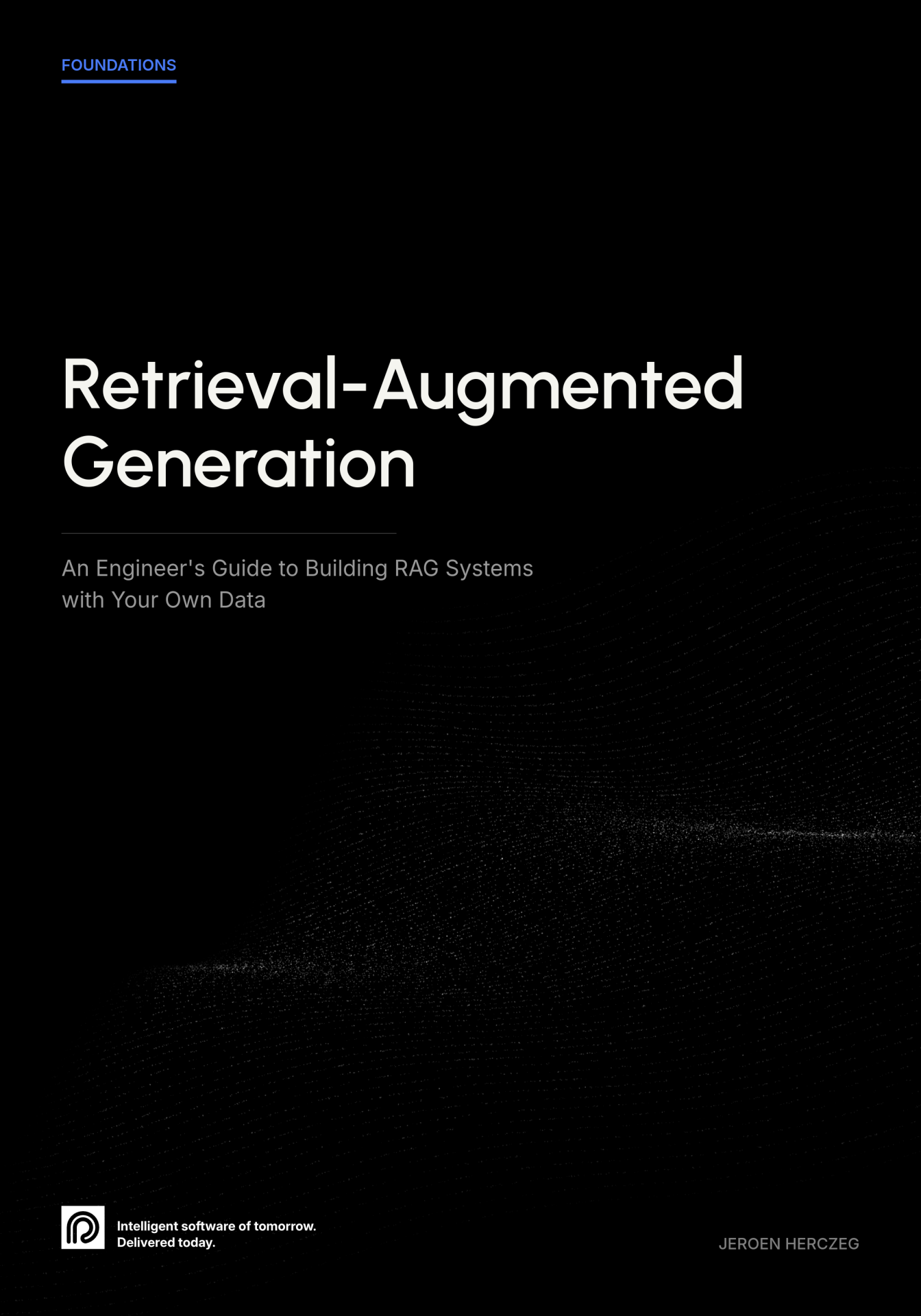
FOUNDATIONS

Retrieval-Augmented Generation

An Engineer's Guide to Building RAG Systems with Your Own Data



Intelligent software of tomorrow.
Delivered today.

