story behind the example and whether it is still valid after source changes.

Provenance is especially valuable during data correction processes. When a source document is deprecated, the system can identify the datasets, chunks, and index records derived from it. Provenance enables targeted root-cause analysis and dependency-aware correction.

Provenance tracks data artifacts across transformations, while versioning identifies the specific state of each artifact in that chain. The next section focuses on data versioning.

Data versioning

Data changes constantly for many reasons, including document updates, label corrections, transformation logic changes, and more. If these changes overwrite the previous state and you do not track the different versions, it becomes very difficult to reproduce an earlier state. Versioning tells the state of each data artifact at the time it was used.

Consider this dependency chain: policy v4 → parser v2 → corpus v7 → embedding configuration v3 → index v12 → application v18. This chain provides enough information to reconstruct the data state behind version 18 of the application. If a regression happens in version 19 of the application, the team can compare the data between versions 18 and 19, not whatever data exists in the latest version.

You should version the artifacts and configurations that materially change the application behavior based on the underlying system. These artifacts may include source data, transformation and parsing configurations, training and evaluation datasets, labeling guidelines, embedding model and configurations, vector index, model version, and application release version.

The version identifier itself needs to be immutable. For example, `evaluation_dataset_v12` is useful only if it always refers to the same set of examples and labels. If any of the evaluation records are silently modified while using the same identifier, the version identifier becomes more misleading than useful.

In addition to tracing and debugging changes, versioning also supports rollback. Assume a new version of the retrieval corpus passes offline evaluation but causes regressions after deployment. If the previous corpus and vector index are stored and versioned, the application can return to the last known good version while the new failure is being investigated. Without versioning, rollback would be a high-risk manual process.

Regarding versioning infrastructure, versions can be represented in several ways. As discussed earlier, smaller systems may use immutable files, manifests, hashes, and naming conventions. Larger workflows may use versioning and artifact management tools such as DVC or MLflow. The implementation details can change as long as you can identify and reconstruct the relevant states.

Versioning and provenance together provide traceability and reproducibility. To complement them, monitoring detects when the current data does not satisfy the production requirements at any point in time.

Monitoring data quality

When a dataset passes quality gates, it does not mean it will remain high-quality forever, because some quality aspects change without modifying the dataset itself. For example, a workflow becomes stale when replaced with a newer version. An evaluation dataset is not representative anymore when production traffic changes or a new category of users is onboarded. A previously reliable parser can fail when new document formats and capabilities are introduced. Monitoring verifies that the assumptions behind quality gates remain valid over time.

It is useful to separate monitoring into two layers of pipeline health and data quality health. Pipeline health monitors the techniques that collect and process data. Some of the useful signals to monitor include ingestion failures, parsing errors, missing metadata, failed jobs, unexpected changes in data

volumes, and index synchronization failures. These failures are usually operational and relatively straightforward to detect.

Pipeline health thresholds are usually expressed as error rates over a predefined processing window. Some cloud data platforms use data-correctness objectives in the low single-digit range, such as less than 2-3% bad records for batch and streaming workloads (Google Cloud, 2026). The data pipeline defines the correctness checks, and records that fail those checks are classified as bad.

This threshold is a practical baseline rather than an industry standard. More stable and critical workloads require much tighter error thresholds, while experimental and less critical pipelines may tolerate higher rates to avoid excessive false alerts. The appropriate threshold should become stricter as the impact of incorrect data increases.

Some requirements should allow zero tolerance, such as the presence of access-control metadata, because even one affected record can expose risky information with severe impact. For other requirements, such as freshness, express the requirement as a latency target: require daily data to be ingested by a defined daily time or set a target window from publication time to ingestion time.

Data quality health monitors the data against application quality requirements. Useful signals include data freshness, category distributions, query and intent distributions, retrieval patterns, scenario coverage, evaluation scores, labeling agreement rates, and divergence between evaluation and production distributions. These signals can change even when every data pipeline works as expected.

Data quality metrics that naturally fluctuate are better monitored against a known-good baseline. An increase of 2-4 times the normal baseline can be used as an alert for triggering investigation. For example, if parsing failures are normally below 0.5%, an increase to 1-2% deserves investigation even if the application did not define a universal threshold for parsing failures. The baselines should be calculated based on a representative operating period and should be calculated separately for important slices.

Consider an evaluation dataset that accurately represented production traffic when it was created. Over time, users started asking about newly launched

products. The evaluation dataset itself did not get corrupted. Its examples remain correct, and all required metadata is present. However, its coverage is no longer matching the production distribution. A quality gate that passed a few months earlier cannot detect this change unless production distributions are monitored.

Measuring distribution changes requires a statistical distance measure rather than a simple percentage. Metrics such as *Jensen-Shannon* divergence can compare the current distribution with a known-good baseline. Some production monitoring systems use configurable thresholds in the 0.1–0.3 range as baseline starting points (Google Cloud, 2026). These values do not mean 10–30% drift. They represent distances defined by the selected statistical metric. You should calibrate the threshold by measuring the distance during known-good operating periods and comparing it with periods where the distribution change is large enough to affect application quality.

Monitoring should also work on data slices that are meaningful to the application in addition to aggregate metrics. For example, an overall correctness score may remain stable but drop significantly for specific languages, user categories, or intents. The same principle applies to data distribution monitoring, as aggregate volume may look normal while an important slice trends unhealthy.

The appropriate threshold depends on what is being monitored. *Table 10.5* shows practical suggested approaches for the main monitoring signals discussed in this section.

Monitoring signal	Threshold approach	Initial suggested threshold	Example
Pipeline data errors	Error rate over a defined window	Low single-digit rates such as 2–3%; tighten for critical pipelines	Malformed or invalid records
Critical integrity requirements	Zero tolerance	0 violations when failure is not acceptable	Missing access-control metadata

Monitoring signal	Threshold approach	Initial suggested threshold	Example
Data freshness	Time-based requirement	Processing deadline based on data source (minutes to hours)	Daily updates feed
Variable quality metrics	Deviation from baseline	Around 2-4X of the normal metric	Parsing failures
Distribution shift	Statistical distance from baseline	Around 0.1-0.3, assuming Jensen-Shannon divergence	Query distribution

Table 10.5: Practical guidance for data quality monitoring thresholds

These values are practical starting points, not universal data quality standards. Final thresholds should be calibrated using a known-good production baseline, acceptable risk levels, expected variations, data volume, and the cost of false alerts versus missed failures. High-risk requirements should use tighter thresholds, including zero tolerance where appropriate. Thresholds should also be revisited when data sources or application requirements change.

Monitoring also complements and extends evaluation into production. When an evaluation metric drops, monitoring may provide additional evidence that helps with understanding and explaining the regression. A drop in retrieval quality combined with a sudden shift in user intent distribution points the investigation in a different direction than the same drop in retrieval without the intent distribution change.

Beyond reports and dashboards, monitoring should also lead to actions, sometimes even in an automated way. For example, a metric drop under a certain threshold should trigger the appropriate workflow, such as targeted sampling for human review. This correction path turns monitoring into an effective quality management tool.

The following diagram shows quality gates, provenance, versioning, and monitoring working together to maintain data quality throughout the data quality governance life cycle:

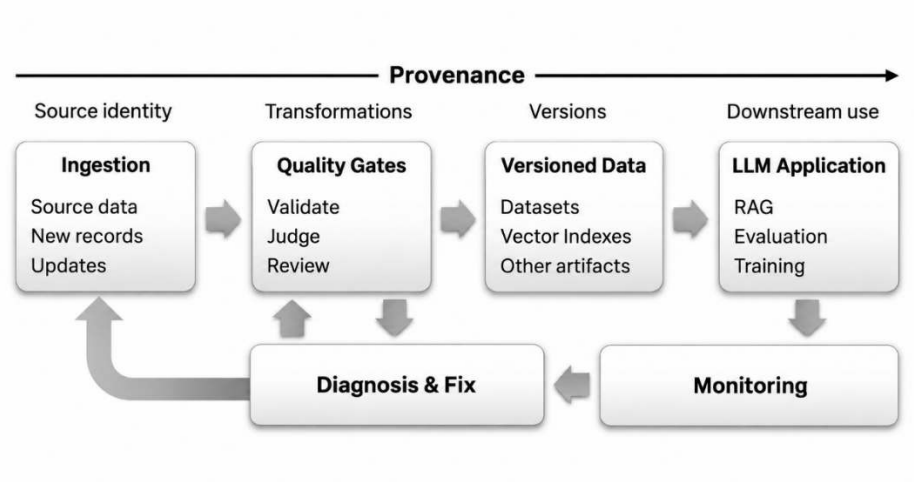


Figure 10.4: Data quality governance life cycle

In this section, we discussed many quality management mechanisms including quality gates, metadata, provenance, versioning, and monitoring. These individual mechanisms are necessary to maintain data quality in production. The final step for this chapter is to connect evaluation, diagnosis, human oversight, correction, storage, governance, and monitoring into one continuous workflow. The next section closes the data quality loop.

Closing the data quality loop

So far, we have examined the individual mechanisms required to maintain data quality. We learned how evaluation signals identify potential data problems, how failures can be traced to their root causes, how human oversight works with LLM judges, and how data governance supports these processes in production. These mechanisms provide the most value when they operate together.

Data quality is a continuous loop, not a sequence that ends at deployment. Evaluation identifies failures. Diagnosis finds the responsible data. Corrections update the responsible data and its dependent artifacts. Evaluation verifies the correction before deployment, and monitoring detects new failures after deployment that start the process again.

Consider a hypothetical scenario using the enterprise assistant example introduced earlier in the book, and assume the assistant is already serving production traffic. Over time, monitoring shows that correctness and faithfulness declined for questions related to employee travel.

The aggregate scores indicate that something has changed, but they do not identify the cause. Looking at the affected evaluation and production examples shows that the assistant provides two different reimbursement limits for similar cases. This identifies a problem in the retrieved data that contributes to these responses.

The development team inspects the retrieved chunks for the failed examples. Some chunks come from the current travel policy, while others come from an older version that still exists in the vector index. Both documents are semantically relevant to the questions, so retrieval similarity cannot identify the one that should be used.

Tracing the outdated chunks backward shows that the old policy was never marked as deprecated when the new version was published. Therefore, the ingestion pipeline processed both documents and considered both valid. The vector index exposed the failure, but the root cause was upstream in the source metadata. Additionally, the ingestion pipeline had a wrong configuration that allowed deprecated documents.

The correction starts at that root cause: marking the old policy as deprecated. Because the authority decision affects business policy, a human reviewer validates the correction before it is propagated. Then, the data pipeline identifies the chunks derived from the old policy and removes them from the active retrieval index.

The correction does not stop here because the same records may return to the index during the next index rebuild because of the wrong ingestion pipeline configuration. The quality gate must be updated as well. The new gate prevents deprecated documents from entering the active retrieval corpus. These fixes convert one observed production failure into a reusable control that prevents similar future failures from happening.

The affected derived artifacts are rebuilt, and the pipeline produces the active document collection, chunks, embeddings, and updated vector index. The new corpus and index receive new version identifiers so the application can be evaluated against a reproducible data state.

Before deployment, the evaluation suite from *Chapter 9* runs against the updated application. Regression checks verify that retrieval and answer quality did not regress for other policy areas. Human sampling checks that the

automated quality process is working as expected. If the updated application satisfies the acceptance criteria, the new data and index versions are deployed.

Production monitoring watches the same quality dimensions that exposed the original problem on overall traffic and other relevant slices. If the failure persists or another failure shows up, the loop continues.

The importance of the complete loop becomes obvious when one of its components is missing or faulty. Assume the same travel-policy correction is deployed without versioning. Production monitoring would later detect a new regression, but the team cannot determine which corpus and vector index were served before the correction or restore the last known good state. Similarly, without monitoring, the corrected application may pass evaluation at deployment time but a policy change that happens later would silently corrupt the retrieval data. *An effective data quality loop requires all four capabilities: detecting failures, tracing changes, validating corrections, and restoring known-good states when needed.*

This example illustrates a general workflow that applies beyond RAG. A training regression may trace back to wrong labels and result in corrected training data and a new model version. An evaluation failure may expose a wrong reference and lead to a corrected evaluation dataset. The artifacts could be different between failure cases, but the data quality loop is the same. The following diagram summarizes this continuous process:

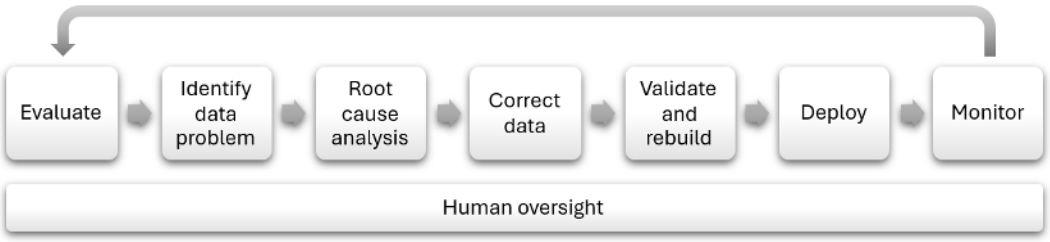


Figure 10.5: Continuous data quality loop

The quality loop has two properties that are important in production. First, corrections should create repeatable controls whenever possible. Fixing one old document solves one example, but adding a quality gate that prevents old documents solves the class of problems that produced the example. Human findings can similarly improve automated judges, review rubrics, metadata

requirements, and monitoring rules. The objective is for each correction to improve the quality system, not only the affected record.

Second, every correction must be followed by evaluation. A change that is obviously correct can introduce a regression somewhere else. For example, removing conflicting documents can reduce coverage. Rebalancing training data can improve one behavior and degrade another. A correction is complete only when the updated system satisfies the application requirements again.

This distinction is also why monitoring and evaluation serve different but complementary roles. Monitoring observes the production environment continuously and detects changes in data, distributions, and application behavior. Evaluation provides controlled measurement against defined requirements and determines whether a corrected application is ready to deploy. Production monitoring eventually generates new signals. Evaluation provides a structured mechanism for measuring these signals and deciding their impact on application quality.

Human oversight is present throughout the data quality loop. Automation can do a lot of steps such as identifying failures, tracing dependencies, running quality checks, rebuilding artifacts, and evaluating large volumes of examples. Humans audit these automated steps, resolve ambiguous cases, provide domain expert judgment when required, and calibrate the quality process as new failures show up. The goal is to minimize unnecessary manual effort without removing human accountability from important quality decisions.

Data quality does not have a final state. There are always changes in the data, requirements, users and techniques. A production data strategy must be designed around continuous change. The goal is not to build a dataset that remains correct forever. Instead, build a system that detects when its data is no longer good enough and corrects it safely.

Summary

In this chapter, you learned how to move from detecting data quality problems to correcting them and preventing them from returning. Building on the evaluation signals from the previous chapter, we traced failures back through the data pipeline, identified their root causes, corrected the affected data, and rebuilt dependent artifacts. You also learned how automated judges,

targeted review, and random sampling make human oversight practical at production scale.

The first takeaway is that data quality matters more than volume at the application layer. The application defines quality by how it uses the data. Retrieval, training, and evaluation data have different quality requirements. Adding more data does not improve an application when that data has become obsolete, contradictory, wrongly labeled, or does not represent the target scenarios.

The second takeaway is that identifying bad data is not enough. Data quality requires an operational system around the data. Root cause analysis identifies the source of a failure. Correction pipelines fix the earliest faulty artifact and rebuild its affected dependencies. Human oversight validates automated decisions and resolves ambiguity. Quality gates prevent known failures from propagating, while provenance and versioning make the data changes traceable, reproducible, and reversible.

Finally, data quality does not end when you publish the application. Data sources change, the user distribution shifts over time, and application requirements evolve. Monitoring detects these changes, and evaluation decides whether the application satisfies its requirements. Monitoring and evaluation together feed new failures back into the continuous data quality loop.

These practices enable development teams to maintain trustworthy data without manually inspecting and correcting every record. The goal is to build a system that identifies when its data is no longer good enough and corrects it safely.

In the next chapter, you move from maintaining the quality of individual data assets to organizing this knowledge and supplying it at runtime, also known as context engineering. You will examine how to represent facts, rules, and entities such as people and systems in a way that supports LLM applications.

