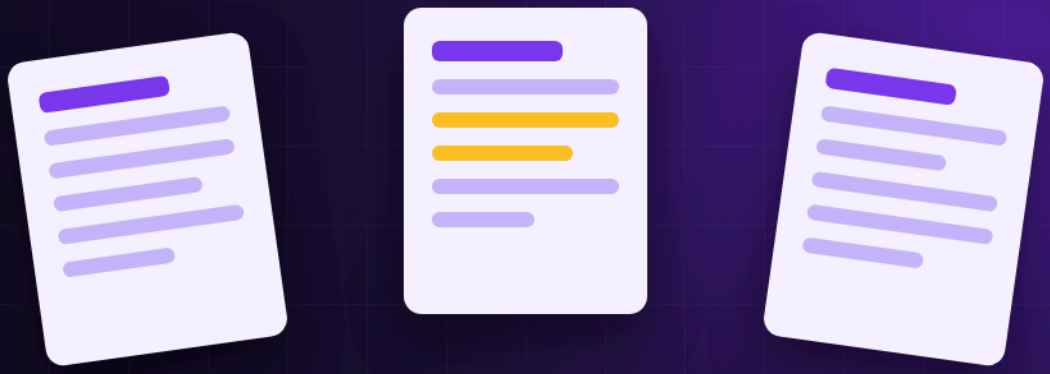


What torque for the KC-BR-2210 caliper?

Tighten to the value in the service manual [doc-12] and re-check after the first ride [doc-31].

THE PRACTICAL GUIDE

RAG in Practice

Building Retrieval-Augmented Generation Systems That Actually Work

Ingestion · Hybrid search · Reranking · Grounded answers
Evaluation · Security · Production

RETRIEVE · RERANK · GROUND · CITE · EVALUATE

THENEXBOOKS

RAG in Practice

Building Retrieval-Augmented Generation Systems That Actually Work

First Edition

THENEXBOOKS

RAG in Practice
Building Retrieval-Augmented Generation Systems That Actually Work

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code, prompts, and configuration examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Language models, embedding models, and retrieval libraries change quickly, and their behavior varies between providers and versions; test every technique with the systems you actually use. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Kestrel Cycles and all people, products, and documents associated with it are fictional. Product names such as PostgreSQL, pgvector, Qdrant, Weaviate, Milvus, Pinecone, Elasticsearch, OpenSearch, Claude, GPT, and Gemini are trademarks of their respective owners and are used for identification only.

Preface

Retrieval-augmented generation is the most common way to give a language model access to knowledge it was never trained on: your product manuals, your policies, your support history, your engineering documents. A weekend prototype is easy — embed some text, search it, paste the results into a prompt. And the first demo usually looks impressive.

Then real users arrive. They ask about a part number the embedding model cannot tell apart from a similar one. They get last year's warranty policy instead of this year's. The system cites a document that does not say what the answer claims. It answers confidently when the information simply is not there. An internal user sees a document they were never allowed to read. Costs climb, latency grows, and nobody can say whether yesterday's change made things better or worse.

This book is about closing the gap between a RAG demo and a RAG system that **actually works**. We treat RAG as an engineering discipline: start with evaluation, understand every stage of the pipeline, measure each change, and design for the failure modes you will certainly meet — ambiguous queries, messy documents, stale data, missing answers, access control, prompt injection, and cost.

Who This Book Is For

- **Software engineers** adding question answering or search over company knowledge to their products.
- **AI and ML engineers** who need a systematic approach to retrieval quality and evaluation.
- **Technical product builders and leads** deciding how to scope, measure, and ship a RAG feature.
- **Data and platform engineers** responsible for ingestion pipelines and vector infrastructure.

What You Will Learn

- How embeddings, vector search, and language models fit together — and where each one breaks.
- How to build a golden dataset and measure retrieval and answer quality from day one.
- How to ingest real documents, chunk them well, and enrich them with metadata.
- Hybrid search, query rewriting, reranking, filtering, access control, and advanced retrieval patterns.
- Grounded prompting, citations, verification, abstention, structured and multimodal answers, and conversation.
- Evaluation, agentic RAG, latency and cost, security, observability, and deployment at scale.
- Two complete case studies you can adapt to your own organization.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains the building blocks and builds a first pipeline, then defines how we will measure success. **Part II — Ingestion and Indexing** (Chapters 6–10) turns messy documents into a well-structured, fresh index. **Part III — Retrieval** (Chapters 11–15) covers everything that decides which passages the model sees. **Part IV — Generation** (Chapters 16–20) turns retrieved passages into trustworthy answers. **Part V — Production** (Chapters 21–26) addresses evaluation, agents, performance, security, observability, and scaling. **Part VI — Case Studies** (Chapters 27–28) brings everything together. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

A Note on Models and Versions

This book is **vendor-neutral**. The techniques apply whichever language model, embedding model, or vector database you choose. Code examples use placeholders such as `LLM_MODEL = "your-llm-model-id"` so they remain valid as models change. Always consult current documentation, and always evaluate on your own data.