JEROEN HERCZEG

Retrieval-Augmented Generation

An Engineer's Guide to Building RAG Systems with Your Own Data

Jeroen Herczeg

This book is available at
https://leanpub.com/retrieval-augmented-generation

This version was published on 2026-05-09

Contents

Chapter 1. The Problem RAG Solves

After this chapter, you will be able to:

- Explain the three core limitations of standalone LLMs in production systems.
- Describe the five stages of a RAG pipeline and the role each stage plays.
- Evaluate the tradeoffs between RAG, fine-tuning, and long-context prompting.
- Diagnose the seven major RAG failure modes and map them to the corresponding pipeline stages.
- Compare the behavior of a standalone LLM and a RAG pipeline on the same domain-specific tasks.

It's Friday afternoon at Acme Corp and you're demoing the chatbot you built. The demo is a success. It answers general questions, explains concepts, summarizes documents, even handles follow-up questions.

On Monday, the CEO walks over and asks about the company's Q3 pricing changes, announced two weeks earlier. The model answers immediately. It gives a percentage, cites an internal memo, and explains the rationale.

Every detail is wrong.

The percentage is fabricated. The memo does not exist. The rationale is a plausible fiction assembled from training data patterns. But the answer reads like it came from someone in the room.

Your CEO turns to you: "*This thing just makes stuff up?*"

Trust collapses in a single sentence.

This is not an edge case. It is the default behavior of every large language model when asked about information it was never trained on.

The model does not know that it doesn't know.

It cannot verify its claims, access your documents, or decide to "look things up." It is a language engine: excellent at producing fluent text and indifferent to whether that text is true.

If you have built anything with LLMs beyond a proof of concept, you have likely hit this wall. The model works beautifully on general knowledge, then falls apart the moment you point it at your own data.

Maybe you tried stuffing documents into the prompt, only to watch costs spike and answer quality degrade as the context window grew. Maybe you heard "*just fine-tune it*," only to discover that fine-tuning changes how a model writes, not what it reliably knows.

The gap between a compelling demo and a system people actually trust is wider than it looks.

Retrieval-Augmented Generation, or RAG, is the engineering answer to that gap. The idea is simple: before the model generates an answer, retrieve relevant documents and include them in the prompt.

The model no longer has to rely entirely on information baked into its parameters. Instead, it reads the evidence you provide and generates an answer grounded in that evidence.

When RAG works well, the system cites sources, admits when the context is insufficient, and produces answers people can verify.

But RAG is not a magic switch.

It is a pipeline, and failures at any stage propagate into the final answer. A bad chunking strategy corrupts retrieval. Weak retrieval misses the right document entirely. A poorly designed prompt causes the model to ignore the evidence it was given.

Understanding where systems fail, and why, is the difference between a RAG application that earns trust and one that makes the same confident mistakes as a standalone LLM, just with more infrastructure.

This chapter provides the map.

You will learn why LLMs fail on private and recent data, trace the RAG pipeline from ingestion to generation, compare RAG with its alternatives, and examine the seven failure points that appear in production systems.

By the end of the chapter, you will have run both a standalone LLM and a RAG pipeline against the same questions and seen the strengths and limits of each firsthand.

1.1 What an LLM can and cannot do

To understand why RAG exists, we first need to be precise about what a large language model actually does.

An LLM is a neural network trained on massive text corpora to predict the next token in a sequence. During training, it learns patterns, facts, and reasoning structures from billions of documents. Those patterns are compressed into the model's parameters: fixed numerical weights that do not change after training.

An LLM's knowledge is frozen at training time.

If a model was trained on data up to January 2025, it knows nothing about events after January 2025. It cannot look things up, access the internet, or read your company's internal wiki. Every answer it produces comes from patterns learned during training.

An LLM is a language engine, not a knowledge base.

It is exceptionally good at understanding language, following instructions, reasoning through problems, and generating fluent text. But knowledge is not

a first-class capability. When you ask a question, the model does not retrieve facts from a database. It generates the most likely continuation of the prompt, which can resemble a correct answer without actually being one.