References

- Breck, E., Cai, S., Nielsen, E., Salib, M., & Sculley, D. (2017). *The ML test score: A rubric for ML production readiness and technical debt reduction*. IEEE

International Conference on Big Data. https://ieeexplore.ieee.org/document/8258038

- Dropbox. (2026). *Engineering VP Josh Clemm on how we use knowledge graphs, MCP, and DSPy in Dash. Dropbox Tech*. https://dropbox.tech/machine-learning/vp-josh-clemm-knowledge-graphs-mcp-and-dspy-dash
- Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H., Daumé III, H., & Crawford, K. (2021). *Datasheets for datasets. Communications of the ACM*, 64(12), 86-92. https://dl.acm.org/doi/10.1145/3458723
- Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., & Gebru, T. (2019). Model cards for model reporting. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229. https://dl.acm.org/doi/10.1145/3287560.3287596
- Polyzotis, N., Roy, S., Whang, S. E., & Zinkevich, M. (2017). Data management challenges in production machine learning. *Proceedings of the 2017 ACM International Conference on Management of Data*, 1723-1726. https://dl.acm.org/doi/10.1145/3035918.3054782
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 28. https://dl.acm.org/doi/10.5555/2969442.2969519
- Jung, J., Brahman, F., & Choi, Y. (2025). Trust or escalate: LLM judges with provable guarantees for human agreement. *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2407.18370
- Klie, J.-C., Eckart de Castilho, R., & Gurevych, I. (2024). Analyzing dataset annotation quality management in the wild. *Computational Linguistics, 50(3)*, 817-866. https://aclanthology.org/2024.cl-3.1/
- Apache Iceberg. (2026). *Apache Iceberg documentation*. Apache Software Foundation. https://iceberg.apache.org/docs/latest
- Delta Lake. (2026). *Delta Lake documentation*. Linux Foundation. https://docs.delta.io

- MLflow. (2026). *ML dataset tracking*. MLflow. https://www.mlflow.org/docs/latest/ml/dataset
- Google Cloud. (2026). *Plan your Dataflow pipeline*. Google Cloud Documentation. https://cloud.google.com/dataflow/docs/guides/plan-pipelines
- Google Cloud. (2026). *Introduction to Model Monitoring*. Google Cloud Documentation. https://cloud.google.com/vertex-ai/docs/model-monitoring/overview

11

Context Engineering: Data Strategy at Agent Runtime

I started *Chapter 1* by explaining the idea of in-context learning, where we feed the context to the LLM to help drive better answers. We showed through multiple examples how the answer will differ by adding the context. Then, we built on this idea in the following chapters, where we crawled sources for the raw data in *Chapter 2, Data Sources*. In *Chapter 3*, we indexed the data into a vector database (DB) to retrieve it in *Chapter 4* using the similarity search. The challenge has been how to put the right context in front of the LLM to give the right answer without hallucination, and without drowning the model in too much context.

One rule ended up underneath every choice in this chapter, and it took me longer than I want to admit to say it plainly: derive everything you can defend, and refuse to fabricate the rest. The easiest way to explain this rule is to think of a new colleague joining your project. Some of what they need is already in the repository: which files exist, what calls what, who owns which part. In the case of automation, a tool can pull that out too and stay honest, because every entry points back to a file you can check and defend. The reasons behind the choices are different. Why the team picked this design, or which decision it reversed last month, was said in a meeting and never written into a file. A tool that answers those questions anyway is guessing and fabricating, and its guess looks just as confident as everything else it writes. So either someone writes the reason down, or the gap stays visible with a name on it until someone does. Refusing to fabricate means leaving that gap visible until a person fills it. An empty file with an honest name is a refusal; a plausible paragraph nobody decided is a fabrication.

I did not arrive at that rule in the abstract. I walked into it. It is worth telling you the path, because the path is the chapter.

It started with retrieval. We built agents that used retrieval-augmented generation (RAG) to pull in what they needed, the way I described earlier in the book, and the way the CoALA framework frames agent memory in general [1, 19]. I thought retrieval was the answer to everything.

Then I hit a wall that had nothing to do with model quality. I was working across roughly 1,200 APIs. Carrying a full RAG pipeline into every project I touched, or updating the central RAG with every code change, turned out to be heavy and fragile. Every new project meant standing up the same machinery again. So I built two lighter, portable things I could drop into any repository: a knowledge base and an index in plain markdown files that let an assistant find what it needed without a vector database behind it, an approach I later wrote up as a markdown knowledge base for engineering teams [2]. I called it a knowledge graph at the time, and I will come back to that choice of name because it caused me some grief. The industry later gave the general idea a name of its own: vectorless RAG.

The markdown knowledge base was the brain for the repository. A repository crawler then studied the code and kept that knowledge current. The index was the map for the repository. The problem that little index solved was narrow and practical: looking things up in markdown files fast, without the overhead of embeddings and a retrieval pipeline. When coding assistants arrived, that same index stopped being a convenience and became a requirement. An assistant that can touch your files needs a map, or it wanders. From there, I built what I call Agentic Repos. The repository carries what the assistant needs to know about the project, and what the team learns while working, the decisions and the lessons, is written back so the next session starts with it [9, 10, 16]. That is how I grow an agent from junior to senior in a domain, and I describe the full framework in my article "Growing AI Agents from Junior to Senior" [10].

That success is what created the problem this chapter is actually about. First, you grow the knowledge base. As the knowledge grows, choosing the right slice to hand the model at the right moment becomes the hard part. Not too little, or the assistant guesses. Not too much, or you burn the window and

confuse it. That discipline, deciding what to put in front of the model and what to leave out, is what the industry started calling context engineering [12, 13], and it is where this whole chapter is heading.

I had been running this pattern for a while before it had a public name, and the public version arrived after mine instead of before it. In April 2026, Andrej Karpathy described the same shape from the other direction: rather than retrieve chunks on every question, compile the raw folder once into an interlinked markdown wiki the model can navigate on its own, in a gist that sketched the layout [20]. Tooling arrived almost immediately. Graphify runs that compile step across an entire codebase and hands back a queryable graph out of the box; the linked note vault and the index file to start from are opt-in outputs in the release I verified in August 2026 [21, 22, 29]. I want to be precise about what that means for this chapter rather than wave at it politely. They have solved the half a parser can solve, and they have solved it better than I am going to: it reads the files, writes the map, and an optional commit hook keeps the code side of the map in step from then on. What a parser cannot write is the half that was never in the files to begin with. Hold onto that split, because it is the spine of everything after this.

Which is also the useful thing to say about this chapter's shelf life: the tools that build maps will keep arriving and keep improving, and you should use them. Learn the part they cannot do. This chapter will walk you through the intuition with simple examples to understand the concept as we have been doing throughout this book. One community-maintained directory of these tools, packaged as skills, is `https://awesome-skills.com/`, which listed more than 150 of them when I checked in September 2026. Directories like this change quickly, so treat it as a starting point rather than a permanent reference. This book will show you the basics it all started from so that you can select the right one.

The example I am about to build is small. Ten files, under two thousand tokens. At that size, the token savings I am going to show you are unimpressive. In one of the five test questions, the map actually costs slightly more than dumping the entire folder into the prompt. I could have hidden that by picking friendlier questions. I would rather tell you, because it points at the real lesson: below about a hundred documents, the map is not a performance optimization. It is what stops the assistant from being

confidently wrong. The token argument does arrive at scale, and I will show you roughly where the crossover sits, but I am not going to lean on it. That argument has been made better than a book example can make it, by tools I will point you at, and it was never the reason I built this. You are going to build up to it one problem at a time, on an example everyone understands, then move to an engineering repository.

In this chapter, you will cover the following topics:

- The junior new session
- Understanding the knowledge map
- Facts that expire
- Rules the agent follows
- Representing people and systems
- Growing junior to senior
- The discipline of context engineering
- Applying the pattern across domains
- Applying the pattern across the four use cases

Technical requirements

Open `Chapter_11_Financial_Agent_Notebook.ipynb` and keep it beside you. The chapter and the notebook follow the same progression in the same order, and you should run each cell as you reach it rather than reading first and running later, because two of the results are meant to surprise you and they land better live. The notebook relies on the setup from *Chapter 1*. If you have not done that yet, do it first. The setup creates the `.env` file with the API key the notebooks in this book use. This notebook reads the key from your environment and, if it is missing, skips the live model calls politely.

You can also run the whole thing **without an API key**. Every step that needs a model degrades to printing the context it would have sent, and the assembly, the scoring, the precedence, and the token accounting all still run. The mechanics are the lesson; the model is the demonstration.

One thing to know before you trust a number. The notebook's token counter is deliberately crude:

```
def tokens(text):
    # Rough token estimate: about 4 characters per token for English text.
    return len(text) // 4
```

Four characters per token, no tokenizer, no dependency. Every finance-demo token count in this chapter comes out of that function, so treat every number as a rough estimate rather than a measurement. I kept it this way on purpose: the ratios are what the chapter argues from, and a ratio survives a sloppy denominator as long as both sides use the same one. If you want exact counts, swap in your provider's tokenizer and everything else works unchanged.

Run the setup cell now (search for "# === Setup ===" in the notebook). It finds the folder and splits it into the three tiers, which is the whole design in five lines of Python:

```
all_md = sorted(str(p.relative_to(AGENT)) for p in AGENT.rglob('*.md'))
content_md = [f for f in all_md if f not in ('AGENT.md', 'INDEX.md')]

print('  T0 contract (AGENT.md):', 'AGENT.md' in all_md)
print('  T1 index (INDEX.md)   :', 'INDEX.md' in all_md)
print('  T2 content            :', len(content_md), 'files')
```

The setup cell will print:

```
Found the knowledge folder.
  T0 contract (AGENT.md): True
  T1 index (INDEX.md)   : True
  T2 content            : 8 files
      archive/2025_allocation.md
      decisions/2026-01-10_annual_review.md
      decisions/2026-03-14_529_vs_brokerage.md
      entities/advisor_okonkwo.md
      entities/employer_equity.md
      facts/accounts.md
      facts/allocation_2026.md
      rules/decision_rules.md
```

Two files are held out of `content_md`, and that exclusion is not housekeeping. `AGENT.md` and `INDEX.md` are the tiers you always pay for, so they never belong in the pile you are choosing from. Keeping them out of that list in the setup cell is what makes every later comparison honest.

The setup is done. Now, the problem the whole design exists to solve, and it starts the same way every session does.

The junior new session

Open a truly fresh session, a bare API call or a chat with the product's memory features off, and ask the assistant about something it helped you with yesterday. The assistant has no idea what you are talking about. Not because it is not capable, but because it starts every session as a junior. The assistant knows nothing about your situation, your history, or the rules you care about.

So, you do what we all do: you paste your accounts, your goals, last month's decisions, and the thing you explained last time. Every session, the same paste. If you are using a tool that can read your files, it does the same thing in its own way, quietly: it re-reads everything to rebuild the understanding it had an hour ago.

That re-reading is not free. I have paid for it enough times to name the cost, and the whole chapter is about getting these three back under control. It costs tokens, because you send the same material again and again. This re-reading costs time, because you sit and wait while the assistant re-derives what it already knew. It consumes space in the context window, which is the one that actually hurts. The window is finite. Every token you spend re-explaining your life is a token the model does not have left for the work. Fill the window with re-discovery, and the model has less room to think, right at the moment you need it thinking most.

The fix is not a smarter model. The model forgets between sessions on purpose. The fix is to give the work a place to keep its memory, so the next session starts senior instead of juniors. That place is a handful of plain files on disk, and the rest of this section builds the smallest version of it I could get away with.

I picked personal finance for the example because everyone has finances and nobody enjoys re-explaining their whole money situation from scratch. The persona is me, to keep it concrete, and every number is invented. None of it is my real financial data. The folder is real, though and you can follow along in it: `chapter_11/finance_agent`, in the book's GitHub repository. To access the repository link, follow the steps in the Download the example code files section in the Preface.

Now, the real question, in *Part 1* of the notebook (search for === *Part 1: The junior problem*). You ask the model with no context at all:

```
QUESTION = 'What is my current target allocation, and can I add more to my tech stock?'
answer = call_llm('', QUESTION)
```

One thing about `call_llm`, because it is a rule I keep for every notebook in this book: it does not pin requests to one fixed selected model. The code discovers the models your key can actually use, tries the preferred ones in order, and keeps the first that passes a one-token test. Hardcoding a model is how a technical book ages badly the month after it ships.

The prompt carries zero tokens of context. A bare, memory-free call sees exactly this. The model can only guess, or, if it is honest, tell you it does not have that information. The lack of context is not a model failure. It is the junior-every-session problem in a single cell. There is nowhere for it to have learned anything about me. Notice too that the call is bare on purpose: one system message, one user message, no tools, no retrieval framework, no coding assistant doing hidden work in the background. I want the problem to be visible, not tucked behind machinery. The exchange is short enough to show in full:

```
Question: What is my current target allocation, and can I add more to my tech stock?
Context sent: 0 tokens (the model knows nothing about the persona)

Junior answer (no context):
I do not have that information.
```

The obvious fix is the one everybody reaches for. Paste everything. That is *Part 2* (to find Part 2 in the notebook, search for === *Part 2: The context dump*):

```
blob = dump_everything()
print(
    'Context dump:', tokens(blob), 'tokens from',
    len(content_md), 'content files'
)
```

The context dump works, which is precisely why people stop here. However, watch two numbers. The first is size. The dump comes to 1,156 tokens across eight content files, for a folder small enough to read over a coffee. Scale that up to a real financial life, or a real codebase, and you have spent your window before the model has done a minute of thinking.

The second number is worse, and it is not a number; it is a contradiction. The archived file, `archive/2025_allocation.md`, says eighty percent equity and twenty percent fixed income. The current file, `facts/allocation_2026.md`, says sixty/forty. Both are sitting in the blob, and nothing in the pile enforces which one wins today. You handed it more context and less clarity.

So, you are stuck between two bad options. Send nothing, and the assistant is a junior who guesses. Send everything, and it is expensive and worse, nothing in it forces the current allocation to win over the one you walked away from last year. More context gave you less clarity, not more.

What you actually need is narrower than "more context." You need the right context: the two or three files that bear on the question, and none of the ones that do not. However, here is the catch that makes it interesting. How would an assistant know which files those are without reading all of them first? That is the question the next section answers.

Understanding the knowledge map

The junior new session left you stuck between guessing and drowning, and it left you holding a question that sounds like a paradox. You want the two or three files that bear on the question. To know which two or three, you have to know what is in the files. Knowing what is in the files means reading them. Reading all of them is precisely the thing that just failed.

That circle is the entire reason for what comes next. Something small has to be able to tell you what is inside something big. Accept that one sentence and the rest of this section stops being a design I am asking you to admire and becomes the only move left on the board.

You already trust the move, by the way. You have used it your whole life. Nobody reads a six-hundred-page reference manual to find the chapter on taxes. You read the two-page table of contents. The table of contents is not the book. It is a description of the book, small enough that you read it from top to bottom every time you pick the thing up, and specific enough that you know which page to turn to. The table of contents is the idea in full. Everything after this is bookkeeping.

Here is the version I find easier to hold on to, and it borrows the person from the last section. Say that junior joined you tomorrow and you had ten minutes

before a meeting. You would not photocopy your bank statements for them. You would write one page: balances are in this file, the rules I invest by are in this one, last year's plan is in the archive folder, and it is dead; do not follow it. That page is a knowledge map. You have written one before. You just did not call it that.

I owe you a note on the name, because I got it wrong first and someone rightly called me on it. I originally called this a knowledge graph [3, 4]. The pushback is the one you are probably already forming: this is not a graph database; there are no nodes and edges and no query engine. Fair. The more honest name is a knowledge map. And that gives us the distinction I will use for the rest of the chapter. The brain is the knowledge itself, the files. The map is what you read to find your way through the brain. Grow the brain, and you have more to know. Keep the map good, and you can actually find it.

So, you have a brain and a map. The next question is what the map itself should look like, and the first thing I learned is that it cannot be a single file.

Organizing into three tiers

The first version of this I built was a single file. One index, one list of entries, every entry carrying every field I had ever found useful. It worked for a while, and then it stopped, and it stopped for a reason I did not expect: the index itself got too big to read. I had built a table of contents that had grown into a book.

So, the design has three layers now, and the only interesting thing about them is what decides which layer a piece of knowledge goes in. The question is not what kind of thing it is. The question is when you can afford not to have it.

Tier zero is what you cannot afford to miss. One file, `AGENT.md`, read every single session, no exceptions, whatever the question. If you use a coding assistant, you already know this idea: it is the job that files such as `CLAUDE.md` and `AGENTS.md` do for a repository. I come back to that. It holds the hard rules, the order of precedence when two files disagree, and the instructions for how to navigate. It is deliberately short, because everything in it is paid for on every question you ever ask.

Tier one is the index, `INDEX.md`. One line per item. The index is the table of contents, and it is the thing you read to decide what to open.

Tier two is the knowledge itself, the actual files, opened only when the index points at them.

Now, the part that makes this work, and it is a filesystem trick rather than a clever data structure. The directory a file sits in carries two pieces of information at once:

```
finance_agent/
├── AGENT.md          tier 0, read every session
├── INDEX.md          tier 1, generated
├── rules/            how I decide
├── entities/         people and systems
├── facts/            what is true now
├── decisions/        what I chose, and why
└── archive/          what used to be true
```

The directory name tells you what kind of thing you are looking at. The order of those directories, top to bottom, is the order of precedence when two files disagree. Rules beat facts. Facts beat decision records. Everything beats the archive, which is history and never current truth.

I want to be plain about why that matters, because it is the difference between this version and the one I shipped first. My original entries carried a type: field and an authority: field. I removed them. Not because they were wrong, but because the folder structure was already saying the same thing, and a field that duplicates the structure is a field you will forget to update. The first time I found an entry marked authority: source-of-truth sitting in an archive folder, I stopped defending the field.

Position encodes both. Free, and the drift surface is small; you will see exactly how small it is when you get to the loader.