The Running Example: Kestrel Cycles

Throughout the book we build RAG systems for **Kestrel Cycles**, a fictional bicycle and e-bike manufacturer. Its knowledge is spread across product manuals in several languages, warranty policies that differ by region and change over time, a help center and years of support tickets, a parts catalog full of part numbers, restricted engineering specifications, and dealer agreements. We build **Kestrel Assist**, a customer support assistant; a dealer question-answering portal; and **Spoke**, an internal engineering search tool with strict access control.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake that can cause wrong answers, data leaks, or outages.

Code, prompts, and SQL appear in `monospace`. Numbers labeled *hypothetical* or *illustrative* are examples, not measurements.

Contents

Preface

PART I — FOUNDATIONS

1 Why RAG: What Retrieval Solves and What It Doesn't

2 LLMs, Tokens, and Context Windows for RAG Builders

3 Embeddings and Vector Similarity Explained

4 Your First RAG Pipeline, End to End

5 Defining Success: Requirements, Metrics, and a Golden Dataset

PART II — INGESTION AND INDEXING

6 Document Ingestion: PDFs, HTML, Office Files, and Tables

7 Chunking Strategies That Preserve Meaning

8 Metadata, Enrichment, and Document Structure

9 Vector Databases and Indexes: pgvector, Dedicated Engines, and Trade-offs

10 Keeping the Index Fresh: Updates, Deletes, and Versioning

PART III — RETRIEVAL

11 Dense, Sparse, and Hybrid Search

12 Query Understanding: Rewriting, Expansion, and Decomposition

13 Reranking and Result Fusion

14 Filtering, Access Control, and Multi-Tenant Retrieval

15 Advanced Retrieval: Parent-Child, Hierarchical, and Graph-Based RAG

PART IV — GENERATION

16 Prompting for Grounded Answers

17 Citations, Attribution, and Answer Verification

18 Knowing When Not to Answer: Abstention and Fallbacks

19 Structured Outputs, Tables, and Multimodal RAG

20 Conversational RAG: Memory and Follow-Up Questions

PART V — PRODUCTION

21 Evaluating RAG: Retrieval Metrics, Answer Quality, and LLM-as-Judge

22 Agentic RAG: Tools, Routing, and Multi-Step Retrieval

23 Latency, Cost, and Caching

24 Security and Privacy: Prompt Injection, Data Leakage, and PII

25 Observability, Feedback Loops, and Continuous Improvement

26 Deploying and Scaling RAG Services

PART VI — CASE STUDIES

27 Case Study: Kestrel Assist — A Customer Support Assistant

28 Case Study: Spoke — Engineering Knowledge Search with Access Control

APPENDICES

A RAG Cheat Sheet

B Prompt and Configuration Templates

C Production Checklist

D Glossary

PART I

Foundations

The building blocks of retrieval-augmented generation, a first working pipeline, and the evaluation mindset that makes every later decision measurable.

1 · Why RAG
2 · LLMs, Tokens, and Context Windows
3 · Embeddings and Vector Similarity
4 · Your First RAG Pipeline
5 · Defining Success

CHAPTER 1 · PART I — FOUNDATIONS

Why RAG: What Retrieval Solves and What It Doesn't

Large language models are fluent but not informed about your world. This chapter explains which knowledge problems retrieval-augmented generation actually solves, which ones it does not, how it compares with the alternatives, and why a system that dazzles in a demo so often disappoints in production.

The Knowledge Problem

Ask a general-purpose large language model (LLM) how to adjust the rear derailleur on a typical road bike and you will usually get a good answer. The model has read thousands of bicycle maintenance guides, forum threads, and product reviews during training, and it can compress that experience into a clear, well-organized explanation. Now ask the same model what torque the stem bolts on a Kestrel Cycles Ridgeline 7 e-bike should be tightened to, whether the 2024 European warranty covers a cracked battery housing, or which brake caliper replaces part `KC-BR-2210`. The answers will still be fluent. They will also, very often, be wrong.