This distinction shapes what you should and should not expect from a standalone LLM.

If you need the model to write code, summarize a document, or explain a well-known concept, it will often perform well. If you need it to answer questions about proprietary company data, cite specific internal documents, or know about something that happened last week, it will fail.

Not because the model is broken, but because you are asking a language engine to do a knowledge engine's job.

1.2 Limitations of a standalone LLM

Standalone LLMs fail in three ways that matter in production. Understanding those failures is the starting point for understanding RAG.

Stale knowledge. Every model has a training cutoff date. Ask about events, documents, or policy changes after that date and the model either refuses to answer or, more dangerously, answers using outdated information without telling you.

In enterprise systems, this is not a minor inconvenience. A customer support bot that references last year's return policy, a compliance tool that cites superseded regulations, or an internal assistant that describes an outdated org structure can cause real damage.

The model does not flag its own staleness. It answers with the same confidence whether the underlying information is current or obsolete.

Hallucination. When an LLM does not know the answer, it does not reliably say "I don't know." Instead, it generates a plausible response.

Researchers sometimes call this confabulation, but the mechanism is straightforward: the model is not reasoning about truth. It is generating text that matches learned patterns, regardless of factual accuracy.

Hallucinations become especially dangerous when they contain specific details. A vague mistake is easy to dismiss. A fabricated answer that includes

dates, percentages, citations, or document names feels authoritative, and that false authority is what destroys trust.

No provenance. Even when an LLM produces a correct answer, it typically cannot tell you where the answer came from. It cannot point to a source document, quote a passage, or provide a page number.

In many domains, that is a dealbreaker. A legal research system that cannot cite the relevant statute is difficult to trust. A medical assistant that cannot trace a recommendation back to clinical guidance becomes a liability.

In high-stakes environments, an unverifiable answer is often equivalent to no answer at all.

These limitations are not temporary bugs that disappear with larger models. They are structural consequences of how LLMs work.

A larger model still has a training cutoff. A more capable model may hallucinate less often, but it will still hallucinate. And provenance does not emerge from scale alone, because provenance requires access to source documents at inference time, something standalone LLMs do not have.

1.3 The RAG mental model

RAG addresses these limitations by giving the model access to external documents at generation time.

At a high level, the process is simple.

When a user asks a question, the system searches a knowledge base for relevant documents. Those documents are inserted into the prompt alongside the user's question, and the model generates an answer grounded in the retrieved context rather than relying entirely on information stored in its parameters.

That is the core idea.

Retrieval grounds the model in real data. Because the knowledge base can be updated independently of the model, the system is not limited by the model's training cutoff. Because the model can reference retrieved passages, hallucinations become less likely. And because the answer is tied to specific documents, the system can provide provenance.

Conceptually, RAG is straightforward. The complexity is in the engineering.

How should documents be split into searchable chunks? How should those chunks be represented so semantic meaning is preserved? How do you search millions of documents with low latency? How much context should be placed into the prompt before performance degrades? How do you evaluate whether the system is actually retrieving the right information?

Each of those questions becomes a major engineering problem in production systems.

But the mental model remains simple:

Retrieve relevant documents. Insert them into the prompt. Generate an answer grounded in evidence.

1.4 The RAG pipeline end-to-end

Now let's make the mental model concrete.

A RAG system consists of five stages organized into two separate flows:

- an offline indexing pipeline that prepares documents for retrieval
- an online query pipeline that handles user questions in real time

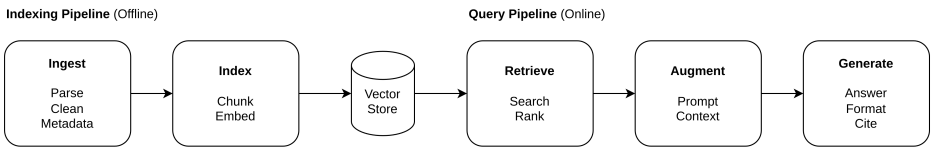


Figure 1. The two flows of a RAG system: an offline indexing pipeline that prepares documents, and an online query pipeline that handles each user question.