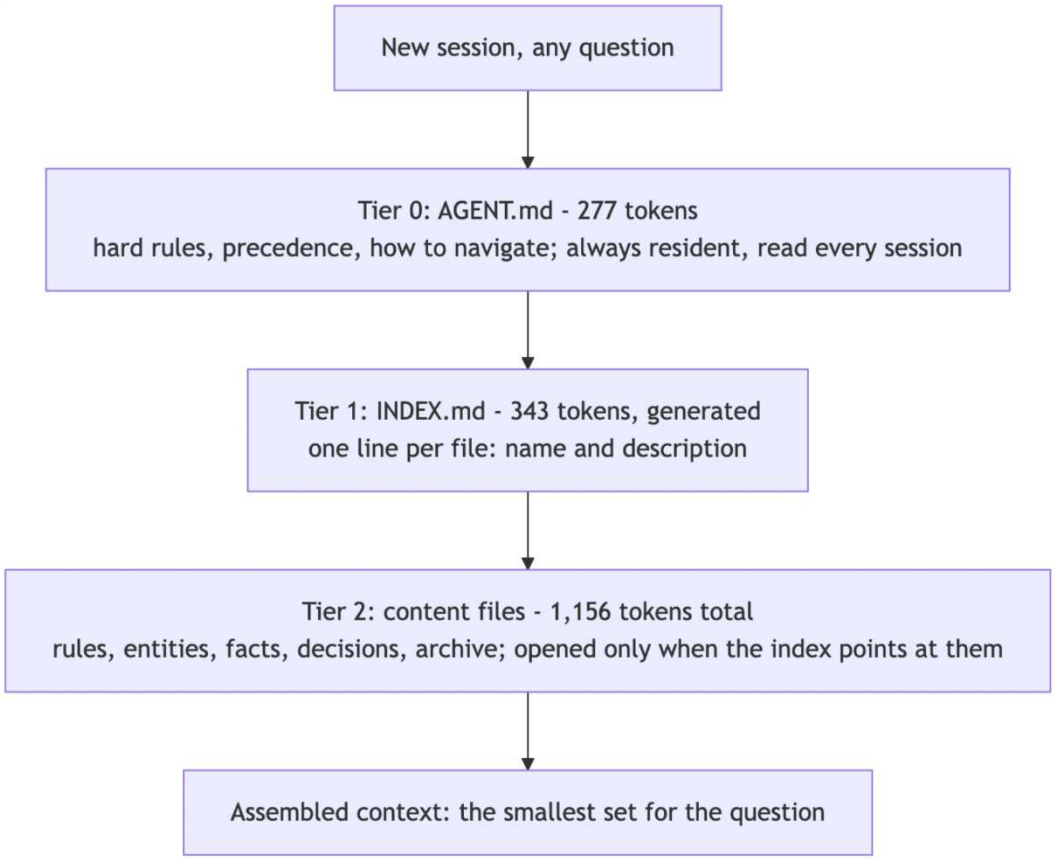


Figure 11.1: The three tiers, from a new session to the assembled context

Read Figure 11.1 top to bottom. Every session pays for AGENT.md before any question is asked, the index decides which content files are worth opening, and only those files get loaded, so what reaches the model is the smallest set that answers the question.

Two honest caveats before you copy those five names onto something real.

The first is that they are the shape for a folder one person owns. A team corpus has to answer a second question, and it is not what kind of thing this is. It is who said so. On the platform I run, the directory that matters most is called *source of truth*, and what it encodes is not type at all. It is permission. Those documents were agreed upon by people who had to agree, and changing one is not something you do on a Tuesday because you disagreed with it [2, 16]. Type and authority are different axes. My five folders answer the first one and say nothing about the second. Once a team shares the folder, that gap shows up within a week.

The second is that five names do not absorb a real corpus. Mine has sixty-six meeting notes. They are not rules, not entities, not current facts, not decisions, and not archive. They are evidence, which is a sixth kind of thing I did not need at eight files and needed badly at six hundred. When you reach that size, give it a directory of its own, and let each decision that rests on a meeting point back to the note. Trust the note less than the decision, though: a meeting note records what someone said at the time, not what is true now, which is how the wrong job title later in this chapter spread into dozens of files. So, take the idea, which is that position should carry meaning so that a field does not have to. Do not take my five names as the answer to a corpus you have not looked at yet.

The directories decide where a piece of knowledge lives and which one wins when two disagree. What they do not decide is what each line in the index actually says, and that one sentence is where most of the design turned out to hide.

Reading what a signpost has to say

An index line looks like this, and there is nothing behind it:

```
- [Target allocation (current)](facts/allocation_2026.md) - The 60/40 target adopted January 2026.
Supersedes the 2025 80/20 target.
```

A markdown link and a sentence. The format has two parts. It took me several bad versions to get down to that, so let me show you what each piece is for by asking what the line has to answer.

Which file are you talking about? The path, and only the path. My earlier design carried a separate ID: field, a short nickname so entries could point at each other. I liked it. I was wrong. The path is already unique, already meaningful, and already the thing you have to type to open the file. A second identifier is a second thing to keep in sync, and I never once needed the nickname for anything the path could not do.

What is inside it? The description is the one nearly everybody skips, and it is the field that earns the whole design. Look at the path `facts/allocation_2026.md` and tell me honestly whether that file holds a target, a history of targets, or a rationale. You cannot tell. Nobody could. A map

promises that you decide *without* opening the file, so a name that forces you to open the file has failed at the one job it had.

So, every line carries one sentence describing what is actually in there, written for a reader who is deciding whether to bother. Then, look at the second half of the one above, because it is doing something quietly clever. "Supersedes the 2025 80/20 target" is not describing this file at all. It is warning you about a different one. The good summaries always end up doing that.

There is one thing that sentence is quietly bad at, and I would rather name it than let you discover it. It carries the words I happened to use. A question phrased in different words can miss it completely. On the platform I keep an explicit list of keywords beside each area for exactly that reason, so that a question about JWT tokens finds the authentication material even though no file says JWT. I left a keywords field out of this format because at eight files the sentence does the job, and I would rather you saw the smallest thing that works before you saw the version with more machinery bolted on. When you outgrow it, that is the first field to add back, and I come back to how it behaves at scale near the end of the chapter.

Is it still true? This is the field I lean on more than any other, and it is the only temporal field in the whole design. Here is how the frontmatter tracks freshness:

```
---
name: Target allocation (current)
description: The 60/40 target adopted January 2026. Supersedes the 2025 80/20 target.
last_verified: 2026-07-01
---
```

The frontmatter has just three fields: `name` for reading, `description` as the source the index line comes from, and `last_verified` for freshness. Only the first two ever reach the index line. The date stays back in the file, on purpose. Selection runs on directory plus description. The date earns its keep later, once a file is actually open, through the validator's stale advisories and the say-the-date convention below.

I had more. I had a `valid_from` and a `superseded_by`, and the `superseded_by` pointed at another entry's ID so the relationship would survive me moving files around. It was a nice piece of design, and I have deleted it, because two

simpler things do the same job better. The archive/ directory handles supersession by position: a file in there is history, full stop, and no field has to say so. And last_verified handles freshness for everything else.

The result of that practice is a convention rather than a mechanism, and the convention is this: when a fact is older than you are comfortable with, say so out loud and name who can confirm it. Not "here is the answer." Instead, "here is the answer, last verified in February, check the directory before you forward it." A date plus a habit of admitting the date. In the finance folder, that habit is written down as the third hard rule in AGENT.md, with a ninety-day threshold. I will show you in a moment why that beats a Boolean.

Those three fields are what the index is built from, which raises the obvious question of who builds it. For a long time the answer was me, by hand, and that is exactly where it went wrong.

The index writes itself

Here is the confession I used to bury and now lead with. Every version of this I hand-maintained went stale. Not in theory. Mine did, twice, in ways I did not notice until an assistant confidently cited a file I had moved three weeks earlier.

The reason is not laziness, and understanding why matters more than the fix. A hand-maintained index has to be updated by whoever adds a file. That works exactly as long as there is one way for files to arrive. When a second path exists, the index stops being a place files pass through. Someone has to remember to visit it, whether the path comes from a direct commit, a generated report, or an assistant writing its own notes. Nobody remembers forever.

So, the index is generated. `tools/build_index.py` walks the directories, reads the three frontmatter fields out of each file, and writes `INDEX.md` from them. Running it is the only way the index changes. Here is the build process in action:

```
$ python3 tools/build_index.py
INDEX.md written: 8 items
```

Eight items across five directories, and the generated file looks like this:

```
# Knowledge Index

<!-- GENERATED by tools/build_index.py - do not hand-edit. -->

## Rules

- [Decision rules](rules/decision_rules.md) - Risk tolerance, rebalancing triggers, giving plan, and
the equity blackout rule. Overrides all facts on conflict.

## Entities

- [M. Okonkwo (financial advisor)](entities/advisor_okonkwo.md) - Independent CFP, fee-only. Owns
rebalancing proposals and the annual review. Does NOT have trading authority.
- [Employer equity position](entities/employer_equity.md) - RSU vesting schedule and the blackout
window that gates all equity sale advice.

## Facts

- [Accounts and balances](facts/accounts.md) - Checking, savings, brokerage, retirement. Invented
balances.
- [Target allocation (current)](facts/allocation_2026.md) - The 60/40 target adopted January 2026.
Supersedes the 2025 80/20 target.

## Decisions

- [2026 annual review](decisions/2026-01-10_annual_review.md) - Why the allocation moved from 80/20
to 60/40, and what was explicitly rejected.
- [529 chosen over taxable brokerage](decisions/2026-03-14_529_vs_brokerage.md) - Why education
savings go to a 529 despite the flexibility loss. Includes the rejected alternative.

## Archive (history - not current truth)

- [2025 target allocation (SUPERSEDED)](archive/2025_allocation.md) - The 80/20 target used through
2025. HISTORY ONLY - superseded by facts/allocation_2026.md.
```

There is a teaching point sitting in the format, and I do not want you to walk past it. Here is the entire parser, and I mean entire:

```
LINK = re.compile(r'\[([^\]]+)\]\(([^)]+\.md)\)')

def parse_index():
    entries, section = [], None
    for line in read('INDEX.md').splitlines():
        s = line.strip()
        if s.startswith('## '):
            section = s[3:].strip()
        m = LINK.search(s)
        if m:
            path = m.group(2)
```

```
            hook = s.split(') - ', 1)[1] if ') - ' in s else ''
            entries.append({
                'label': m.group(1), 'path': path, 'hook': hook,
                'section': section, 'dir': path.split('/')[0]
            })
    return entries
```

Thirteen lines and one regex. No import that is not already in the standard library. The YAML version it replaced was longer, needed a third-party parser, and broke the first time somebody put an unquoted colon in a summary.

Look at the last line of the dictionary, because it is the design paying off rather than the parser being clever:

```
'dir': path.split('/')[0]
```

The parser recovers the type of every entry by splitting its own path on a slash. Not looked up, not declared, not stored anywhere. The folder already knew, so the parser just asks it. That single expression is what replaced the type: and authority: fields I used to maintain by hand.

Markdown links are clickable for you, searchable by a shell script, and parseable with one regex by anything else. Those are three readers of the same link. I did not choose the format to be clever. I chose it because it is the only one that all three readers handle without complaint.

That is the whole format, and every reader I care about can handle it. Now, watch it do the one job it was built for, and do it by hand first, before any code is involved.

Walking it by hand

Do this with a finger before you run any code. If you can walk it on paper, you will never once be confused about what the notebook is doing, and you will be able to debug it when it goes wrong on your own files.

The question is the one from the last section: what is my current target allocation, and can I add more to my tech stock? Throw away the words carrying no information, and a small handful is left. Target. Allocation. Tech. Stock. Now, read each of the eight lines and count how many of those words show up.

Here is what that actually produces, and I want you to look at it before I tell you what it means, because two of these are surprising:

```
score 0 | rules/decision_rules.md
score 0 | entities/advisor_okonkwo.md
score 0 | entities/employer_equity.md
score 0 | facts/accounts.md
score 2 | facts/allocation_2026.md
score 1 | decisions/2026-01-10_annual_review.md
score 0 | decisions/2026-03-14_529_vs_brokerage.md
score 2 | archive/2025_allocation.md
```

The current allocation scores two. The annual review scores one, because it discusses the allocation change. The archive scores two as well, tied with the current file, because last year's plan is, word for word, about the same subject.

Now, the two surprises, and neither is a bug.

The rules file scored **zero**. A question about my allocation does not use the words in my rules file, and yet the rules are the thing most likely to change the answer. Hold that, because it is a whole section later.

And `entities/employer_equity.md` scored zero too, on a question that literally contains the words tech stock. My equity file talks about RSUs (restricted stock units) and concentration and blackout windows. The equity file does not use the phrase tech stock. A word counter is not a reader.

Open the highest scorers, most authoritative first, answer from those, and leave the rest sitting closed on disk. This process is retrieval. No database, no embeddings, no vendor, no pipeline to maintain.

Notice what the walk did not need. The walk did not need to know what a vector is. It needed a folder structure and eight sentences. Notice what it just failed at twice, in eight lines, on the first question I tried.

Encoding the same walk in Python

Everything you just did by hand is *Part 3* of the notebook (search for `=== Part 3: Map-driven assembly`), and it is four small functions where each one is a step you already performed with a finger.

You have seen the first, `parse_index`. Here are the other three, and I want you to notice how much of the design is sitting in two constants rather than in any logic:

```
# Directory name is the type; this order is the precedence order.
PREC = {'rules': 0, 'entities': 1, 'facts': 2, 'decisions': 3, 'archive': 4}
STOP = {'my', 'the', 'a', 'an', 'is', 'are', 'do', 'does', 'and', 'or',
        'to', 'of', 'in', 'on', 'for', 'what', 'how', 'i', 'me',
        'can', 'should', 'with', 'it', 'that', 'this', 'more',
        'add', 'current', 'some'}

def score_entries(entries, question):
    q = {w.strip('?.,').lower() for w in question.split()} - STOP
    out = []
    for e in entries:
        words = set(re.findall(
            r'[a-z0-9]+', (e['label'] + ' ' + e['hook']).lower()))
        out.append((sum(1 for w in q if w in words), e))
    return out

def pick(entries, question, allow_archive=False):
    chosen = [
        e for s, e in score_entries(entries, question)
        if s > 0 and (allow_archive or e['dir'] != 'archive')
    ]
    for e in entries:     # rules always apply, even when nothing matched
        if e['dir'] == 'rules' and e not in chosen:
            chosen.append(e)
    chosen.sort(key=lambda e: PREC.get(e['dir'], 9))
    return chosen

def assemble(entries):
    parts = []
    for e in entries:
        parts.append('----- ' + e['path'] + ' -----')
        parts.append(read(e['path']))
    return '\n'.join(parts)
```

That is the whole retrieval engine. Read pick closely, because three separate arguments from this chapter are sitting in five lines of it.

`e['dir'] != 'archive'` is supersession. Not a `superseded_by` field, not a relationship graph. A directory name, compared to a string. That one condition is what stops the assistant quoting last year's plan as though it were today's, and it is the entire mechanism.

`if e['dir'] == 'rules' and e not in chosen` is the rules always applying. Everything else in this system earns its place by matching the question. Rules earn nothing, because a rule that only applies when you happen to mention it is not a rule. Two lines, and they are the difference between an assistant that knows your rules and one that follows them.

`chosen.sort(key=lambda e: PREC.get(e['dir'], 9))` is ordering by trust. Change that dictionary and the whole system's opinion about what wins changes with it.

I want to be careful about what I am claiming there, because I overclaimed it in an earlier draft and a reader with the folder open would have caught me. The order is written down in more than one place. PREC (the precedence table at the top of the loader) is the copy the code obeys. `AGENT.md` states it in plain English, because the model is a different reader than the interpreter and it needs to be told. The generator lists the directories in that same order so the index comes out sorted the way the loader will read it. Three readers (the loader, the model, and the generator) and three copies (PREC, AGENT.md, and the generator's directory list). The loader reads PREC, the model reads AGENT.md, and the generator keeps its own directory list. The copies can drift apart from each other.

What changed is the count. The old authority field lived on every record, so it had as many copies as I had files and each one could rot on its own, quietly, forever. The new design has one copy per reader, and there are only ever three of them. That is not zero. It is small enough to check in a minute, which the per-record version never was.

assemble glues the chosen files together, each under a header naming its path, so the model can see where every fact came from and say so in the answer.

Now, run the cell.

Then, the numbers, and I promised you the awkward version, so here it is. Here are the actual token costs for this question, showing both the whole-folder dump approach and the two map-driven approaches:

```
Token cost for this question:
  dump (all content)   : 1156
  map, code navigates  : 440 (only the chosen files)
  map, model navigates : 1060 (AGENT.md 277 plus INDEX.md 343 plus chosen 440)
```

Three numbers, because there are genuinely three cases and conflating them is how this pattern gets oversold.