That gap between fluency and knowledge is the reason this book exists. An LLM is a model of language trained on a large, fixed snapshot of text. It is extraordinarily good at producing plausible continuations of a prompt, and in many domains plausible and correct overlap heavily. But four structural problems appear as soon as you point a model at a specific organization's questions.

1. The training cutoff

Every model's knowledge stops at some point. Training data is collected, cleaned, and frozen; the model is trained, evaluated, and released; and from that moment on, the world keeps changing while the model's parameters do not. For general knowledge this matters less than you might expect, because most facts about physics or grammar do not change month to month. For a business, it matters enormously. Kestrel Cycles revises its return policy, publishes change notices for components, and launches new models every season. None of that is inside a model that finished training before those documents existed.

A newer model only moves the cutoff forward; you cannot retrain a foundation model every time a warranty clause is edited.

2. Private data

The more important problem is that most of what an organization knows was never public in the first place. Kestrel's engineering specifications, dealer agreements, internal support ticket history, and pre-release manuals were never on the open web, so no public model could have learned them. Even the public documents — the help-center articles and the downloadable PDF manuals — may be too obscure, too recent, or too similar to competitors' documents for a model to have memorized reliably. A model that has read a hundred different brake-bleeding guides will blend them; it has no way to know which one is Kestrel's.

3. Hallucination

When a model lacks knowledge, it does not reliably say so. The term **hallucination** is widely used for output that is fluent and confident but not supported by fact or by the provided sources. The mechanism is not mysterious: a model trained to predict likely text will produce a likely-looking torque value (a number followed by `Nm`, in a plausible range) whether or not it has ever seen the right one. Chapter 2 looks at this mechanism more closely. For now the important point is practical: a support assistant that invents a torque value or a warranty term is worse than no assistant at all, because customers act on it.

4. Citations and auditability

Even when a model answers correctly from its training data, it cannot tell you *where* the answer came from; its knowledge is diffused across its parameters, not stored as retrievable documents. A support agent needs to open the manual page to confirm. A dealer needs to see which version of the agreement applies. An engineer needs to know whether a specification came from the current change notice or a superseded one. Auditability is often a hard requirement.

The Core Idea of Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) addresses these problems with a simple architectural move: instead of asking the model to answer from memory, you first *retrieve* relevant passages from your own document collection and then ask the model to answer *using those passages*. The model's job changes from "recall the answer" to "read these sources and compose an answer from them" — a task at which LLMs are much more reliable.

The term comes from Lewis et al. (2020), who combined a neural retriever with a sequence-to-sequence generator and trained them together for knowledge-intensive tasks such as open-domain question answering. In their framing, the model has two kinds of memory: **parametric memory**, the knowledge encoded in the network's weights, and **non-parametric memory**, an external index of documents that can be searched at inference time. The key insight is that non-parametric memory can be inspected, updated, and cited, while parametric memory cannot.

Most RAG systems built today do not train the retriever and generator jointly. They assemble off-the-shelf components — an embedding model, a search index, a general-purpose LLM behind an API — connected by prompts and application code. That modularity makes RAG accessible, and it is also why production systems break at the seams between independently optimized parts.

Mapped against the four problems above, RAG helps as follows:

- **Training cutoff:** the index can be updated minutes after a document changes, with no model training at all.
- **Private data:** the index can contain any document you have the right to search, and access control can be enforced at retrieval time.
- **Hallucination:** grounding the answer in retrieved text reduces (but does not eliminate) invention, and makes it detectable, because you can compare the answer to the sources.
- **Citations:** each retrieved passage has a known origin — document, page, version — so the answer can point to it.

Note the careful wording in the third bullet. RAG does not make a model truthful. A model can ignore the retrieved passages, misread a table, merge two passages that describe different products, or fill a gap with invented detail. Retrieval can also return the wrong passages, in which case the model faithfully summarizes irrelevant or outdated text. Much of this book — especially Part IV — is about closing those gaps.

RAG Versus the Alternatives

RAG is not the only way to give a model access to specialized knowledge. Teams evaluating an approach usually consider four options, often in combination.

Fine-tuning continues training a model on your own examples, adjusting its weights. It is excellent for teaching *behavior*: a consistent tone, a specific output format, a classification scheme, domain terminology. It is a poor tool for teaching *facts* that change, because every update requires another training run, the model may still blend new facts with old ones, and it still cannot cite where a fact came from.