To see how the pieces fit together, imagine a user asks:

"*How do I get my money back?*"

Stage 1: Ingest.

Before the system can answer questions, it must process the source documents.

This stage parses PDFs, HTML pages, markdown files, and other raw inputs. The system extracts text, removes formatting artifacts such as OCR errors or duplicate headers, and captures metadata including titles, timestamps, and document categories.

Ingestion runs offline, either once per document or whenever the document changes.

Stage 2: Index.

Raw documents are too large to search efficiently, so the text is split into smaller units called chunks.

Each chunk is converted into an embedding: a numerical vector that captures semantic meaning. Those embeddings are stored in a vector database optimized for similarity search.

The result is a searchable index that retrieves documents by meaning rather than exact keyword matches.

Like ingestion, indexing runs offline.

Stage 3: Retrieve.

When a user submits a question, the system converts the query into an embedding using the same embedding model used during indexing.

The vector database then searches for chunks with similar embeddings and returns the closest matches. Those retrieved chunks become the system's best estimate of which documents are relevant to the question.

This is where the refund policy document should surface.

Stage 4: Augment.

The retrieved chunks are inserted into the prompt alongside the user's question.

The prompt typically instructs the model to answer using the provided context, cite sources when possible, and say "*I don't know*" when the retrieved evidence is insufficient.

This stage sounds simple, but prompt construction has an outsized effect on system quality. Small design choices can significantly change how the model uses retrieved context.

Stage 5: Generate.

Finally, the LLM generates a response using the augmented prompt.

Because the model now has access to the actual refund policy document, it can reference specific policies, quote relevant sections, and generate an answer grounded in evidence rather than parametric memory.

A common mistake is treating the indexing pipeline and query pipeline as a single system.

They are separate workflows with different operational constraints. The indexing pipeline runs offline in batch over the entire document corpus. The query pipeline runs online, per request, under real-time latency constraints.

The vector store connects the two, but the pipelines operate independently.

1.5 RAG vs. fine-tuning vs. long-context prompting

Before choosing RAG, it is important to understand the alternatives and the tradeoffs each one makes.

Three approaches dominate this design space:

- fine-tuning
- long-context prompting
- retrieval-augmented generation

Fine-tuning modifies a model's weights using your data. This is effective for teaching a model a particular writing style, response format, or domain-specific vocabulary. But fine-tuning is unreliable for injecting precise factual knowledge, especially through continued pretraining on raw documents. [1]

A fine-tuned model may learn to *sound* like it understands your domain without reliably recalling the underlying facts.

[1] O. Ovadia et al., "Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs," EMNLP 2024. https://arxiv.org/abs/2312.05934

Use fine-tuning when you want to change how the model behaves, not when you need to continuously update what it knows.

Long-context prompting takes advantage of modern context windows, ranging from hundreds of thousands to millions of tokens, by placing large document collections directly into the prompt.

For small corpora, this can work surprisingly well. But the approach becomes increasingly expensive and unreliable at scale.

The first problem is cost. Every query pays for every token in the context window, even though most of that information is irrelevant to the user's question.

The second problem is attention degradation. Research on the *Lost in the Middle* effect shows that models attend less reliably to information placed in the middle of long contexts, meaning critical facts can effectively become invisible depending on where they appear. [^liu]

The third problem is provenance. Models can reference document identifiers included in the prompt, but they can still misattribute claims, omit relevant sources, or blend information across documents without a reliable audit trail.

The rise of million-token context windows has led some practitioners to argue that RAG is becoming obsolete. This conclusion is premature.

Long-context prompting and RAG solve different problems.

Long-context prompting works well when the entire corpus fits comfortably inside the context window and changes infrequently. RAG becomes necessary when the corpus exceeds the context window, updates continuously, or requires verifiable provenance.

Most production systems that operate on large, evolving enterprise datasets still require retrieval.

RAG occupies the practical middle ground for most production applications. It scales to large corpora, supports continuously changing data, and enables traceable answers tied to source documents.

The tradeoff is engineering complexity.

The rest of this book is about managing that complexity well.