When the Python code does the navigating, only the chosen files ever reach the model, so the cost is 440 tokens against 1,156 for the dump. When the

model does the navigating, which is what happens inside a real coding assistant, the index has to be in the context too, and the honest comparison is 1,060 against 1,156. Less, but barely. The asymmetry is worth naming before moving on: the 440-token context answers the allocation half of the compound question, while the dump could answer both halves. The tech-stock half stays unanswerable until the rules section opens the equity file. Figure 11.2 puts the three side by side, and the dump and the model-navigating map come out nearly the same height:

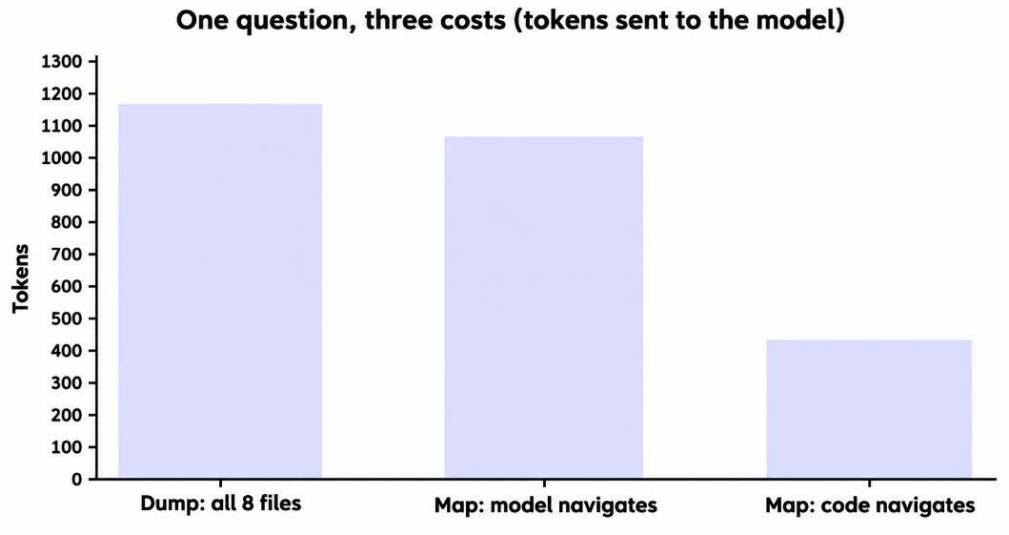


Figure 11.2: Tokens sent to the model for one question, three ways

Each bar is the same question answered a different way. The first is the dump, all eight content files. The third is the map with the code navigating, which sends only the files it chose. The middle one is the map with the model navigating, which has to carry AGENT.md and INDEX.md too, and that is why it lands so close to the dump.

Run all five test questions and the picture is worse than barely:

```
query                                    dump    map  ratio
Q1 current target allocation             1156   1060  1.09x
Q2 who is my advisor, what recommended   1156   1002  1.15x
Q3 raise cash for the 529                1156   1069  1.08x
Q4 why the 529 over a brokerage          1156   1213  0.95x
Q5 checking balance (control)            1156    883  1.31x
TOTAL                                    5780   5227  1.11x

Q3 also loads entities/employer_equity.md: the T0 blackout rule names that
file for any sale question, so the table counts it in the payload.
```

The dump costs eleven percent more overall, and look at Q4. The map costs *more* than dumping the entire folder, because that question needs the decision record, the rules, the index, and the resident contract on top, and by the time you have loaded all four, you have paid more than the eight files would have cost you outright.

I want to sit in that rather than move past it, because a book that only shows you the flattering number is teaching you to trust the wrong thing. At eight content files, the map is roughly break-even on tokens and occasionally worse. The always-resident floor, `AGENT.md` at 277 plus `INDEX.md` at 343, is about 620 tokens against a total of roughly 1,700. That is thirty-six percent of everything I own, spent before a question is asked, and it is far worse than the five percent I aim for on a real repository, a budget of my own rather than an established threshold.

The reason is arithmetic, not design. At ten files, the floor is mostly the rules, and the rules do not shrink. You cannot amortize a fixed cost across ten items.

So, why do it at all at this size? Because of the three questions the token count cannot see, and they are the next three sections. Which of two contradicting files is true. Whether the assistant will honor a rule it was never asked about. Who a person is, when no file is about them.

Every one of those is a correctness question. The map earns its place on correctness at ten files, and on tokens somewhere around a hundred. I will show you the crossover before the chapter ends.

One more thing to do to park. That split between the code navigating and the model navigating comes back later, and it turns out to decide more than cost.

This idea has company, which is how I know it is not just a personal quirk. Anthropic has written about letting an agent search a codebase with plain tools, such as grep, instead of pre-indexing everything into vectors [12], a point Gergely Orosz also covered in his engineering writing [5]. VectifyAI's PageIndex takes a similar navigate-first stance [6]. The industry umbrella term is vectorless RAG. I want to be careful here: I am not telling you semantic search is dead, and I am not predicting it dies. Anthropic has also published on when contextual retrieval is worth the machinery, which is worth reading before you decide you need it [15]. What I am pointing at is a convergence. Several people, working separately, landed on the same move: navigate to the

knowledge instead of embedding all of it up front. I wrote up my own version of this across a few articles [2, 3, 4], and it grew out of that 1,200-API problem, not out of theory.

I should answer the objection you are probably holding, because a reasonable engineer raises it about now: if the files are right there on disk, why not just let the assistant grep them? Grep is genuinely good, and I use it constantly. However, grep finds the word, and the word was never the hard part. Grep has no opinion about which of two matching files you should trust, and no idea that one of them stopped being true in January. Trust and time are the two things a text search cannot give you.

The next section proves it, because it turns out to be the whole argument.

Facts that expire

Ask the folder what my target allocation is, using nothing but text search, and watch what comes back in *Part 4* of the notebook (search for === *Part 4: Temporal precedence*):

```
Baseline: search the corpus for the numbers.
  text search "60/40" or "80/20" -> 8 hits across 5 files:
      archive/2025_allocation.md
      decisions/2026-01-10_annual_review.md
      entities/advisor_okonkwo.md
      facts/allocation_2026.md
      rules/decision_rules.md

  Both 80/20 (archive) and 60/40 (current) appear. A raw hit list enforces
  no precedence: a search-first agent can answer 80/20 and cite a real file.
  That is the failure.
```

Eight hits across five files. One of them says sixty/forty and one says eighty/twenty, and the search presents them in whatever order it happened to walk the directory. Nothing in that output ranks the current file above the dead one, and the SUPERSEDED label the archived file carries never surfaces in this list of matching paths. A text search cannot tell you what year it is.

Read the middle sentence of that note again, because it is the one this whole chapter exists for. A search-first agent can answer eighty/twenty **and cite a real file for it.** Hold that thought for two pages.

The temporal drift problem from *Chapter 5*, one level up. Back then it was stale rows in the data. Here, it is a stale decision sitting in a file with an authoritative tone.

Two things break the tie, and neither of them is a field on a record.

The first is position. `archive/2025_allocation.md` is in the archive directory, and `AGENT.md` states the precedence: archive is history, never current truth. The loader drops it before scoring even matters. No **`superseded_by`** pointer, no relationship graph, just a folder whose name means something and one line in tier zero that says what it means.

The second is `last_verified`. The current allocation file carries 2026-07-01. The archived one carries 2025-01-01. When the model is the one navigating, that date does real work, because a model can weigh a January note against a question about right now in a way that forty lines of Python cannot.

Which brings back the split I asked you to park. When code navigates, the dates do nothing for selection, because they are not in the index and the loader never looks at them. When the model navigates, they become the mechanism. Same field, two completely different jobs, depending on who is holding the map.

Position and a date are what break the tie between the current file and the archived one. Which leaves a question I get asked every time I show this: if the archived file is never the right answer, why keep it at all?

Keeping the dead file

Keeping the old file is not clutter; it is the point. Because I kept it, the assistant can still answer a different question: what was I aiming for back in 2025, by opening the archived plan as history when the question is explicitly about the past. That is the line I want you to hold on to: old facts become history, not lies. A deleted fact takes its context with it. A superseded fact keeps the story.

This is the same instinct behind bi-temporal knowledge graphs, such as Zep and Graphiti [7], which track both when something was true and when the system learned it, so the past stays queryable without polluting the present. Two clocks. Mine has one. My files record when a fact was last verified and say nothing about when I found out, and for a folder I maintain by hand that has never cost me anything, because I am the one who put the file there. On a

system ingesting facts from somewhere else, where the gap between a thing being true and you hearing about it is the whole problem, the second clock stops being academic. Borrow the idea before you need it, and you are maintaining one more field, the date you learned the fact beside the date you last verified it. Borrow it after you need it, and you are reconstructing a history you no longer have.

I hit the need for this while organizing my own agentic repository, when I could not reconstruct why a particular decision had been reached. The decision was documented. What replaced it was documented. The thread between them was not. That one stayed inside my own repository. The one I am about to show you did not, and at this point I would rather show than keep arguing: trust and time are what a text search cannot give you, and here is that going wrong on a real system, with real consequences. It did not happen in a notebook.

Some time ago, an AI assistant working across my platform documentation invented a job title for a colleague. Not maliciously, and not from nothing. The assistant read a meeting note that carried the wrong title, and it repeated what it read. That wrong title then propagated into dozens of files, because every document that referenced the person copied it forward. We found it when the wrong title turned up in front of exactly the wrong audience.

We fixed the files and wrote down the correction. Now, here is the part that matters for this chapter.

Months later, I asked two different systems the same question about that colleague. The first searched the documentation corpus, more files than anyone could read. The first system found the old meeting note, which still exists because meeting notes are history and you do not rewrite history. It answered with the fabricated title, and it gave me a precise file and line number for where it got it.

That citation was correct. The file really does say that. The answer was still wrong.

The second system read a small, curated index, opened one file about that person, and answered with the right title, the date it was last verified, and an unprompted warning that the verification was months old, and that I should confirm it before putting it in anything I would send to someone else.

I want to name the two lessons in that, separately, because they are usually collapsed into one and they are not the same.

A text search cannot detect a supersession that is not written inside the file it found, and this is structural rather than a matter of quality. The stale fact lives in a genuine, dated file. Nothing in the corpus marks it as retired, because the correction is not a property of that file. It is a property of the relationship between that file and a newer one, and relationships are exactly what a search over file contents cannot see. Ranking alone cannot fix this. The information needed is not in the thing being searched.

A citation is not a correctness guarantee. This is the part I find genuinely uncomfortable. The wrong answer had a real file path and a real line number. It passed every mechanical check you could write. It looked *more* rigorous than the right answer, because it was more specific. If your defense against hallucination is "make the assistant cite its sources," you have defended against the assistant inventing a file. You have not defended against it faithfully quoting a file that stopped being true.

That is what `last_verified` plus the habit of admitting the date is for. Not a Boolean saying current or not current, which is a claim nobody can keep updated. A date, and a convention of saying the date out loud when it is old.

I keep this story in the chapter because it is the strongest argument I have for the whole method, and because it is the one that cost me something. The token savings are a convenience. This is the part that protects you.

The map now points to the right facts, and current ones. However, facts are not the only thing the assistant needs from me. I have standing rules I always want followed. Standing rules are not facts about my accounts; they are instructions about my behavior. How do those travel with the assistant, so it follows them without me repeating them every single session?

Rules the agent follows

Facts tell the assistant what is true. They do not tell it what you want.

My earlier design handled this with a field. Entries carried type: procedural, and the loader made one extra pass to append any procedural entry whether it matched the question or not, because a rule that only applies when you

happen to mention it is not a rule. That worked, and I want to show you the exact question that broke it, because the break is instructive.

Ask this, in *Part 5* of the notebook (search for "=== Part 5: T0 safety rules"): should I sell some stock to fund the 529?

Score that against the index. The words that carry meaning are 529, fund, sell, stock. The 529 decision record matches on 529. The rules load anyway, because rules always load. And `entities/employer_equity.md` scores zero, because the question is phrased about the 529, not about equity, and nothing in it uses the words in the equity file.

That file is where the blackout window lives. My employer equity cannot be sold between quarter-end and the earnings release, and as of the notebook's pinned date, the twenty-fifth of July 2026, that window runs from the thirtieth of June to the twenty-ninth of July, dates I invented along with every other number in this folder. A loader that opens files by keyword match never opens it. So, the assistant answers a reasonable-sounding yes, and it is a yes to something I am not permitted to do.

Notice what went wrong, because it is not a scoring bug. Scoring worked correctly. The question genuinely is not about equity, in the words the question used. **The rule was safety-critical, and its presence depended on keyword matching.** Any mechanism that decides what to load by relevance will eventually fail this way, because relevance to the question and relevance to the safety of the answer are different things.

So, the fix is not a better scorer. It is structural. The hard rule lives in `AGENT.md`, tier zero, read every session no matter what was asked. Here is the hard rule that must be resident in every session:

```
## Hard rules (never lazily loaded)

1. **Blackout windows.** Never propose selling employer equity between quarter-end and the
   earnings release. Check `entities/employer_equity.md` for the current window before any
   equity advice.
```

Look at the shape of that rule, because it is doing two jobs at once. The constraint is resident, so it is never absent. And it *names the file to check* rather than restating the window, so the resident copy never goes out of date when

the window moves. The rule is in tier zero. The detail is one hop away in tier two.

The principle that survived from the older design is this: anything you cannot afford to have missed does not get to be conditional. Everything else earns its place by matching the question.

I will give you the cost of that, because tier zero is not free. The rules and the precedence and the navigation instructions come to about 277 tokens, and I pay them on every question forever. On a corpus of ten files, that is a meaningful fraction. On a corpus of four hundred it is a rounding error. Fixed costs are the one thing that get cheaper as you grow.

Whether it actually obeys

Knowing the rule and following the rule are different things, so the second half of *Part 5* runs the same 529 question with the context assembled properly: `AGENT.md` resident, the rule pointing at the equity file, and the equity file open because the rule said to check it.

Run it with a key, and the answer changes shape. It declines the sale, names the blackout, and quotes the window it read from the equity file. Nothing clever happened in the model. The knowledge did the work; the model just read it and honored it. The mechanism is the whole trick, and it is almost boring once you see it: behavior you want reliably is behavior you write down where the assistant will always look.

Now, let me be precise instead of triumphant, because two different claims are tangled together in that result and only one of them is guaranteed.

The first claim is structural: the rules are always present. Tier zero puts them in the context every single time, and that is code; it does not rely on anybody's goodwill. The second claim, that the rules win, is not structural at all. Read the line at the top of the rules file: overrides all facts on conflict. That sentence is the enforcement. It is a piece of English the model reads and chooses to respect, which in my experience it does, consistently, and which it remains entirely free not to do.

So if you are building something where a broken rule is expensive, do not let a markdown file be your only defense. Put the hard check in code, outside the model's discretion, and test it like any other code, because an incomplete

check fails quietly too. For the blackout rule, that means the code that would act on the advice, an order form or a trade ticket, reads the window from the equity file itself and refuses a sale inside it, whatever the model said. The map is very good at guaranteeing the model knows the rule. The map cannot guarantee obedience, and that is a limit to design around rather than discover.

If this sounds familiar, it is the same convention that has quietly taken over serious AI-assisted work: the `CLAUDE.md` and `AGENTS.md` files people keep at the root of a repository so the tools that read them inherit the same rules [8]. I keep those files tool-neutral on purpose, so the rules survive when I switch between assistants. The rule belongs to the work, not to the vendor.

Representing people and systems

Here is a question that broke my design in a way I did not see coming, and it is the most common kind of question I actually ask.

Who is my advisor and what have they recommended?

Score that against an index of documents and watch it fail. Not fail loudly, fail quietly, which is worse. Nothing in the index is *about* the advisor. She turns up inside the annual review, because she recommended the allocation change. She appears once more in a line of the current allocation file. Two files mention her, but neither is her file. The assistant can only reconstruct a person from those passing references and hope it caught them all.

I lived with this for a long time without noticing it was a design gap, because I was thinking about knowledge as documents. Documents have topics. People are not topics. A person is a thing that appears across topics, and an index whose unit is the document has no place to put them.

So, the folder has a tier for them, and *Part 6* of the notebook walks it (search for *=== Part 6: Per-entity files):*

```
entities/
├── advisor_okonkwo.md
└── employer_equity.md
```

One file per recurring entity. A person, a system, a service, a program, anything you keep having to know about. The contents are not a biography.

They are the things I keep needing and keep forgetting. Here is what an entity file contains, organized for immediate reference:

```
# M. Okonkwo - Financial Advisor

**Role:** independent fee-only CFP. Engaged 2024. **No trading authority** - proposals only; the
human executes.

**What they own:** the annual review, rebalancing proposals, tax-loss harvesting suggestions.

**What they have recommended**
| Date | Recommendation | Outcome |
|------|----------------|---------|
| 2026-01-10 | Move 80/20 to 60/40 ahead of retirement horizon | Accepted - see `decisions/2026-01-
10_annual_review.md` |
| 2026-03-14 | Fund education via 529 rather than taxable brokerage | Accepted - see `decisions/2026-
03-14_529_vs_brokerage.md` |

**Constraint:** they are not a tax advisor. Route tax questions elsewhere and say so.
```

Four things in there, and every one of them is a thing my old document index had no place for.

The **authority limit** is the one I care about most. No trading authority, proposals only. The authority limit is not a fact about my finances. It is a boundary on how the assistant should treat anything attributed to her, and if it lives nowhere then every recommendation reads as an instruction.

The **recommendation history with outcomes** turns a scatter of mentions into a record, and notice that it links out rather than restating. The entity file is an index of its own, one level down.