Long-context stuffing means placing entire documents directly in the prompt and letting the model find what it needs. As model context windows have grown, this has become viable for small collections. If your whole knowledge base is one forty-page manual, putting the whole manual in the prompt may beat any retrieval pipeline you could build in a week. But it scales poorly: cost and latency grow with every token you send on every request, models may attend unevenly to very long inputs (Chapter 2 discusses this), and most organizations' document collections are orders of magnitude larger than any context window.

Tools (also called function calling) let the model request structured operations — query an order database, check inventory, call a warranty-lookup API — and receive exact results. For data that lives in structured systems, tools are usually better than RAG: the answer to "is order 88213 shipped?"

should come from the order system, not from a text passage that describes shipping in general. Chapter 22 shows how retrieval itself can become one tool among several in an agentic design.

Approach	Best for	Freshness	Citations	Main cost
RAG	Large, changing text corpora	Minutes to update	Natural	Pipeline complexity
Fine-tuning	Style, format, behavior	Retrain to update	None	Training and data prep
Long context	Small, stable document sets	Immediate	Possible	Tokens per request
Tools / APIs	Structured, live records	Real time	Via the tool	Integration work

In practice these approaches compose. Kestrel Assist, the customer support assistant we build throughout this book, uses RAG over manuals and policies, a tool to look up order status, and possibly a lightly fine-tuned or carefully prompted model to keep the brand's tone. The question is never "RAG or fine-tuning?" but "which knowledge belongs in which mechanism?"

> **TIP**
>
> A useful rule of thumb: if the answer is a *record* (an order, a stock level, an account balance), use a tool. If the answer is in *prose or semi-structured documents* that change over time, use retrieval. If you are trying to change *how* the model responds rather than *what* it knows, consider prompting first and fine-tuning second.

When RAG Is the Wrong Tool

Because RAG has become a default architecture, it is worth being explicit about cases where it adds complexity without adding value.

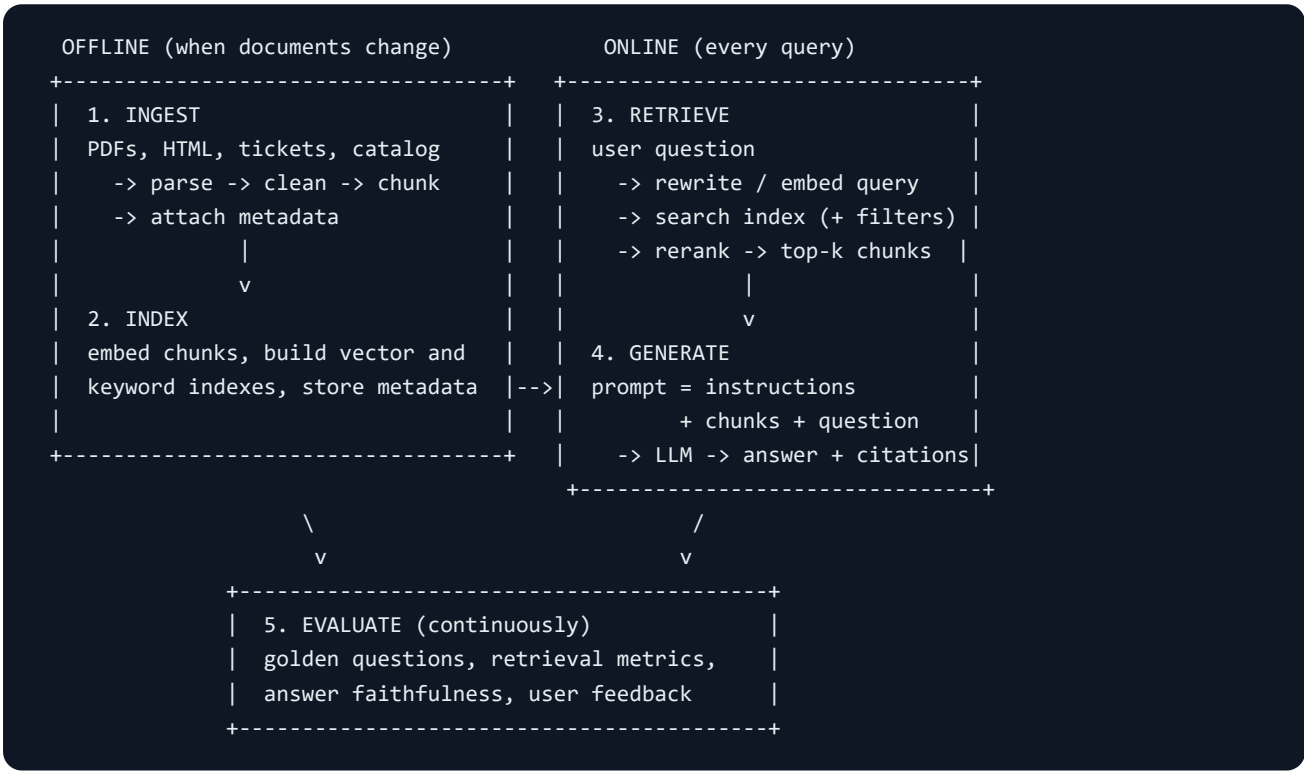
- **The corpus is tiny and stable.** A single policy document that changes twice a year can go straight into the system prompt. You will spend less time and get more predictable behavior.
- **The questions are aggregations.** "How many warranty claims did we receive for the Ridgeline 7 last quarter?" is not answerable by retrieving a handful of passages; it requires counting over all records. That is a SQL query (possibly generated by a model), not a retrieval problem.
- **The answer requires exact, live data.** Stock levels, prices, and order status belong in tools that query the system of record.
- **The documents do not contain the answer.** If Kestrel has never documented whether a third-party child seat fits the cargo rack, no retrieval strategy will find it. RAG can at best say "I don't know" (Chapter 18) — which is valuable, but it means the real fix is writing the documentation.
- **The task needs reasoning over everything.** "Summarize every change to the brake specifications since 2021" requires reading a whole class of documents. Retrieval of the top few chunks will silently omit some. Batch processing or a map-reduce summary is a better fit.