1.6 The seven failure points

RAG systems fail like pipelines: a problem at one stage propagates into everything that follows.

In 2024, Barnett et al. proposed a taxonomy of seven common RAG failure modes that has since become a widely used diagnostic framework. [2] Understanding these failures early will make the rest of this book easier to reason about, because nearly every retrieval, ranking, prompting, and evaluation technique exists to address one or more of them.

Failure Point 1: Missing content.

The required information simply does not exist in the knowledge base.

No retrieval system, regardless of sophistication, can retrieve a document that was never ingested. The source may have been excluded from ingestion, added after the most recent indexing run, or never existed at all.

In a well-designed system, the model should respond with "*I don't know.*" In a poorly designed system, it will generate a plausible answer with no indication that the retrieved evidence was incomplete or absent.

This is one of the most dangerous failure modes because the response can appear fully credible.

Failure Point 2: Missed top ranked.

The relevant document exists in the index, but the retriever ranks it too low to appear in the final results.

The answer may exist at rank 47 while the system only evaluates the top 10 retrieved chunks. This commonly occurs when the user's phrasing differs from the document language or when the embedding model fails to capture the semantic relationship between them.

This failure originates in retrieval.

Hybrid search and reranking are the standard mitigations and will be covered in Chapters 8 and 9.

Failure Point 3: Not in context.

The relevant document was retrieved but never included in the final prompt.

[2] S. Barnett et al., "Seven Failure Points When Engineering a Retrieval Augmented Generation System," 2024. https://arxiv.org/abs/2401.05856

This can happen when relevance thresholds exclude borderline matches, when prompt budgets force chunk selection, or when prompt ordering pushes critical information into positions the model under-attends to.

The retriever succeeded. The context assembly stage failed.

Failure Point 4: Not extracted.

The answer exists in the retrieved context, but the model fails to extract it correctly.

Barnett identifies two common causes:

- excessive noise in the retrieved context
- contradictory evidence across retrieved documents

In both cases, the information is technically present, but the model fails to synthesize it into the final answer.

This failure occurs during generation and is heavily influenced by prompt design.

Failure Point 5: Wrong format.

The model retrieves and extracts the correct information but presents it in a form that does not satisfy the user's request.

A yes-or-no question produces a multi-paragraph explanation. A request for structured output returns narrative prose. A request for precise figures produces a qualitative summary instead.

This is primarily a generation-stage problem and is usually addressed through prompt constraints and output formatting instructions.

Failure Point 6: Incorrect specificity.

The response is technically correct but delivered at the wrong level of detail.

Some users need concise factual answers. Others need explanation, context, or step-by-step guidance. A tutoring system, for example, should usually expand answers rather than returning only the minimal fact.

The inverse problem also occurs: vague user questions often produce vague answers.

This failure typically reflects a mismatch between user intent, prompt design, and query formulation.

Failure Point 7: Incomplete.

The retrieved context contains all the necessary information, but the model fails to synthesize the full answer.

This often appears in multi-part questions such as:

"*What are the key points covered in documents A, B, and C?*"

The model focuses on one portion of the context while underrepresenting the rest.

The solution is usually not better retrieval, but better query decomposition: breaking a complex question into smaller sub-questions and answering them separately before synthesis.

Here is how the seven failure points map onto the RAG pipeline:

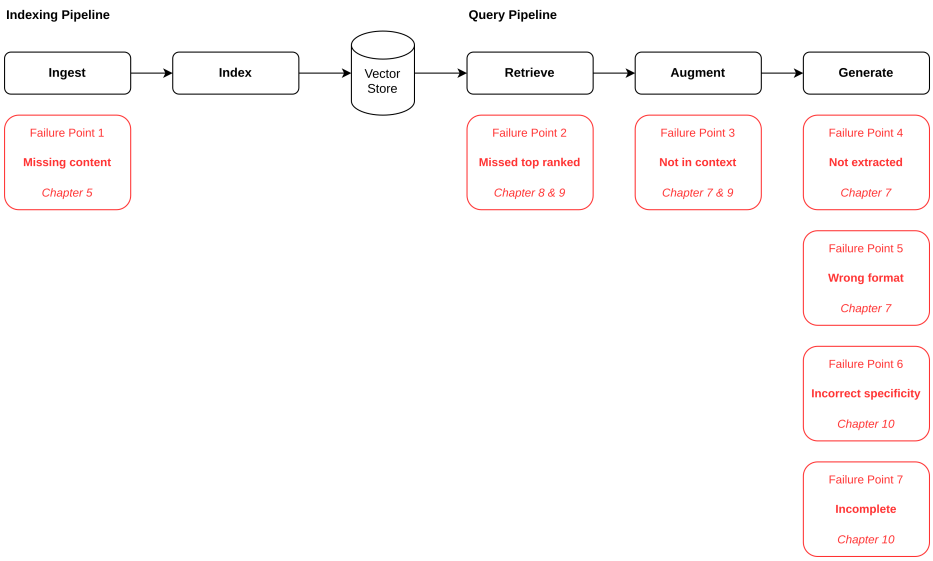


Figure 2. The seven failure points mapped to their corresponding pipeline stages.

This taxonomy serves as a diagnostic framework for the rest of the book.

Whenever we introduce an improvement technique, we will identify which failure modes it addresses. And when debugging a production system, the first step is almost always the same: classify the failure, then identify the pipeline stage responsible for it.

1.7 Common misconceptions

Before moving to the hands-on examples, it is worth clearing up several common misconceptions about RAG systems.

"RAG eliminates hallucinations."

It does not.

RAG reduces hallucinations by grounding the model in retrieved evidence, but retrieval alone does not guarantee correctness. Models can still ignore the provided context, fabricate realistic-looking citations, or generate confident answers from irrelevant retrieval results.

A production system must explicitly define fallback behavior, including when the model should respond with "*I don't know*." Just as importantly, you must verify that the model consistently follows those instructions.

RAG is a mitigation strategy, not a cure.

"Embeddings are all you need."

Semantic search is powerful because embeddings capture conceptual similarity rather than exact keyword overlap. But embeddings alone perform poorly on many classes of queries, including identifiers, abbreviations, code snippets, and exact regulatory references.

For example, a purely semantic system asked about "*HIPAA Section 164.312(a)(1)*" may retrieve general healthcare privacy documents instead of the specific regulation being referenced.

Production retrieval systems rarely rely on embeddings alone. Most combine semantic retrieval with keyword-based methods such as BM25, a hybrid approach we will build in Chapter 8.

"Better models fix bad retrieval."

They do not.

If retrieval returns irrelevant or incomplete context, even the strongest generation model will fail. The model can only reason over the information placed into its prompt.

Upgrading the LLM will not fix missing documents, weak ranking, poor chunk selection, or fragmented evidence. Retrieval quality sets the upper bound on generation quality.

"Set it and forget it."

RAG systems are not static.

Documents change over time. Policies are updated, records are deleted, and new information continuously enters the corpus. Meanwhile, the embeddings stored in the vector database still reflect older versions of those documents.

This creates what practitioners often call embedding rot: a gradual degradation in retrieval quality as the index drifts away from the current state of the underlying data.

The dangerous part is that the system often continues to answer with full confidence, making the degradation difficult to notice until users begin reporting incorrect results.

Production RAG systems require ongoing maintenance, including re-indexing strategies, embedding refresh policies, versioning, evaluation pipelines, and monitoring.

We will revisit these operational concerns in Chapter 12.

1.8 Seeing it for yourself: Standalone LLM vs. RAG

Theory matters, but nothing clarifies a system faster than watching it succeed and fail on real questions.

In this section, you will run two scripts:

- one that sends domain-specific questions to a standalone LLM
- one that sends the same questions through a prebuilt RAG pipeline

For now, the RAG pipeline is treated as a black box. By Chapter 7, you will build the entire system yourself.

The running example: Acme Corp

Systems that perform well on curated demos often fail on messy real-world corpora. To avoid that trap, every chapter in this book uses the same fictional company knowledge base: Acme Corp, a 500-person SaaS company

with roughly 110 internal documents spanning HR, security, operations, compliance, product, and engineering.