The **out-of-scope constraint** is a rule attached to a person rather than to a topic, which is a category my original design had no room for at all.

The **dates** let the freshness convention apply to a person the same way it applies to a fact.

This tier replaced something, and it is worth saying what, because the thing it replaced looked more sophisticated. My earlier design had a `relates_to` field: each entry listed its neighbors, and following those links pulled in related material. It worked, and the notebook used to show it working. It also dragged in my account balances for a question about what I was aiming for in 2025, because the rules entry listed accounts as a neighbor and a cluster has no idea

what you asked. A question that needed two files pulled four, and one of the four was pure noise.

Clusters are blunt. A per-entity file is scoped, individually citable, individually updatable, and costs nothing until somebody asks about that entity. I gave up a graph and got better answers, which was not the trade I expected to be making.

It closes a loop I left open two sections ago. I promised that keeping the archived 2025 plan meant the assistant could still tell me what I was aiming for back then. The advisor's recommendation table is where that thread now lives: her January recommendation links to the decision record, which explains the move from eighty/twenty and what was rejected. The history is reachable on purpose, because something points at it deliberately, rather than leaking in. After all, the cluster was too broad.

Supersession keeps history from showing up when nobody wanted it. The entity files let you go and fetch it when somebody did.

Growing junior to senior

The map handles a handful of files without breaking a sweat. But a real brain grows. Statements pile up, decisions accumulate, notes and reviews stack into a year of history. The fair question is whether the pattern still holds at that size, and the more interesting one is whether the assistant improves as the brain fills in. It does, and that is the payoff. Each session, the assistant starts from the accumulated brain, so it knows more this month than it did last month. Not because the model got smarter, but because the knowledge it can navigate got deeper. The experience lives in the repository, not in the model's weights. You can swap the model underneath and keep the seniority, which is exactly backward from how most people think about this, and exactly why it is durable.

Generate, then govern

I told you earlier that the index is generated, and I want to come back to it here because this is where it stops being a convenience and becomes the thing that holds the whole design together.

At the size of that finance folder, hand-writing eight lines is genuinely the correct answer. Twenty minutes, finished. Hand-writing stops being the correct answer fast. At a few hundred entries nobody is hand-writing signposts, and anyone who tries ends up with an index that went stale somewhere around entry ninety.

So, the generator does the mechanical part. The generator reads each file, takes the name and the description out of the frontmatter, works out which directory the file is in, and writes the index. The description still needs review, whether a person writes it or an assistant drafts it, because it guides the choice of which file to open.

Generate, then govern. The machine writes the signposts. You keep the trust.

One thing to know before you point that script at a repository you actually care about. It walks five named directories, one level deep, because one level is all this folder has. I ran it against a different corpus of mine, six hundred files nested fifteen deep, and it indexed six of them, the six that sat directly inside folders with those five names. Everything else was in a folder the script does not know about, more than one level down, or not a markdown file. That is not a bug; it is the teaching version being the teaching version. A nested corpus needs recursion, a way to skip vendored subtrees that are somebody else's code, and paths that stay correct once they are more than one segment long. That is code rather than configuration, and it is the first hour of work on any real repository.

Then you need one more thing, which I resisted for far too long: something that fails loudly when the index and the files get out of step.

Part 7 is the cell I would insist you actually run rather than read, because it breaks the system in front of you and then fixes it. It works on a throwaway copy, so nothing you do here touches the real folder. Here is the validation code that deliberately introduces a drift and catches it:

```
def run_tool(tool, root):
    r = subprocess.run(
        [sys.executable, str(Path(root) / 'tools' / tool)],
        capture_output=True, text=True
    )
    return r.returncode, (r.stdout + r.stderr).strip()

tmp = Path(tempfile.mkdtemp(prefix='finance_agent_'))
```

```
shutil.copytree(AGENT, tmp, dirs_exist_ok=True)

rc, out = run_tool('build_index.py', tmp)
print('build_index :', out)

rc, out = run_tool('validate.py', tmp)
print('validate    : rc =', rc)
print(out)

# Simulate hand-drift: delete one real pointer line from the generated index.
idx = tmp / 'INDEX.md'
lines = idx.read_text().splitlines()
dropped = next(l for l in lines if 'advisor_okonkwo.md' in l)
idx.write_text('\n'.join(l for l in lines if l != dropped) + '\n')

rc, out = run_tool('validate.py', tmp)
print('validate    : rc =', rc, '(non-zero means the drift was caught)')
print(out)

# Fix by regenerating. The generator is the single source of truth.
rc, out = run_tool('build_index.py', tmp)
print('regenerate  :', out)

rc, out = run_tool('validate.py', tmp)
print('validate    : rc =', rc)
print(out)
```

The output, which is the most satisfying thing in the notebook:

```
build_index : INDEX.md written: 8 items
validate    : rc = 0
checked 8 files, 8 pointers
  ADVISORY stale 133d: entities/advisor_okonkwo.md
  ADVISORY stale 196d: decisions/2026-01-10_annual_review.md
  ADVISORY stale 133d: decisions/2026-03-14_529_vs_brokerage.md
PASS - all checks green

Hand-edited the index: removed the line for entities/advisor_okonkwo.md
validate    : rc = 1 (non-zero means the drift was caught)
checked 8 files, 7 pointers
  ADVISORY stale 133d: entities/advisor_okonkwo.md
  ADVISORY stale 196d: decisions/2026-01-10_annual_review.md
  ADVISORY stale 133d: decisions/2026-03-14_529_vs_brokerage.md

FAIL:
  [2] not indexed: entities/advisor_okonkwo.md

regenerate  : INDEX.md written: 8 items
validate    : rc = 0
checked 8 files, 8 pointers
```

```
  ADVISORY stale 133d: entities/advisor_okonkwo.md
  ADVISORY stale 196d: decisions/2026-01-10_annual_review.md
  ADVISORY stale 133d: decisions/2026-03-14_529_vs_brokerage.md
PASS - all checks green
```

Read that middle block. **Eight files, seven pointers.** The advisor's file is still sitting on disk, perfectly intact, and it has become invisible: no line in the index means nothing will ever open it. Every question about her would come back thin, and nothing about the failure is loud. The file is there. The answer is worse. That is exactly how a hand-maintained index rots, and the only reason you saw it here is that something counted.

A nonzero exit is the whole point. Wire this into whatever runs your tests and drift stops being a thing you discover months later in an answer you have already forwarded.

Six checks, each one earned by something that went wrong: every pointer resolves, every file is indexed exactly once, nothing exceeds the size ceiling, no two files share a basename without disambiguation, every file has its three frontmatter fields, and no index claims to be automatically updated without carrying the generated marker that backs the claim.

Notice the advisories too, because they are the freshness convention doing its job unasked. Three files are over ninety days past their `last_verified` date, and the validator says so every run without failing the build. That is the distinction I want: a broken pointer is an error, and an aging fact is a nudge. Conflate the two and people start ignoring the output.

One note on those day counts: the notebook pins a reference date, the twenty-fifth of July, so the numbers you see match the ones printed here. A live repository uses the real clock instead, because a real system has to report the true age of a fact, not a frozen one. I froze it only so the example stays reproducible, which is the `last_verified` idea turned back on the book itself.

The sixth check sounds petty, and it is the one I would keep if I could only keep one. I once had an index with a line at the bottom saying it was automatically updated. It was not. It updated its own timestamp and warned about problems it never fixed. So, it looked maintained every day, while drifting further from the truth, and nobody checked it because the file said it

was fine. A false claim of automation is worse than no automation, because it buys silence.

I will also tell you what the validator cannot do, because I have been careful elsewhere and this is a place where people over-trust tooling. The validator checks that the index is structurally honest. The validator has no idea whether the description is *accurate*. A file whose description says one thing while the contents say another passes every check. Structure is verifiable by a script like this one. Meaning is not, and that stays your job.

When I wrote that last line, I expected the gap it describes to stay open for a while. This validator still cannot verify meaning. You can, though, catch the moment it may have changed. Nexla has published a pattern for doing this with the tool definitions it generates from its data products [34]. Each definition keeps a hash of the schema it was built from, and a mismatch on the next ingestion is a drift event, as is a glossary edit on a term that appears in a description. The definition is then regenerated from the source rather than patched by hand. Additive changes such as a new optional parameter are approved automatically, and anything that changes the meaning goes to a person for review. Part of that runs today: their data products compare the schema on every ingestion and flag what changed, and tools are generated from those data products, so regeneration means rerunning the generation. The automatic link from the drift event to the rebuild is the part they present as the pattern they are working toward, as of September 2026. That is the same generator and validator from Part 7, triggered by the change itself instead of by someone remembering to run them. In your own repository, the trigger can be a merge to the main branch or an edit to a policy document, and the review then reaches the person who owns that file while the change is still fresh.

The same trigger helps the facts tier from earlier in this chapter. When a source document changes, you can flag every fact that names it as its source for re-verification that day, instead of waiting for its last_verified date to pass ninety days, and a new file that seems to replace an old one can ask its owner whether the old one belongs in the archive. That event also gives you the second clock my own files lack: a record of when you found out. Whether the new file really supersedes the old one is still the owner's decision.

This is the moment when the repository stops being storage and becomes the agent. The knowledge is its brain. Grow the brain, and you grow the agent, the same way you grow a new hire from junior to senior by giving them context, history, and standards until one day they stop asking you the same question every morning. I have been running my own work this way for a while, and I wrote up the pattern as Agentic Repos and as growing AI agents from junior to senior [8, 9, 10, 11].

The agent becomes senior in your domain, on the finances or the codebase you have taught it. The agent does not become senior in some absolute sense. It is a specialist you grew, not a genius you rented.

Understanding when tokens become critical

I owe you the crossover I promised, and it is a table rather than an argument. Read the last two rows as planning heuristics from this corpus shape rather than universal thresholds. Table 11.1 shows how a flat index scales across corpus sizes and where the token argument becomes meaningful:

Corpus	Flat index size	Load every session?	What to do
10 files (this chapter)	~350 tokens	yes, trivially	Read it all if you like. The index earns its place on correctness, not tokens.
~100 files	~3,000 tokens	yes	The index now saves tokens too.
~400 files	~10,000 tokens	yes, at its ceiling	A flat index is near its practical limit.

Table 11.1: How a flat index scales with corpus size, and what to do at each threshold

The last row shows where a flat index reaches its limit, but not what to do about it. Past a few hundred files, the index itself is too big to read every session, which is the same problem the files had at the start of this chapter. So, you split the index the same way you split the files: a short root index with one line per area, read every session, and a separate index for each area that the assistant opens only when the question points there. When a design fixes

its own limit by being applied again, that is a good sign it will keep working as your corpus grows.

There is a version of this argument I find more persuasive than any of my own numbers. The assistant I use every day works exactly this way, and I did not build it. It keeps a small index of one-line entries, always resident, and a much larger set of individual notes it opens only when one is relevant. The index is about three percent of the whole, measured in tokens when I checked my own setup in July 2026 (about 5,200 of roughly 157,000). When I ask it about a person or a decision from a year ago, it reads the index, opens two or three notes, and answers with citations and dates. It is the same three tiers, arrived at independently, which is good evidence that the shape is right rather than merely mine.

I will name the uncomfortable implication, because it cuts against the artifact this chapter is named for. The win is the *tiering*, not any particular file. A single knowledge map is the right shape for a small folder and the wrong shape for a large one, and I spent a year defending the file when I should have been defending the layering.

The discipline of context engineering

The brain grew, the agent got better, and that success is what created the real problem. There is now more relevant knowledge than fits in the window. You cannot send it all. You have to choose. This is the moment the "context window" cost I flagged all the way back at the start of the chapter finally comes due.

Choosing well is context engineering: the discipline of assembling the smallest, useful context for each task. Not the most context. The smallest context that still does the job. The map is what makes the choosing possible, by letting you see what is available without loading it. The discipline is what makes the choosing good.

Three questions sit on the table: which slice do you feed the model, in what order, and at what moment. You have already met all three without my putting names to them.

Start with which slice, because you read about that one going wrong. When I measured the earlier cluster design, following neighbor links pulled four files

and 881 tokens for a question that needed two and 536, and the extra was not a rounding error; it was my account balances turning up for no reason at all. The answer is to read the index first and open entries on demand, one at a time, instead of front-loading everything that might conceivably be useful. Progressive disclosure, if you want the term [12], and it is the same budgeting instinct I wrote about for context windows generally [14].

Order is the easy one, and the folder structure has been handling it for you since we agreed that directories carry precedence. Where something sits in the context changes how much weight it carries, because a model often attends to the beginning and the end of the window more reliably than to the middle [28]. So, the rules go where they will actually land, and the directory order decides that instead of you deciding it fresh, by feel, every time.

At what moment is the one I underestimated for longest. It is the difference between loading a file because the question mentioned it and loading a file because the *answer* depends on it. The blackout window shows it: a question about a 529 needed a rule about equity, and keyword matching was never going to connect them. Some things load because they matched. Some things load because you cannot afford to be wrong. Knowing which is the discipline.

So how do you know you picked the right size? You do not guess it; you measure it. I hold what every session pays, the tier-zero file and the index, to about five percent of the repository, and I keep every content file under the validator's size ceiling so that opening one has a known price. Then, for each question, I load the rules, open what the index points at in precedence order, and stop. Whether that was enough is a job for two numbers from *Chapter 9*, *Task-Specific Evaluation Data*. Context recall asks whether everything the answer needed made it in, and context precision asks how much of what made it in was needed. Fix recall first. The 440-token context for the first question in this chapter looked like the win, and it was too small, because the tech-stock half needed the equity file. A recall miss gives you a wrong answer. A precision miss costs you tokens and, at worst, a contradiction like the one the dump handed you.

Applying the pattern across domains

Everything so far ran on my finances, and I kept promising the pattern had nothing to do with money. The pattern does not, and I can prove it in the domain where I actually built it: an engineering repository. Not just code. A real engineering project carries its architecture, the decisions behind it, the runbooks, and the rules the team works by, and that is exactly the material a map is for. The finance folder was the teaching version, small enough to read top to bottom in minutes. What I run on a production platform is the same shape at a size that punishes a bad map, and it is where the method earns its keep.

I should be honest that the finance example was built backward from this one. For more than two years, I used AI coding assistants every day on a production API gateway platform, more than 1,200 endpoints across a dozen service domains, and I watched the waste from the opening of this chapter happen on an industrial scale. Every session started with me re-explaining the platform. So I did on that repository exactly what we just did on the finance folder, and I kept going until it stopped costing me anything. I wrote the method up as I went, in a handful of articles I will point at as we pass each piece [2, 3, 4, 16].

Applying the same tiers to an engineering repository

Point it at a repository, and the parts line up one for one. There is one difference: a codebase punishes a bad map faster than a personal folder ever will, so what I could wave at earlier now has to be written down.

The `rules/` directory grows into the material the whole team works by, and the top of it is read-only, changed by a human and never by the assistant. The moment AI output mixes into the material you stand behind, you can no longer tell what you verified from what a model guessed.

`entities/` turns out to matter more on a codebase than on a folder of finances, which surprised me. A service, a gateway, an authentication component, the team that owns a subsystem: those are exactly the things people ask about and exactly the things no single document is about.

`decisions/` becomes the most valuable directory in the repository over time, and it is the one teams skip. Not what the architecture is, which the code

already tells you, but why it is that and what was rejected. A year later, the code answers the first question, and nothing answers the second. I learned that when an agent confidently rewrote a module along a design the team had ruled out months before. The decision lived in a meeting note, not the code, so the agent never saw it [10]. A decision record the index points at is what makes it available to later sessions.

Evidence gets an upgrade too. On a real repository, any claim that matters points at the files that back it, so when the assistant says the platform uses one authentication pattern and not another, it names the design doc it read that from. A claim with a source stapled to it is far harder to invent quietly, and that one habit did more to cut hallucination than anything else I tried. With the caveat I gave you earlier, and I will repeat it because it is the thing people forget: a source stops the assistant inventing a file. It does not stop the assistant quoting a file that stopped being true.

Revealing the search the finance example hid

The finance folder used one small scorer, about forty lines, because eight content files never needed more. A real repository does, and this is the part the teaching version quietly simplified. On the platform, the search runs in stages instead of a single pass [3].

It runs cheapest first. A topic index maps a question straight to a file and answers most of them on its own. When there is no direct hit, the assistant checks the concept groupings, which carry keyword metadata, so a search for "JWT tokens" lands on the authentication material even when no file uses that exact phrase, which is semantic recall with no embeddings anywhere. When that misses, it reaches for a set of saved search patterns, the greps I ran so often I stopped retyping them. Only when all three come up empty does it fall back to a full-text scan, and even then it looks in the most likely directories first.

Measured against a plain full-text scan on the real repository, the staged search came back roughly three to six times faster on search latency, and it lands on the right file for curated knowledge without a vector store, an embedding pass, or a reranker. Those numbers come from my own repository when its knowledge base held a little over fifty documents, as I reported in January 2026: the search step alone went from 300 to 500 milliseconds down

to 50 to 150 [3]. Your corpus and your hardware will give you different numbers.

Which brings up the question you should be asking by now, and if you are not asking it, somebody in your first design review will.