- **Mistakes are unacceptable and verification is impossible.** If a wrong answer could cause injury and there is no human in the loop, a generated answer may be inappropriate no matter how good the retrieval is. Showing the source passage directly — search, not generation — may be the safer product.

That last point deserves emphasis for a bicycle manufacturer. A torque value for a carbon handlebar clamp is safety-relevant. Over-tightening can crack the bar; under-tightening can let it rotate. Throughout this book, Kestrel's rule is that safety-critical numbers must be quoted verbatim from a cited source, never paraphrased or computed, and the assistant must decline rather than guess.

The Anatomy of a RAG System

Every RAG system, from a fifty-line prototype to a multi-region production service, has the same five stages. Two happen offline, when documents change; two happen online, on every query; and one runs continuously alongside both.



Ingest. Raw documents arrive in many formats. Kestrel's manuals are PDFs with multi-column layouts, torque tables, and captions describing exploded diagrams; the help center is HTML with navigation chrome; support tickets are semi-structured text with customer data that must be handled carefully; the parts catalog is structured data. Ingestion extracts clean text and structure, splits it into **chunks** — passages small enough to retrieve precisely but large enough to be meaningful — and attaches **metadata** such as source, language, product model, region, version, effective date, and access level. Chapters 6 through 8 cover this stage, and it is where most quality problems originate.

Index. Chunks are converted into searchable form. For semantic search this means computing an **embedding** — a vector of numbers that represents meaning (Chapter 3) — and storing it in a vector index. Most production systems also build a keyword index, because exact terms like part numbers are poorly served by embeddings alone. Chapter 9 covers storage and indexing choices; Chapter 10 covers keeping the index in sync as documents change.

Retrieve. At query time, the user's question is optionally rewritten, embedded, and matched against the index, with metadata filters applied (only English documents, only the current policy version, only documents this user may see). A reranker may reorder the candidates. The result is a small set of chunks — often somewhere between three and twenty. Part III, Chapters 11 through 15, covers retrieval in depth.

Generate. The retrieved chunks are placed into a prompt along with instructions and the question. The LLM produces an answer, ideally with citations to the chunks it used, and ideally declining when the chunks do not contain the answer. Part IV covers prompting, citations, abstention, structured outputs, and conversation.

Evaluate. Evaluation is not a final step but a loop. You need a set of representative questions with known good answers and known relevant documents, metrics that tell you whether retrieval found the right passages and whether answers are faithful to them, and production signals such as user feedback. Chapter 5 introduces this early on purpose; Chapter 21 develops it fully.