The corpus is intentionally heterogeneous. It includes PDFs, scanned PDFs with OCR artifacts, markdown files, HTML pages, and DOCX documents. If your ingestion pipeline cannot handle those formats reliably, it will not survive contact with a real enterprise environment.

The corpus is also intentionally imperfect.

The retention policy says customer data is stored for seven years. The deletion workflow says customer records are removed after ninety days. That contradiction is deliberate. The goal is not to create a clean benchmark dataset, but a realistic system with ambiguity, inconsistency, and missing information.

Several questions used throughout the book also have no answer in the corpus. This allows you to observe one of the most important behaviors in retrieval systems: whether the model correctly says "*I don't know.*"

All code examples for the book are available in the accompanying GitHub repository. [3]

Clone the repository, install the dependencies with `pip install -r requirements.txt`, and copy `.env.example` to `.env` with your API credentials. Each chapter contains its own runnable examples and setup instructions.

The corpus itself lives in `data/corpus/`. Most chapters use that shared dataset, although Chapters 2 and 4 introduce additional data sources where appropriate.

We begin with the standalone LLM.

The first script sends five questions about Acme Corp's internal systems and policies directly to the model with no additional context. The relevant documents exist in the corpus, but they were never part of the model's training data.

[3] https://github.com/retrieval-augmented-generation/book

```
$ python ch01-the-problem/ch01.1-standalone_llm_test.py

Q1: How do I get my money back?
A1: To get your money back, you will need to reach out to the company
    or individual you made the payment to and request a refund based
    on their refund policy and any applicable consumer rights or
    regulations.

Q2: What is the SLA uptime guarantee?
A2: The SLA uptime guarantee is 99.9%.

Q3: How does the deployment pipeline work?
A3: The deployment pipeline automates the steps needed to take code
    from development to production, including building, testing, and
    deploying software changes.

Q4: How do I authenticate API requests?
A4: You can authenticate API requests using methods like OAuth 2.0,
    API keys, or tokens.

Q5: What does CVE-2024-1234 affect in Acme Corp's systems?
A5: CVE-2024-1234 affects Acme Corp's CRM system, specifically the
    contact information extraction functionality.
```

Two failure patterns appear immediately.

Questions about general concepts (Q1, Q3, and Q4) produce plausible but non-specific answers. The model does not know Acme Corp's actual policies, so it falls back to generic responses that could apply to almost any company.

Questions requiring precise factual knowledge produce hallucinations. Q2 invents an SLA value, and Q5 fabricates a nonexistent CRM subsystem.

More importantly, the model never acknowledges uncertainty. The vague answers sound reasonable but provide little operational value. The specific answers are more dangerous because they sound authoritative.

Now compare the same questions against the prebuilt RAG pipeline.

This version retrieves relevant documents from the indexed corpus and inserts them into the model's prompt before generation.

```
$ python ch01-the-problem/ch01.2-rag_pipeline_test.py

Loaded 112 documents from data/embeddings

Q1: How do I get my money back?
A1: To get your money back, return the item within 30 days of
    delivery, and once it is received and inspected at the warehouse,
    a credit will be issued to your original payment method within
    2 business days, typically appearing on your statement within
    5-7 business days after that.
Source: return_policy.html

Q2: What is the SLA uptime guarantee?
A2: The SLA uptime guarantee is 99.95%.
Source: sla_dashboard_guide.md

Q3: How does the deployment pipeline work?
A3: The deployment pipeline at Acme Corp follows a blue-green
    deployment model with a canary rollout, consisting of three
    phases: canary, progressive rollout, and full cutover. Each
    phase includes automated metric checks, and the final step
    requires manual confirmation by the deployer.
Source: deployment_runbook.md

Q4: How do I authenticate API requests?
A4: To authenticate API requests to the Acme Corp API, you can use
    either API key authentication by including the API key in the
    Authorization header, or OAuth 2.0 authentication using the
    appropriate flow for your application.
Source: api_auth_docs.html

Q5: What does CVE-2024-1234 affect in Acme Corp's systems?
A5: I don't know.
Source: security_advisory_q3_2024.md
```

The difference is immediate.