Using existing tools and what they cannot build

There are good tools for this now. Point one at a folder. The tool parses the code and pulls out the relationships. The tool writes an index file and a set of linked articles. The tool installs a commit hook so the whole thing rebuilds itself every time you commit. It is free, it is openly licensed, and it reads more languages than anything you or I are going to write on a weekend [21, 29].

So, use it. I mean that plainly. If your problem is that nobody can find anything in a large codebase, that problem is solved, and it was not solved by me. Tools are good at discovery, meaning finding where things are and how they connect. However, whether they are still true, who owns them, and which source wins when two disagree are different questions, and the rest of this section is about them.

What I want you to notice is what the tag on each edge is actually telling you. Graphify marks each relationship it produces as extracted, inferred, or ambiguous, which is an honest and useful thing to publish. Read it closely though, because it answers a narrower question than it appears to. The tag tells you how confident the parser is that the relationship is really there in the source. It says nothing at all about whether the source is still true.

Go back to my colleague with the invented job title. Point one of these tools at that documentation. The tool reads the meeting note, builds an edge from the person to the wrong title, and tags that edge extracted, and the tag is correct, because the edge genuinely is in the file. Then, the commit hook fires and rebuilds it. Then, it fires again tomorrow. The artifact gets fresher every single day while the answer stays wrong, and it looks more trustworthy each time, because the timestamp keeps moving and nothing else does.

I went looking for the vocabulary of that problem in Graphify's architecture document (the ARCHITECTURE.md file in its repository, as I read it in September 2026) [21]. Last verified: not once. Stale: not once. Superseded,

authority, precedence, curated, outdated, conflict: not one occurrence of any of them, anywhere in the file, as of this writing. I am not saying that as a criticism. I read their absence as a sign that the things they describe were never in the source, and a program that only reads source has nowhere to get them from. Graphify's README does describe more than parsing: a work memory that records how past answers turned out, tags what it learned as preferred, tentative, or contested, and flags a lesson for re-verification when the code behind it changes [21]. That is real progress on freshness. It is still learned from how answers turned out, not decided by the person who owns the rule, and that difference is the one this chapter is about.

Which is the distinction the whole chapter rests on, so let me put it as plainly as I can. The map is anything you can derive from the artifacts. Delete it, and you have lost an afternoon, not knowledge, because you can rebuild it from what is still there. Now, delete the rule that says never sell inside the blackout window, or the line that says last year's allocation is dead, or the note that this person proposes and does not execute. Nothing rebuilds those. A tool can read them once they are written down, and link them to the code, and it should [21, 29]. What it cannot do is decide them, or invent them when nobody has. People settled those decisions in a conversation, and somebody has to write them down. If there is no record, I leave the gap visible until the person responsible fills it.

Running the pattern in production

The week I wrote this chapter, I wired one of those parsers into my own conversion pipeline as a default step [21, 29], with an installer that only gives up when installation is provably impossible, and loudly when it does. The difference was not subtle. On one production service, the generated index went from a few hundred records my hand-written extractors could find to two thousand seven hundred [27], every function and class in the codebase with a verified line number, plus a dependency graph with seventeen thousand edges that answers what imports this and what breaks if I change it. And remember the tags from a moment ago: extracted, inferred, ambiguous. In my pipeline they stopped being documentation and became routing. Extracted records enter the index only after a citation gate checks the line they claim. Inferred and unverifiable ones go to a quarantine file a human has to

promote. The parser's honesty about its own confidence turned out to be the most useful thing it publishes, once something downstream actually acts on it.

Retrieval over that bigger map deserves a word, because two thousand seven hundred records sounds like it needs a search engine, and it does not. The index is one file. A question about any symbol, endpoint, or configuration is one scan of that file, and the row hands back a kind, a name, and a file and line the gate already checked, so the assistant opens the file at that line and reads the real thing instead of a summary of it. Impact questions go to the dependency edges the parser produced: what imports this, what breaks if I change it, which is a lookup rather than a guess. Everything else follows the same ladder the finance agent used, cheapest first. The session starts from a small orientation tier the build keeps under a measured byte budget, the knowledge map routes topic questions to documents, and full-text search is the last resort instead of the first move. There is still no vector store anywhere in this.

The part I did not expect is what verification does to retrieval. When every row in the index has been checked against the line it cites, following the map stops being an act of faith. And the records that could not be verified are sitting in quarantine, where the assistant cannot retrieve them by accident. Retrieval quality turned out to be less about finding more and more about refusing to serve what you cannot defend.

The same pipeline now ends by scoring the repository it just converted. Five levels: orientation, hygiene, instruction layer, knowledge layer, governed autonomy, each gated on the one below it, the same way Factory's agent readiness model gates its maturity levels [23], except every criterion here is a checkable filesystem fact rather than a model's judgment. A `README` is there, or it is not. The index is verified, or it is not. What I like most is where repositories actually land. The two services I scored the week I wrote this both passed the instruction and knowledge layers in one run. Both stalled at hygiene for the same mundane reason: a missing ownership file that the pipeline had already drafted but neither team had yet reviewed and renamed. And the top level does not pass on day one, because the rubric requires at least one written session log, and the pipeline only creates the empty folder. The first log appears only after someone has actually worked in the repository. In

the end the rubric agrees with the chapter. You can generate the map this afternoon. The brain has to be lived in. Both the pipeline and the scoring live in the framework repository if you want to run them on your own code [24]. The scoring script is `readiness_report.py`, where you can read the exact checks.

The week after I scored those two services, the map outgrew the single repository. A group of services has knowledge that no one repository can hold: which service imports which library, which infrastructure repository deploys which service, which configuration file quietly names a sibling it depends on. So, the same discipline went up a level. A discovery script walks the whole group and proposes a roster, every repository starts as observe, indexed but never written to, until a person promotes it, and the conversions themselves run through the ticket pipeline as reviewed merge requests instead of local edits on my machine. What comes out the other side is a registry with one measured row per member, a router file that points at each repository's own knowledge rather than copying it, and a cross-repo graph where every edge carries the resolver that produced it (one for each of those three kinds of link), a confidence tag, and the file and line it came from. Anything that looks like a cross-repository reference but cannot be proven lands in a needs-verification sidecar instead of the graph, and a person decides whether to promote it.

The router cap is the part I will defend the hardest. The project file a session eager-loads has a hard size limit, 64 KB, because the first thing I tried was a hand-maintained mega graph that inlined every member's knowledge, and it was stale the day I finished writing it. Point, do not copy. The fleet tooling lives in the same framework repository [24].

The first full run of that fleet conversion is a night I am still thinking about. I pointed the pipeline at a group of sixty-five repositories after dinner and let it work: fifty-eight tickets, fifty-one merge requests by morning, each one a gated change with a verified knowledge layer behind it. Somewhere around two in the morning I noticed the queue was not waiting on the agents at all. Every merge request was waiting on me. Eighteen months of tuning had moved the bottleneck from delivery to approval, and the constraint left in the system was the person who built it.

What changed that night was not the tooling; it was my job description. I stopped standing in the loop, approving every change as it arrived, and moved

onto the loop: set the gates up front, let verified changes flow, sample and audit instead of inspecting everything, and route every correction through a comment the pipeline can ingest rather than a push of my own. I wrote up the night, the operating model it forced, and the architecture underneath it in three pieces the same week [25, 26, 27].

So, let the tool build the map and keep it current for nothing. What is left over is the part you are being paid for, and it is the harder half, which is the brain. I want to explain how I build that knowledge base and keep it trustworthy as it grows.

How to build the brain (the knowledge base)

In the financial agent example, I did not build the knowledge. I built a map that guides the assistant to the right files. The other half, building the knowledge base itself and then growing it from junior to senior, is the work I have been doing on real repositories for a while now, and it is the part I am still learning [9, 10, 16]. It leans heavily on a coding assistant, Claude Code, Devin, or Cursor, to do the reading. It works by crawling the repository and learning the domain, much the way we did in *Chapter 2* but pointed at code instead of documents. The deeper it reads, the more it adds, and the more senior the agent gets. Findings that pass the evidence checks become part of the knowledge base, so later sessions can use them without repeating the same investigation [27]. I am describing the shape of that practice rather than handing you a recipe, because it has changed three times since I started and I expect it to change again. The financial agent and the chapters before it are the intuition you want first.

A fair question at this point is how much of this I automate, because I now run it as a conversion pipeline I point at any repository [16]. Concretely, a conversion leaves files you will recognize from the finance folder: instruction files and a START_HERE entry point read every session, a knowledge graph and a document index as the map, a code index where every row cites its file and line, a Source of Truth folder for the decisions people own, and a progress tracker with session logs [24]. The pipeline generates the map and drafts knowledge from evidence, but it does not decide the project's vision [27]. Everything a parser can defend gets derived and checked: the endpoints, the dependency edges, the likely owners from commit history, and an index

where every row is verified against the line it cites. Where a decision is missing, it leaves a template to complete or a clearly marked proposal to review. The vision file is a template until a human fills it. The ownership file it writes is a proposal the team has to review and rename before it counts, because Git shows who committed to each file, but ownership itself is a decision. That line, derive everything you can defend and refuse to fabricate the rest, is the whole method compressed into a build script.

I can also tell you what the rest of the industry converged on, because they arrived at the same boundary from the other side. Some teams building agent memory spent 2026 retreating from automated rewrites. In April 2026 Mem0 replaced the pass that reconciled new facts against old ones with a single extraction that only adds, so a new fact now lives alongside the one it supersedes instead of overwriting it [31]. The research lane hit the same wall and named it context collapse, an agent compressing eighteen thousand tokens of accumulated playbook into a hundred and twenty-two in a single self-rewrite, ending up worse than no playbook at all [32]. So, the recipe that survives is short. Let the machine draft from real work. Append and supersede, never overwrite. Date every entry and let staleness nag. Gate every write on evidence. Keep one human click between proposed and true, because an ungated self-writing brain is no longer just a quality risk; it is a documented attack surface [33].

Building the brain is half of it. The other half is making sure each session actually starts from what the last one learned, and that comes down to a small ritual I run every time.

Maintaining session continuity

This is the move I use every day, and it is the same one the chapter opened with. On the real repository, the ritual is a small number of reads, every session, before any work begins: the tier-zero file for the rules and the navigation, the index for the map, and a progress note for where we left off. At the end of the session, the progress note is rewritten with what got done, what is in flight, and the next few priorities. It took me a while to get this down to something I would actually keep doing. The version I use now lives in the framework rather than in my habits, so the session opens by reading the entry file, then the knowledge graph, then the progress tracker, and the

assistant does that before it is allowed to touch anything. Here is roughly what the tracker says when a session opens:

```
# Progress - repo conversion
Current state: index rebuilt and verified, 2,700 rows.
Blockers: hygiene gate red, no ownership file in the repo.
Next: write the ownership file, re-score, then roster the rest of the group.
```

Four lines, and the assistant is caught up. In between, I do not touch it, because the writing is a command at the end of the session rather than something I have to remember: it writes the session log, updates the tracker from it, and lists the next three priorities specifically enough that the next session can act on them without asking me. The next session opens by reading it back.

The effect is hard to oversell. The team stopped re-introducing the project to the assistant, the way you stop re-introducing yourself to a colleague who sat in yesterday's meeting. You just continue.

All of this, the brain, the map, the ritual, and the discipline of reading only what you must, adds up to a kit. And there is one kind of engineer whose whole job that kit is built for.

The forward-deployed engineer

There is a person this whole method is really for, and the industry has finally settled on a name for them. Years ago, I led a small team we nicknamed the impossible missions force. We took the work everyone else had written off, and we got it done. I had no title for it then, but we were already doing forward-deployed engineering; the name had not caught up yet. A forward-deployed engineer is dropped into a system they did not build and is expected to be useful fast, and in 2026 that role went from a curiosity to one of the most sought-after jobs in the field [30], because the bottleneck stopped being the model and became deployment, the work of making a capable model useful inside one specific, messy, real codebase.

Everything in this chapter is the kit for that job. Dropped into an unfamiliar repository, you do not try to memorize it. You give it a brain and a map, and you let the assistant come up senior on the first day. You watch before you touch, you ship something small and correct early, and you let the map carry the context instead of your opinions. You leave the context behind you when

you move on. The failure mode that separates a forward-deployed engineer from a hired gun is knowledge hoarding; the goal is to make yourself unnecessary, not indispensable, and a knowledge base your tools can read is exactly how you manage that [17, 18].

Starting your own map

Where you go from here is simple enough, and you should do it in this order.

1. Write the tier-zero file first, and keep it short. The rules you always want followed, the order of precedence when two files disagree, and how to navigate. If it runs past a page, you have put something in it that belongs a tier down.
2. Make the directories, because they are doing more work than they look like they are: rules, entities, facts, decisions, archive. The names carry the kind, and the order carries the precedence, and you will not have to write either of those down as a field.
3. Put three fields at the top of each file. Name, description, last verified. Write the description for somebody deciding whether to open the file, because that is the only reader it has.
4. Generate the index instead of writing it, even at ten files, even though hand-writing it would be faster today. Not for the time saved. For the habit, because the version of you in six months has three ways for files to arrive and no memory of this paragraph.
5. Add the validator, and make it fail the build.
6. When it outgrows a folder you can read in one sitting, tier the index the way you tiered the corpus.

What I run in production is the same shape at a size that punishes a bad map, and I should be honest that it is still moving. I have been revising it since 2023, based on whatever broke most recently, and I will still be revising it after this book is printed. That is not a disclaimer; it is what a practice looks like as opposed to a finished product. Take the tiers and take the discipline. Let the file names be yours.

Applying the pattern across the four use cases

If you have been following one of the book's four case studies rather than reading front to back, here is where this chapter lands for each of them.

The personal assistant got the full treatment, because the finance folder is one: personal documents and personal rules, one person's history. Every demo in the chapter ran on it. The one thing I would add for calendar and email work is that most of that data is live, pulled through a connector at answer time, so the map governs the rules and notes around the live data rather than the inbox itself. `AGENT.md` rides along on every session either way.

For a domain-specific assistant, the chapter's domain is already yours: the persona is a small financial system. What changes at professional scale is who writes tier zero. The blackout window was one personal compliance rule. A real advisory product carries a regulator's worth of them, far too many to sit resident, so what belongs in the always-resident tier is the rule that names which policy to check, the same shape the blackout rule used, because a relevance scorer will skip the rule that makes the answer legal. The detail lives one hop down, loaded by product and jurisdiction. Structured schemas take over the facts tier at that scale, meaning the facts sit in database tables the map points at instead of in markdown files; precedence and the last-verified date transfer unchanged.

The enterprise assistant is where the token argument stops being academic. An enterprise corpus is where the flat index hits the ceiling that *Table 11.1* warns about, and past it the design changes: a short resident root index, one branch opened on demand. The entities tier earns its keep here more than anywhere, because "who owns this service" is the most common question in a large organization and no single document is about it. That was the surprise I flagged on the engineering repository, and it holds for the org chart as much as for the codebase.

Content moderation looks like the odd one out, and for the queue it is. Items flow through at decision time, and nothing about them belongs in an index. The policy store is another matter. The rule that selects policy is tier zero, resident on every decision rather than retrieved when it seems relevant, with the detailed policy one hop away by content class, and it needs the archive discipline too, because "which policy was in force when this was posted" is the same temporal precedence question the 2025 allocation answered. The

blackout demo is the moderation failure in miniature: the rule that scored zero on relevance was the one that made the action safe.

Summary

We started with an assistant that begins every session as a junior, and we ended with one that starts senior because the knowledge lives in files it can navigate. In between, the whole method fit in a folder small enough to read in one sitting: a brain of plain markdown and a map generated over it, three tiers so the assistant reads what it cannot afford to miss and reaches for the rest, precedence and a last-verified date so facts retire without deleting history, rules resident instead of retrieved, and a file per entity for the people and systems no single document is about. Then, we moved the same shape onto an engineering repository, let the parsers build the map, and kept the judgment, the half no parser can regenerate for you. Choosing the smallest, useful context for each task is context engineering, and you have now done it end to end.

Which brings you back to the spine of this whole book. The model is rentable. Anyone can call the same API you can. What you actually own, the thing that compounds and cannot be copied out from under you, is your data and how well you have organized it for the model to use. The map, the history, the rules: that is the moat.

If you take one thing from this book rather than the whole method, take the smallest one. It is not the token savings, which are modest until you are large. It is that an assistant with a curated map can tell you when it is not sure, and an assistant searching a pile of files can hand you a wrong answer with a perfect citation attached. I have been on the receiving end of both. Only one of them cost me anything.