> **NOTE**
>
> The diagram shows evaluation last, but you should build it first. Without a way to measure, every change to chunking, embedding models, or prompts is a guess, and you will not notice regressions until customers do.

Why Demos Work and Production Fails

With an embedding model, a few lines of numpy, and an LLM API, a prototype that answers questions about a handful of documents can run in an afternoon; Chapter 4 does exactly that. It will answer most questions you think to ask and feel close to done. It is not, because the demo succeeds under conditions that production removes one by one.

The demo corpus is small and clean

With twenty help-center articles, almost any retrieval method finds the right one, because there are few competitors. With forty thousand chunks from manuals in three languages, six years of policy versions, and a ticket archive full of near-duplicate complaints, the right passage competes with hundreds of similar-looking wrong ones. Retrieval precision that was irrelevant at small scale becomes the dominant factor.

The demo questions are written by the builder

When you write test questions, you unconsciously borrow the vocabulary of the documents you just read. Real users write "my brakes squeal like crazy after it rained," paste part numbers with typos, and ask in German about manuals that exist only in English. That vocabulary mismatch is one of the central problems of retrieval.

The demo never meets questions without answers

In a demo, you ask questions you know the documents can answer. In production, a meaningful share of questions — the exact proportion varies widely and you should measure your own — are not answerable from the corpus at all. The system must recognize these and decline, and the naive prototype never does. It retrieves the nearest chunks, which are always *something*, and the model dutifully composes an answer from them.

The demo ignores time and permissions

Kestrel's return policy has had several versions; the demo indexes whichever files happen to be in the folder. Production must return the version that applies to the customer's region and purchase date. Likewise, the demo lets everyone see everything, while production must keep engineering change notices away from customers and dealer margins away from other dealers. These are not retrieval-quality problems; they are correctness and security problems, and they cannot be fixed by a better embedding model.

The demo is never measured

Finally, the demo is judged by impression. Production needs numbers — how often the right document is retrieved, how often answers contain unsupported claims, how often the system abstains correctly — tracked over time.

> **WARNING**
>
> The most dangerous RAG failure is not an obviously wrong answer; it is a confident, well-formatted, cited answer that quotes the wrong document version. It looks exactly like success. Only systematic evaluation against known cases will catch it.

The Running Example: Kestrel Cycles

Abstract advice about RAG is easy to agree with and hard to apply. So throughout this book we follow one fictional but realistic organization and build its systems chapter by chapter. Every technique is illustrated against Kestrel's data, and every failure mode is one Kestrel's teams would actually hit.

Kestrel Cycles is a mid-size manufacturer of bicycles and e-bikes, selling through its own web store and a network of independent dealers in North America, Europe, and Japan. Its catalog spans road bikes, gravel bikes, commuter e-bikes, and cargo e-bikes, along with components and accessories. It is

large enough to have a real documentation problem and small enough that a single platform team owns the AI initiatives.

The knowledge sources

Source	Format	Audience	Main challenge
Manuals and assembly guides	PDF, EN/DE/JA	Public	Tables, layouts, languages
Warranty and return policies	Versioned docs	Public	Region and date versions
Help-center FAQ	HTML	Public	Page chrome, overlap
Support ticket history	Text records	Support staff	PII, noise, duplicates
Engineering specs and notices	Docs, PDF	Engineering only	Access control, jargon
Dealer agreements	Contracts	Partner team only	Confidentiality
Parts catalog	Structured data	Public, dealers	Exact IDs, compatibility

Each source stresses a different part of the pipeline. The **manuals** are long PDFs where the most important content — torque specifications, tire pressure ranges, battery charging instructions — often lives in tables that naive text extraction scrambles into meaningless sequences of numbers. Diagrams are represented by captions and callout labels ("Fig. 12: Rear axle assembly, items 1–7"), and the same manual exists in English, German, and Japanese, sometimes at different revision levels.

The **warranty and return policies** are short but treacherous. Kestrel's European policy differs from its North American one, and both have changed over time: a battery warranty extended from two years to three for bikes purchased after a certain date, a return window shortened for customized builds. A question about a warranty has a correct answer only relative to a region and a purchase date, and old versions must stay available because customers who bought under them are still covered by them.

The **help-center FAQ** — short HTML articles for customers — is where the first prototype starts (Chapter 4). Its difficulties are mundane: repeated navigation chrome, overlapping articles, and articles that link to manuals instead of answering.