References

- [1] Sumers, T., Yao, S., Narasimhan, K., & Griffiths, T. (2023). *Cognitive Architectures for Language Agents (CoALA)*. **arXiv**:2309.02427 (published in TMLR, 2024). https://arxiv.org/abs/2309.02427
- [2] ElHousieny, R. (2026). *Building an AI-Powered Markdown Knowledge Base System for Your Engineering Team*. Clearwater Engineering Blog

(Medium). https://medium.com/cwan-engineering/building-an-ai-powered-markdown-knowledge-base-system-for-your-engineering-team-4bccea3cdbfe

- [3] ElHousieny, R. (2026, January 9). *From Knowledge Graphs to Smart Search: Making AI Assistants Actually Fast*. LinkedIn. https://www.linkedin.com/pulse/from-knowledge-graphs-smart-search-making-ai-actually-rany-lniec/
- [4] ElHousieny, R. (2026, January). *How Markdown Knowledge Graphs Transform GenAI Code Assistants from Guessing to Knowing*. LinkedIn. https://www.linkedin.com/pulse/copy-how-knowledge-graphs-transform-genai-code-from-rany-jml7c/
- [5] Orosz, G. (2025). *Building Claude Code with Boris Cherny*. The Pragmatic Engineer. https://newsletter.pragmaticengineer.com/p/building-claude-code-with-boris-cherny
- [6] VectifyAI. (2025, September). PageIndex: Vectorless, Reasoning-based RAG. https://github.com/VectifyAI/PageIndex
- [7] Rasmussen, P., et al. (2025). *Zep: A Temporal Knowledge Graph Architecture for Agent Memory*. arXiv:2501.13956. https://arxiv.org/abs/2501.13956; *Graphiti*: https://github.com/getzep/Graphiti
- [8] ElHousieny, R. (2026). *The CLAUDE.md File: Your project's AI Brain*. LinkedIn. https://www.linkedin.com/pulse/claudemd-file-your-projects-ai-brain-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-ywylc/
- [9] ElHousieny, R. (2026). *Agentic Repos: The Framework That Turns Any Repository into a Senior AI Development Environment*. LinkedIn. https://www.linkedin.com/pulse/agentic-repos-framework-turns-any-repository-senior-rany-ilsyc/
- [10] ElHousieny, R. (2026, July 19). *Growing AI Agents from Junior to Senior*. LinkedIn. https://www.linkedin.com/pulse/growing-ai-agents-from-junior-senior-engineering-work-rany-tai6c/
- [11] ElHousieny, R. (2026). *You Are Not Behind on AI. Your Repository Is*. Medium. https://medium.com/@ranyel/you-are-not-behind-on-ai-your-

repository-is-784a0921e00b

- [12] Anthropic. (2025, September 29). *Effective context engineering for AI agents*. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
- [13] Karpathy, A. (2025, June 25). *Endorsement of "context engineering" over "prompt engineering."* X. https://x.com/karpathy/status/1937902205765607626
- [14] ElHousieny, R. (2026). *Context Window Management Skills Nobody Teaches You*. LinkedIn. https://www.linkedin.com/pulse/context-window-management-skills-nobody-teaches-you-rany-sebyc/
- [15] Anthropic. (2024, September 19). *Introducing Contextual Retrieval. Anthropic Engineering*. https://www.anthropic.com/engineering/contextual-retrieval
- [16] ElHousieny, R. (2026, July). *How to Turn any repository into a Senior AI-Powered Development Environment*. Clearwater Engineering Blog (Medium). https://medium.com/cwan-engineering/how-to-turn-any-repository-into-a-senior-ai-powered-development-environment-fe9ad4f616d6
- [17] ElHousieny, R. (2026, May 30). *What an AI Forward Deployed Engineer Really Is*. LinkedIn. https://www.linkedin.com/pulse/what-forward-deployed-engineer-really-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-buunc
- [18] ElHousieny, R. (2026). *The Skill That Will Define Your Career in the AI Era*. Medium. https://ranyel.medium.com/the-skill-that-will-define-your-career-in-the-ai-era-004e2e32e0d5
- [19] Sklaroff, J. (2024, September 23). *Constructing the Ultimate Gen AI Chat/Copilot Experience: Part 1*. Clearwater Engineering Blog (Medium). https://medium.com/cwan-engineering/constructing-the-ultimate-gen-ai-chat-copilot-experience-part-1-55c2e07a1feb
- [20] Karpathy, A. (2026, April 4). llm-wiki. GitHub Gist. https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f
- [21] Shamsi, S. (2026). *Graphify: turn any codebase, with its docs, SQL schemas, configs, and PDFs, into a queryable knowledge graph*. GitHub. https://github.com/Graphify-Labs/graphify (accessed August 15, 2026)

- [22] Pandey, S. (2026, July 27). *Andrej Karpathy Asked for a Tool. 48 Hours Later, Graphify Went Viral*. Towards Deep Learning (Medium). https://www.towardsdeeplearning.com/andrej-karpathy-asked-for-a-tool-48-hours-later-graphify-went-viral-10d8ead5f50e
- [23] Factory. (2026). *Agent Readiness*. Factory Documentation. https://docs.factory.ai/agent-readiness/overview (accessed August 15, 2026)
- [24] ElHousieny, R. (2026). *Agentic-Repo: the Agentic Repos framework*. GitHub. https://github.com/ranyelhousieny/Agentic-Repo
- [25] ElHousieny, R. (2026, August 16). *The New Engineering Bottleneck Isn't AI. It's You*. LinkedIn. https://www.linkedin.com/pulse/new-engineering-bottleneck-isnt-ai-its-you-rany-elhousieny-phd%E1%B4%AC%E1%B4%AE%E1%B4%B0-wlo4c/
- [26] ElHousieny, R. (2026, August 16). *Stop Being Human in the Loop. Start Being Human on the Loop*. Level Up Coding (Medium). https://medium.com/gitconnected/stop-being-human-in-the-loop-start-being-human-on-the-loop-2ca06d2eeeb7
- [27] ElHousieny, R. (2026, August 16). *Anatomy of an Agentic Repo: The Architecture That Stops AI From Guessing*. Level Up Coding (Medium). https://medium.com/gitconnected/anatomy-of-an-agentic-repo-the-architecture-that-stops-ai-from-guessing-f3105c6367b8
- [28] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). *Lost in the Middle: How Language Models Use Long Contexts*. Transactions of the Association for Computational Linguistics, 12, 157-173. https://doi.org/10.1162/tacl_a_00638
- [29] Shamsi, S. (2026). *Graphify* (author repository; redirects to the Graphify-Labs mirror). GitHub. https://github.com/safishamsi/graphify (accessed August 24, 2026)
- [30] Bellan, R. (2026, July 30). *Forward-deployed engineers are the AI industry's latest talent obsession*. TechCrunch. https://techcrunch.com/2026/07/30/forward-deployed-engineers-are-the-ai-industrys-latest-talent-obsession/
- [31] Yadav, D. (2026, April 16). *Introducing the token-efficient memory algorithm*. Mem0 Blog. https://mem0.ai/blog/mem0-the-token-efficient-

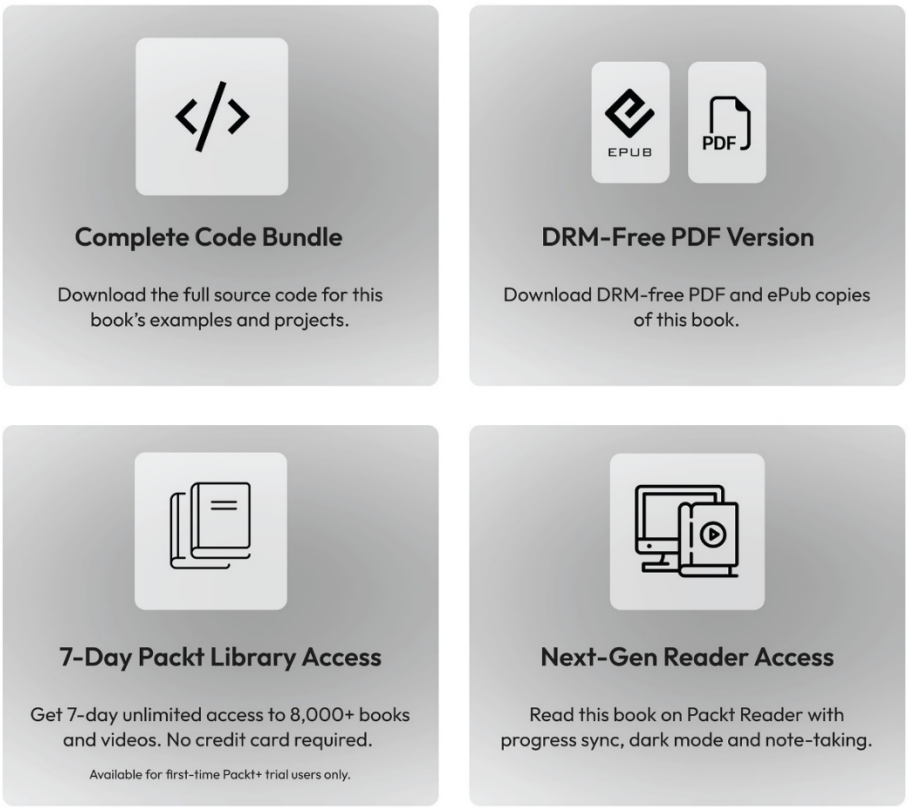
memory-algorithm

- [32] Zhang, Q., Hu, C., Upasani, S., et al. (2025). *Agentic Context Engineering: Evolving Contexts for Self-improving Language Models* (ICLR 2026). arXiv:2510.04618. https://arxiv.org/abs/2510.04618
- [33] Chen, Z., Xiang, Z., Xiao, C., Song, D., & Li, B. (2024). *AgentPoison: Red-teaming LLM Agents via Poisoning Memory or Knowledge Bases*. arXiv:2407.12784. https://arxiv.org/abs/2407.12784
- [34] Panigrahi, D. (2026, August 20). Schema Drift Reaches the Tool Definition. Nexla Blog. https://nexla.com/blog/context-drift-reaches-the-tool-definition/

12

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

../../EPUB/NOTE

Note: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 12.1* on desktop), search for this book by name and select the correct edition.

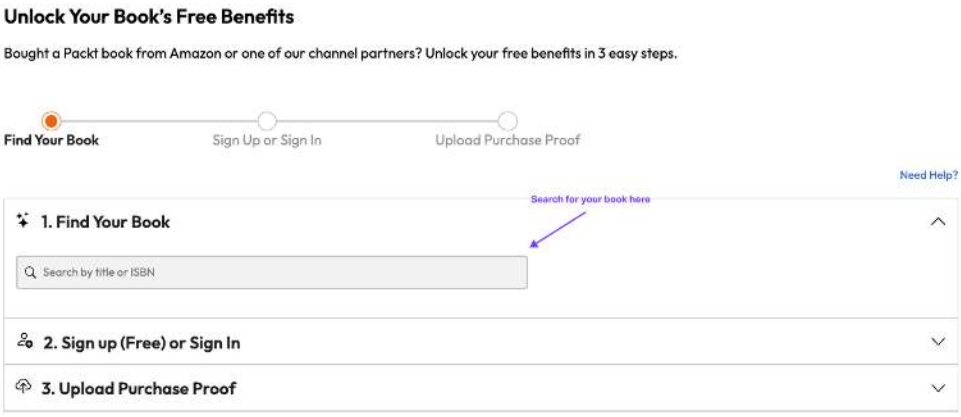


Figure 12.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit https://www.packtpub.com/unlock-benefits/help for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.



../../EPUB/NOTE

Note: If you are still facing issues, reach out to customercare@packt.com.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

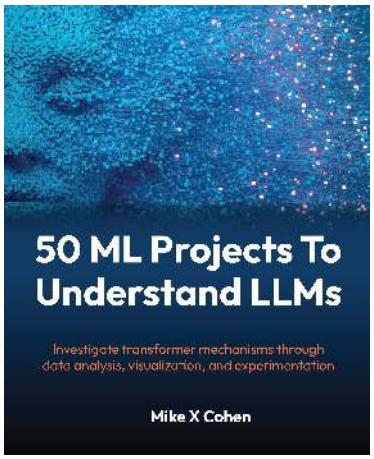
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

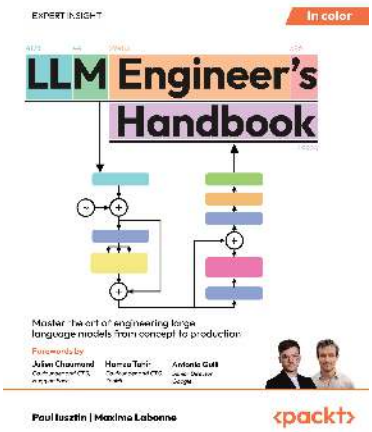


50 ML Projects To Understand LLMs

Mike X Cohen

ISBN: 9781808082559

- Tokenization schemes and their statistical properties
- Embedding spaces: cosine similarity, semantic axes, and analogy vectors
- Output logits, softmax distributions, perplexity, and language biases
- Layer-by-layer transformer dynamics and dimensionality
- Attention mechanisms: QKV weights, attention scores, head ablation, and activation patching
- MLP subblocks: neuron tuning, mutual information, subspace analysis, and statistics-based causal manipulations
- Logit lens, indirect object identification, and causal tracing



LLM Engineer's Handbook

Paul Iusztin, Maxime Labonne

ISBN: 9781836200079

- Implement robust data pipelines and manage LLM training cycles
- Create your own LLM and refine it with the help of hands-on examples
- Get started with LLMOps by diving into core MLOps principles such as orchestrators and prompt monitoring
- Perform supervised fine-tuning and LLM evaluation
- Deploy end-to-end LLM solutions using AWS and other tools
- Design scalable and modularLLM systems
- Learn about RAG applications by building a feature and inference pipeline

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Data Strategy for LLMs*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/1806116995

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

ARC (AI2 Reasoning Challenge) 67

Adaptive RAG 152

Agentic Chunking 100

Amazon Bedrock AgentCore 128

Apple iPhone 76

affected data
rebuilding 334, 335

agent rules 384, 385

artificially generated datasets 164

automated judges
using, for scale 336

B

BIG-Bench 67

BLEU 278

BLOOM model 65

Benchmarking Information Retrieval (BEIR) 149

BookCorpus 39, 40, 65

C

C4 (Colossal Clean Crawled Corpus) 65

CRAG (Corrective RAG) 152

ChainPoll 308

ChromaDB 139

Common Crawl **33**, **64**

Content Moderation Assistant **104**

chunking **92**
- context-aware chunking 101
- fixed-size chunking 93, 94
- hierarchical chunking 101
- LLM-driven chunking 100
- semantic chunking 99, 100
- sentence-based chunking 94, 95

cloud-native vector options **129**

code corpora **65**

code implementation **84**–**88**

context engineering **393**

contextual chunking **147**

continuous integration (CI) pipeline **290**

cosine similarity **83**

customization data
- behavioral data 236–238
- building 232, 233
- data quality requirements 243–245
- feedback data 239–241
- knowledge data 233–236
- preference data 239–241
- synthetic data 242

customization patterns
- for AI assistants 270

customization workflow
- customization data scaling 251, 252
- customization life cycle 245, 246
- data connection 246

designing 245
evaluation, as gating function 254, 255
inference-time controls 248–250
post-training 253
retrieval and grounding 250, 251

customization workflow implementation **256**
customization data curation 259–263
customized assistant, deploying 268, 269
few-shot example selection 263, 264
inference-time controls, assembling 265, 266
retrieval and grounding, integrating 267
target behaviors, defining 256–258

D

Domain-Specific Assistant **102, 106**

data augmentation for robustness **179**
enterprise example 181
personal assistant example 181
techniques 180

data extraction **61, 62**

data failures
root cause, correcting 332, 333
tracing, to sources 330, 331

data preparation strategy **108, 109**

data quality
gates, enforcing 349, 350
loop, closing 355–358
metadata 350

monitoring 352–354
problems, identifying 327–329
process, calibrating 338, 339
provenance 351

data scarcity
managing 217

data sources **31**
for assistant use cases 70–72

data versioning **351, 352**

direct preference optimization (DPO) **176**

domain grounding **43**

domain specialization dimensions **227**
compliance requirements 232
decision-making 229
knowledge 228
policies 231
terminology 228
workflows 229–231

domain-specific content **42**
applications 55, 56
benefits 53
best practices 54, 55
challenges 54
considerations 54
emails and internal communication 43
financial domain data 45–47
internal documents 43
legal domain data 47–49
medical domain data 44, 45
meeting notes and transcripts 43
scientific research data 49, 50
software development data 51–53

Wikis and internal knowledge bases 43

E

End-to-End mini pipeline **95** – **98**

Enterprise Knowledge Assistant **102**, **107**

embedding model **83**
selecting 78

embedding space **83**

embeddings **76** – **78**
practical example 79 – 81

entity-relationship extraction **142**

environment setup, OpenAI API
automated local setup 12
for macOS users 12 – 15
for Windows users 13 – 15
Google Colab 8 – 11

evaluation metrics **314**
content moderation system metrics 318
domain-specific assistant metrics 317, 318
enterprise knowledge assistant metrics 318
general metrics 314, 315
metric selection framework 318
personal assistant metrics 317
prompt comparison 315

evaluation platforms **319**
core capabilities 319
cost at scale evaluation 320
criteria 321
differentiation 320
integration and developer experience 319
lock-in and portability 320
production readiness 320

F

fine-tuned model evaluation **301** – **305**

fixed-size chunking **93**, **94**

flat index **392**

forward-deployed engineer **401**

G

GSM8K (Grade School Math 8K) **67**

GitNexus **155**

Google Colab **8** – **11**

GraphRAG community **142**

graph database **103** – **105**

integration 102, 103

H

HTML documents **56**

best practices 62

challenges of parsing 56

parsing 57 – 59

processing techniques 57

HellaSwag **66**

HumanEval **67**

hierarchical chunking **101**

human oversight

scaling 335

significance 187

human review

prioritizing 337, 338

human-in-the-loop strategies **187**

enterprise example 188

human-in-the-loop workflows

visualizing 188

humans integration, into synthetic data workflows

patterns 187, 188

I

in-context learning **19–24**

indexing pipeline **83**
chunked ingestion upgrade 89–91

inference-time controls **193**
few-shot examples 194, 195
instruction structure 194
post training 197
retrieval, as behavioral control 195, 196
tools, for actions and computation 196

K

knowledge map **367–380**

L

LLM data
infrastructure decisions 345–348
source and derived data, separating 341–343
storing 340, 341

LLM to Intelligent Assistant transition **25–27**

LLM-as-a-judge **277–279**
example 277
golden dataset, building 280
prompt comparison 280
prompts, running through evaluation 281–
function 284
regression report, reading 284, 285

LLM-driven chunking **100**

LangChain
using, in RAG system 123–125

LoRAX (LoRA eXchange) **304**

Low-Rank Adaptation (LoRA) **200**
models, adapting with 210–213

licensed data **68**
benefits 68
types 69
usage guidance 69

live production data evaluation **306**

M

Managed RAG **126**
cost 127
enterprise search options 126, 127
trade-offs 127

Massive Text Embedding Benchmark (MTEB) **149**

massive multitask language understanding (MMLU) **66**

metadata **92, 101**
enrichment 107

model customization **224, 225**
as System-Level Process 225, 226
data, as foundation 226, 227

multi-metric judge **315**
basic score, comparing with 316
interpreting 317

multilingual corpora **65**

multimodal augmentation workflows
building 185, 186

multimodal data augmentation **183**
domain example 184

significance 184
techniques 184

N

Naive RAG **133**
keyword exact match retrieval, adding 137 – 140
multi-hop failures, handling 141 – 143
second source, adding 140 , 141
temporal data drift, correcting 135 , 136
temporal data drift, detecting 135 , 136

Naive RAG data architecture
access-controlled sources 145
building 144
chunking code 147
chunking strategy 146
code retrieval 153 – 155
contextual chunking 147
cost buckets 149
entity links storing 146
governance 150
index architecture, designing 148
routing 151 , 152
schema 145
self-correcting retrieval 152 , 153
structure-aware chunking 146
use cases comparison 156

National Association of Insurance Commissioners (NAIC) **278**

O

OpenAI API
calling 16 – 18
core AI function 18 , 19
environment setup 8

setting up 5 – 7

OpenWebText 37

optical character recognition (OCR) 60

overfitting 217

P

PDF documents 56
best practices 63
challenges of parsing 57
parsing 59
processing techniques 57

Personal Productivity Assistant 102 , 107

Predibase 304

Project Gutenberg 36

Prompt A responses
scoring 316

Prompt B responses
scoring 316

PubMed 44

pattern
applying, across domains 394 – 401
applying, across four use cases 402

people and systems 386 , 387

post training 192 , 193
integrating, into learning loops 218 , 219

post-training spectrum 204

post-training techniques 197
comparing 204
Low-Rank Adaptation (LoRA) 200
preference alignment 201
reinforcement learning 201 – 204
supervised fine-tuning (SFT) 199

preference alignment data 176

domain example 177
enterprise example 177
implementing 176

preference pairs 301
generating, programmatically 178, 179

production evaluation platform
applying 306–308

public datasets 63
benefits 64
leveraging 63
LLM benchmarking datasets 66, 67
LLM training datasets 64–66
selection 67

Q

query decomposition 143

question-answer generation 166
domain example 168
enterprise example 168
implementing 168–170
significance 167

R

RAG pipeline evaluation 286
ChromaDB, loading 286
combined report, reading 292
full pipeline, running 288, 289
golden dataset, building 288
RAGAS automated metrics, running 290, 291

RAG sources 75

RAPTOR 152

ROOTS 65

ROUGE 278

RefinedWeb 65

Retrieval Pipeline **117**
Q1/Q2 example 121 – 123
RAG Function, coding 117 – 120

Retrieval-Augmented Generation (RAG) system **115**
Indexing Pipeline 117
LangChain, using 123 – 125
Retrieval Pipeline 117

real data
limitations 164

reinforcement learning **202**
agentic behavior, optimizing with 213 – 216

reinforcement learning from human feedback (RLHF) **176**

relevant data sources **70**
best practices 71 , 72
for assistant use cases 70
for content moderator 71
for domain-specific assistants 70
for enterprise assistants 71
for personal assistants 70
selecting 70

retrieval-augmented generation (RAG) pipeline **171**

S

Self-RAG **152**

StarCoder Training Dataset **65**

semantic chunking **94** , **99** , **100**

sentence-based chunking **94** , **95**
End-to-End mini pipeline 95 – 98

similarity numbers **81** , **82**

structure-aware chunking **146**

structured data **170**
enterprise example 172, 173
in enterprise systems 170, 171
transforming 171, 172

supervised fine-tuning (SFT) **199**
applying 205–207, 210

synthetic QA generation
from structured data 173–175
techniques 167

synthetic data **164**
as strategic complement 165
complementary roles, visualizing in training 166
domain example 166
enterprise example 165
purpose 165
significance 164, 165

synthetic data evaluation **292**
answerability 298
diversity 299, 300
faithfulness 293–297
position bias check 301
preference pair 301

T

The Pile **65**

TruthfulQA **67**

task-specific evaluation datasets
building 308
data sourcing 309
golden dataset, creating 311
high-quality examples 312
schema 310, 311

sizes, determining 312, 313
starter sets, building 313, 314
task-specific evaluation 308
temporal data drift, Naive RAG
correcting 135, 136
detecting 135, 136
text augmentation techniques
implementing 181–183

V

vector **76**, **83**
vector database
selecting 128, 129
vector database optimization **105**
chunk size 106
data ingestion 107
indexing pipeline 107
overlap 106
vector indexes
managing 343–345
vectorization **76**–**78**

W

Wikipedia **35**, **36**
web data **32**, **33**
forums and discussion boards 33
online reviews 33
social media 33
webpages 33
web data collections **33**

best practices 41, 42
BookCorpus 39, 40
challenges 40
Common Crawl 33
OpenWebText 37
Project Gutenberg 36
Wikipedia 35, 36

web-based corpora
selection 64

Untitled

- Preface
  - Free benefits with your book
- Part 1: Building the Data Foundation
- Chapter 1: Getting Started with LLMs: From First API Call to Data Strategy
  - Technical requirements
    - How this chapter fits into the book
  - Getting started: setting up OpenAI API
    - Environment setup
      - Option 1: Google Colab (recommended for beginners)
      - Option 2: Automated Local Setup (Optional for advanced users)
      - For macOS users
      - For Windows users
      - For both Windows and Mac users:
      - Calling the OpenAI API
    - Understanding our core AI function
  - In-context learning
  - The journey: From basic LLM to Intelligent Assistant
    - Stage 1: Direct LLM Integration (Chapter 1)
    - Stage 2: Data Collection (Chapter 2): Building Your Knowledge Foundation
    - Stage 3: Data Preparation (Chapter 3): From Raw Data to Vector-Ready Knowledge
    - Stage 4: Retrieval-augmented generation (Chapters 4 and 5): building and refining retrieval systems

Stage 5: Synthetic Data Generation (Chapter 6): Enhancing Your Knowledge Base
Stage 6: Post-training and Model Customization (Chapters 7 and 8): Changing Model Behavior Through Data
Stage 7: Evaluation Data (Chapter 9): Measuring Whether the System Works
Stage 8: Data Quality and Human Oversight (Chapter 10): Keeping Production Systems Reliable
Stage 9: Context Engineering (Chapter 11): Delivering the Right Data at Runtime
Summary
References
Chapter 2: Data Sources
Technical requirements
Using web data
Types of web data
Example web data collections
Common Crawl
Wikipedia
Project Gutenberg
OpenWebText
BookCorpus
Challenges in collecting web data
Best practices for collecting web data
Acquiring domain-specific content
Types of domain-specific content
Medical domain data
Financial domain data
Legal domain data
Scientific research data
Software development data

- Benefits of domain-specific content
- Challenges and considerations
- Best practices for using domain-specific content
- Applications of domain-specific content

Processing HTML and PDF documents
- Challenges of parsing HTML documents
- Challenges of parsing PDF documents
- HTML and PDF processing techniques
  - HTML parsing
  - PDF parsing
  - Optical character recognition (OCR)
  - Data extraction
- Best practices for processing HTML and PDF

Leveraging public datasets
- Benefits of public datasets
- LLM training datasets
- LLM benchmarking datasets
- Public dataset selection

Exploring licensed data
- Benefits of licensed data
- Common types of licensed data
- Guidance for using licensed data

Selecting relevant data sources
- Data sources for assistant use cases
  - Personal assistant
  - Domain-specific assistant
  - Enterprise assistant
  - Content moderator
- Best practices when selecting data sources

Summary

References

Chapter 3: Data Preparation: From Raw Data to Vector-Ready Knowledge
- Vectorization and embedding fundamentals: the language of AI
  - Choosing your first embedding model: OpenAI's text-embedding-3 series
  - Your first embedding: A practical example
  - A quick word on other embedding models
  - The indexing pipeline: building our knowledge catalog
  - The code: Step-by-Step implementation
  - Chunking
  - The naive approach: fixed-size chunking
  - A smarter approach: semantic chunking
  - LLM-driven chunking: the agentic approach
  - The final layer: hierarchical and context-aware chunking
- Graph database integration: connecting the dots
  - What is a graph database?
- Vector database optimization: from prototype to production
  - The Goldilocks problem: optimizing chunk size and overlap
  - Beyond the text: metadata enrichment
  - Data ingestion and indexing pipeline: getting it all together
- Putting theory into practice
- Summary
- Further reading and references

Part 2: Delivering Knowledge at Inference Time

Chapter 4: Building Your First RAG System: From Theory to Implementation

The two halves of a RAG system: indexing and retrieval
The retrieval Pipeline: finding and using knowledge
Coding our RAG function
Understanding the Q1/Q2 Example
Common pitfalls in first-time RAG builds
Graduating to a framework
Why use a Framework?
Managed RAG and enterprise search options
How to choose (practical guidance)
Cost and ops trade-offs (at a glance)
Choosing a vector database (Top 5)
Cloud-Native Vector options (AWS, Elastic, MongoDB)
Applying RAG to the four case studies
Summary
Chapter 5: Data Strategy for Advanced RAG: When Your Data Tells You What You Need
Technical requirements
Detecting and correcting temporal data drift
Adding keyword exact match retrieval
Adding a second source exposes volume and schema mismatches
Handling multi-hop failures when answers span multiple systems
Building your data architecture
Deciding what goes in the schema
Handling access control
Storing entity links for multi-hop retrieval
Choosing a chunking strategy
Structure-aware chunking
Contextual chunking
Chunking code

Designing index architecture
Understanding what it costs
Establishing governance
Routing and self-correcting retrieval
Retrieving code
Comparing the four use cases
Summary
References and further reading
Part 3: Shaping LLM Behavior with Data
Chapter 6: Synthetic Data Generation
What is synthetic data
Why synthetic data has become essential
Visualizing the role of synthetic data
Enterprise example: HR policy queries
Synthetic data as a strategic complement
Domain example: content moderation challenges
Visualizing complementary roles in training
Question–answer generation
Why QA generation matters
Techniques for synthetic QA generation
Enterprise example: HR policy FAQs
Domain example: medical knowledge expansion
Implementing QA generation in practice
Utilization of structured data
Structured data in enterprise systems
Transforming structured data
Enterprise example: HR policy conversion
Enterprise example: financial data transformation
Building synthetic QA from structured data
Preference alignment data
Synthetic preference alignment in action
Enterprise example: aligning HR responses for clarity

Domain example: moderation messages with the right tone
Generating preference pairs programmatically
Data augmentation for robustness
Techniques for Robustness
Enterprise example: overcoming OCR challenges
Personal assistant example: handling speech-to-text errors
Implementing text augmentation techniques
Multimodal data augmentation
Why multimodal augmentation matters
Techniques for multimodal augmentation
Domain example: content moderation across modalities
Building multimodal augmentation workflows
Human-in-the-loop strategies
Why human oversight matters
Hybrid approaches
Visualizing human-in-the-loop workflows
Enterprise example: legal review for compliance QA
Summary
References
Chapter 7: Understanding LLM Post Training
Technical requirements
The role of post training
Exhausting inference-time controls
Structuring instructions for behavior
Using few-shot examples
Retrieval as a behavioral control
Using tools for actions and computation
When is post training required?
The post-training spectrum

Supervised fine-tuning (SFT)
Low-Rank Adaptation (LoRA)
Preference Alignment and Reinforcement Learning
Comparing techniques in practice
Reading the Spectrum
Applying supervised fine-tuning
Adapting models with LoRA
Optimizing agentic behavior with RL
Managing data scarcity and overfitting
Integrating post-training into learning loops
Summary
References
Chapter 8: LLM Model Customization
Understanding model customization
Customization as a System-Level Process
Data as the foundation of customization
Dimensions of Domain Specialization
Knowledge and terminology
Workflows and decision making
Policies and compliance requirements
Building customization data
Knowledge data
Behavioral data
Preference and feedback data
Synthetic data
Data quality requirements
Designing the customization workflow
Understanding the customization life cycle
Connecting data to the customization workflow
Starting with inference-time controls
Expanding through retrieval and grounding
Scaling the customization data

Escalating to post-training
Using evaluation as a gating function
Implementing the Customization Workflow
Defining target behaviors
Curating customization data
Selecting few-shot examples
Assembling inference-time controls
Integrating retrieval and grounding
Deploying the customized assistant
Customization patterns for AI assistants
Summary
References
Part 4: Building and Maintaining Reliable LLM Systems
Chapter 9: Task-Specific Evaluation Data
Technical requirements
Using LLM-as-a-judge
Step 1: building the gold dataset
Step 2: defining two prompts to compare
Step 3: running both prompts through the evaluation function
Step 4: reading the regression report
Evaluating the RAG pipeline
Step 1: loading the ChromaDB from Chapter 3
Step 2: building the RAG golden dataset
Step 3: running the full pipeline and scoring
Step 4: reading the summary
Step 5: running RAGAS automated metrics
Step 6: reading the combined report
Evaluating and using Chapter 6's synthetic data
Faithfulness
Answerability
Diversity

- Preference pairs and the position bias check
- Evaluating the fine-tuned model from Chapter 7
- Evaluating live production data
  - Applying a production evaluation platform
- Building task-specific evaluation datasets
  - Understanding why task-specific evaluation matters
  - Sourcing data for evaluation sets
  - Defining the evaluation dataset schema
  - Creating the golden dataset
  - Prioritizing quality over quantity
  - Determining evaluation set size
  - Example: building starter sets for the four assistant types
- Selecting evaluation metrics by assistant type
  - Applying general metrics across all assistants
  - Seeing the difference in the notebook
    - Step 1: defining the multi-metric judge
    - Step 2: scoring prompt B responses
    - Step 3: scoring Prompt A responses
    - Step 4: comparing basic score vs.multi-metric score
    - Step 5: interpreting what the extra metrics caught
  - Personal assistant metrics (case study 1)
  - Domain-specific assistant metrics (case study 2: medical diagnosis)
  - Enterprise knowledge assistant metrics (case study 3)
  - Content moderation system metrics (case study 4)
  - Metric selection framework
- How to evaluate evaluation platforms
  - Core capabilities
  - Integration and developer experience
  - Production readiness

Evaluating costs at scale
Differentiation
Lock-in and portability
From criteria to decision
Summary
References
Chapter 10: Data Quality and Human Oversight
Identifying data quality problems
Diagnosing and correcting data failures
Tracing failures to their sources
Correcting the root cause
Rebuilding affected data
Scaling human oversight
Using automated judges for scale
Prioritizing human review
Calibrating the quality process
Storing LLM data
Separating source and derived data
Managing vector indexes
Selecting data infrastructure
Governing and monitoring data quality
Enforcing data quality gates
Tracking metadata and provenance
Data versioning
Monitoring data quality
Closing the data quality loop
Summary
References
Chapter 11: Context Engineering: Data Strategy at Agent Runtime
Technical requirements
The junior new session

- Understanding the knowledge map
  - Organizing into three tiers
  - Reading what a signpost has to say
  - The index writes itself
  - Walking it by hand
  - Encoding the same walk in Python
- Facts that expire
  - Keeping the dead file
- Rules the agent follows
  - Whether it actually obeys
- Representing people and systems
- Growing junior to senior
  - Generate, then govern
  - Understanding when tokens become critical
- The discipline of context engineering
- Applying the pattern across domains
  - Applying the same tiers to an engineering repository
  - Revealing the search the finance example hid
  - Using existing tools and what they cannot build
  - Running the pattern in production
  - How to build the brain (the knowledge base)
  - Maintaining session continuity
  - The forward-deployed engineer
  - Starting your own map
- Applying the pattern across the four use cases
- Summary
- References

Chapter 12: Unlock the Code Bundle and the PDF Version

- Unlock this book's free benefits in three easy steps

Index