The **support ticket history** holds years of real customer problems and agent resolutions. It is the richest source of how customers actually phrase things, and a goldmine for building evaluation questions. It is also full of personal data and of resolutions that were wrong, superseded, or specific to one customer. Chapter 24 treats it with care.

The **engineering specifications and change notices** describe components, tolerances, test results, and changes (for example, a notice that superseded one brake rotor with another after a supplier change). They are restricted to engineering staff. The **dealer agreements** — margins, territories, service obligations — are restricted to the partner team. Both force us to treat access control as part of retrieval, not as an afterthought.

The **parts catalog** is structured data: part numbers such as `KC-BR-2210` (a brake caliper), `KC-DR-4102` (a rear derailleur), compatibility tables mapping parts to frame models and model years, and replacement relationships. It is the canonical case of data that embeddings handle badly and exact search handles well.

The three products

Kestrel Assist is a customer-facing support assistant on the Kestrel website and inside the companion app. It answers questions about assembly, maintenance, troubleshooting, warranty, and returns, drawing on the manuals, policies, FAQ, and public catalog. Its users are riders — not mechanics — and many ask in German or Japanese. It must cite sources, decline when unsure, never reveal restricted documents, and hand off to a human agent for anything involving a specific claim. Chapter 27 presents it as a complete case study.

The **dealer Q&A portal** serves Kestrel's independent dealers. Dealers ask more technical questions ("Is the 2023 cargo frame compatible with the `KC-DR-4102` derailleur hanger?"), need exact part numbers and compatibility data, and have access to dealer-only service bulletins — but each dealer may see only its own agreement. This product stresses hybrid search, structured data, and multi-tenant filtering (Chapter 14).

Spoke is internal engineering search. Kestrel's engineers use it to find specifications, test reports, and change notices, and to ask questions that span them ("Which rotors were affected by the 2024 supplier change?"). Spoke's challenges are document-level access control, dense jargon, and questions requiring several documents to answer. Chapter 28 is its case study.

The recurring hard cases

Five kinds of questions will come back again and again, because they are where naive RAG fails most often and most visibly:

1. **Part numbers and identifiers.** "What replaces `KC-BR-2210`?" Embeddings treat identifiers as fuzzy text; `KC-BR-2210` and `KC-BR-2201` can look almost identical to a semantic model while referring to different parts.
2. **Torque values and units.** "What torque for the seatpost clamp?" The answer lives in a table, must match the right model and component, must keep its unit (Nm versus in-lb), and is safety-relevant.
3. **Outdated policy versions.** "Is my battery covered for three years?" The answer depends on region and purchase date, and superseded versions look just as relevant to a retriever as current ones.
4. **Multilingual queries.** A Japanese question about a manual section that exists in English and Japanese at different revisions, or a German question where only an English source exists.

5. **Unanswerable questions.** "Can I fit a 29-inch wheel on the Urbano 3?" when no document addresses it. The correct answer is a clear, helpful "the documentation does not say," not an extrapolation.

If a technique in this book does not help with at least one of these, we will say so. If it makes one of them worse, we will say that too.

How This Book Is Organized

The book follows the life of a RAG system, from first principles to production operations.

Part I — Foundations (Chapters 1–5) builds the mental models. Chapter 2 covers the LLM concepts that matter most to RAG builders: tokens, context windows, position effects, and sampling. Chapter 3 explains embeddings and similarity from the ground up. Chapter 4 builds a complete minimal pipeline for Kestrel's help center and then dissects what goes wrong with it. Chapter 5 defines success: requirements, metrics, and a golden dataset, so that every later chapter can be measured.

Part II — Ingestion and Indexing (Chapters 6–10) handles the offline side: parsing PDFs and HTML and tables, chunking so that meaning survives, adding metadata and structure, choosing and tuning vector storage, and keeping the index fresh as documents are updated, deleted, and versioned.

Part III — Retrieval (Chapters 11–15) is the heart of quality. It covers dense, sparse, and hybrid search; query rewriting and decomposition; reranking and fusion; filtering and access control; and advanced patterns such as parent-child and graph-based retrieval.

Part IV — Generation (Chapters 16–20) turns retrieved passages into trustworthy answers: grounded prompting, citations and verification, abstention, structured and multimodal outputs, and conversation with follow-up questions.

Part V — Production (Chapters 21–26) covers evaluation in depth, agentic retrieval, latency and cost, security and privacy (including prompt injection through retrieved documents), observability and feedback loops, and deployment.

Part VI — Case Studies (Chapters 27–28) assembles everything into two complete systems, Kestrel Assist and Spoke, with the design decisions, trade-offs, and lessons from each.

You can read front to back, but the parts also stand alone; wherever a chapter depends on an earlier idea, it names the chapter. Code uses Python with stable libraries — numpy, sentence-transformers, rank_bm25, PostgreSQL with pgvector, and the Anthropic Python SDK as a representative LLM client. Model names appear as placeholders such as `"your-llm-model-id"`, because model lineups change faster than books do and the techniques do not depend on a provider.

A Mindset for the Rest of the Book

Three principles shape every recommendation that follows. **Retrieval quality bounds answer quality:** if the right passage is not among the chunks you give the model, no prompt can produce a correct, grounded answer, so the first debugging question is always "did retrieval find it?" — answerable only if you log what was retrieved. **Most problems are data problems:** a scrambled torque table or two policy versions both marked current will defeat any embedding model, so read your chunks. **Measure before you optimize:** every technique has a cost in latency, money, or complexity, and only a representative evaluation set tells you whether it pays off for your data.

With those principles in place, we turn to the component at the end of the pipeline — the language model itself — and the handful of its properties that every RAG builder needs to understand.

Key Takeaways

- LLMs have four knowledge problems for business use: a training cutoff, no access to private data, hallucination, and no ability to cite sources.
- RAG (Lewis et al., 2020) retrieves relevant passages at query time and asks the model to answer from them, turning an inspectable, updatable index into the model's working memory.
- RAG reduces but does not eliminate hallucination; retrieval errors and model misreadings still produce wrong answers.
- Fine-tuning teaches behavior, long-context stuffing suits small stable corpora, tools suit live structured records, and RAG suits large, changing text. Real systems combine them.
- RAG is the wrong tool for aggregations, live records, whole-corpus reasoning, and questions the documents do not answer.
- Every RAG system has five stages — ingest, index, retrieve, generate, evaluate — and evaluation should be built first, not last.
- Demos succeed on small, clean corpora with builder-written questions; production adds scale, real phrasing, unanswerable questions, versions, and permissions.
- Kestrel Cycles and its three products — Kestrel Assist, the dealer portal, and Spoke — anchor the book, with part numbers, torque values, policy versions, multilingual queries, and unanswerable questions as recurring hard cases.

CHAPTER 2 · PART I — FOUNDATIONS

LLMs, Tokens, and Context Windows for RAG Builders

You do not need to understand transformer internals to build good RAG systems, but you do need a working model of how an LLM reads your prompt, how it counts, and how it chooses its words. This chapter covers exactly the parts that change the design of a retrieval pipeline.

A Working Intuition: Next-Token Prediction

At its core, a large language model does one thing: given a sequence of tokens, it produces a probability distribution over what the next token should be. To generate an answer, the application samples one token from that distribution, appends it to the sequence, and asks the model again. Repeat a few hundred times and you have a paragraph. Everything else — chat formatting, system prompts, tool calls, "reasoning" — is built on this loop.

Modern chat models are further trained, after their initial pretraining, to follow instructions and behave helpfully. That post-training is why a model answers your question instead of continuing it as if it were the beginning of an essay. But the underlying mechanism is unchanged, and three consequences matter directly to RAG builders.

First, the model is always conditioning on the whole prompt. Every token it generates is influenced by every token you sent — instructions, retrieved chunks, conversation history, and the question. That is what makes RAG work at all: put the right passage in the prompt, and the probability of the correct torque value rises sharply because the value is literally present in the context. It is also why irrelevant chunks hurt: they are conditioning too.

Second, the model optimizes for plausibility, not truth. When the prompt does not contain the answer, the model still produces the most plausible continuation. For "The recommended torque for the Ridgeline 7 stem bolts is", a plausible continuation is a number between roughly 4 and 8 followed by "Nm", because that is what stem-bolt torque values usually look like in the model's training data. That is the mechanism behind hallucination described in Chapter 1, and it explains why hallucinated values are so convincing: they come from the right distribution, just not from the right document.

Third, generation is left to right and cannot be undone. Once the model has written "Yes, your battery is covered", the subsequent tokens are conditioned on that commitment, and a model will often rationalize an early mistake rather than reverse it. Prompt designs that ask the model to identify relevant sources *before* answering (Chapter 16) partly exist to put the evidence ahead of the commitment.