Q1 through Q4 are now grounded in actual documents. The SLA value is correct. The deployment process references Acme Corp's real rollout strategy. Every answer includes a traceable source.

Q5 still fails, but in a more instructive way.

The retriever surfaced the Q3 2024 security advisory rather than the Q1 bulletin that actually references CVE-2024-1234. Semantic embeddings capture conceptual similarity, not exact identifier matching, so the retriever selected a topically related document instead of the precise one.

Because the retrieved context did not contain the answer, the model responded with "*I don't know*" rather than fabricating a system name.

This is still a retrieval failure, but it is a safer one.

The fix is hybrid retrieval: combining semantic search with exact keyword matching. We will build that system in Chapter 8.

At production scale, every stage in the pipeline becomes a potential source of failure. Retrieval quality, chunking strategy, prompt construction, ranking, and context assembly all shape the final answer.

The rest of this book is about improving those stages systematically.

1.9 Summary

- LLMs are language models, not knowledge systems. They generate plausible text from fixed training data and cannot independently access private, recent, or external information.
- The three core limitations of standalone LLMs in production systems are stale knowledge, hallucination, and missing provenance.
- RAG addresses these limitations by retrieving relevant documents at query time and grounding generation in external evidence.
- A RAG system consists of two pipelines: an offline indexing pipeline (ingest and index) and an online query pipeline (retrieve, augment, and generate).
- Fine-tuning changes model behavior and style, but it is unreliable for continuously updating factual knowledge. Long-context prompting works for small or static corpora but becomes increasingly expensive and unreliable at scale.
- RAG does not eliminate hallucinations. It reduces them when retrieval, ranking, prompting, and context assembly all work correctly.
- The seven failure points provide a practical framework for diagnosing RAG systems: missing content, missed ranking, missing context, failed extraction, incorrect formatting, incorrect specificity, and incomplete synthesis.

* * *

You have now seen both the strengths and the limits of retrieval-augmented generation. Underneath the entire system is a deceptively simple idea: text can be represented as numbers that preserve semantic meaning, and those numerical representations can be compared mathematically.

In the next chapter, we will move from raw text to vector embeddings, explore how semantic similarity is represented in vector space, and build the intuition needed to understand modern retrieval systems.

Chapter 2. Embeddings: From Text to Vectors

After this chapter, you will be able to:

- Explain what embeddings are, how they differ from raw text, and why semantic similarity can be computed mathematically.
- Generate embeddings using both local models and API-based embedding services.
- Compute cosine similarity between embeddings and interpret the resulting scores.
- Visualize high-dimensional embedding spaces in two dimensions using UMAP.
- Evaluate and select embedding models based on retrieval quality, operational constraints, and production requirements.

Now that we have seen RAG work end-to-end, the next question is how retrieval actually finds the right documents.

From this point forward, the book begins unpacking the pipeline piece by piece. This chapter starts with the foundation: representing text as numbers in a way that preserves semantic meaning.

In the previous chapter, a RAG pipeline correctly answered the question "*How do I get my money back*?" by retrieving the company's return policy.

The query shared almost no vocabulary with the document itself. There was no "*refund*," "*return*," or "*credit*" in the question. A purely keyword-based search system would have struggled to connect the two.

Yet retrieval still found the correct document.

How?

Before retrieval occurs, both the user's query and the stored documents are converted into numerical representations called vectors. Those vectors occupy positions in a high-dimensional space where semantically similar texts appear near one another.

As a result, "*How do I get my money back*?" and "*Return Policy*" end up geometrically close even though they share few or no overlapping terms.

Retrieval becomes a geometry problem:

find the stored vectors closest to the query vector.

This transformation from text into vectors is one of the foundational ideas behind modern retrieval systems. It makes semantic search possible.

Without embeddings, retrieval systems are forced to rely primarily on keyword overlap, and keywords are an unreliable proxy for meaning. Users ask for "reimbursement" while documents say "refund." They ask about "latency" while documentation describes "response time." Exact matching fails precisely where semantic understanding matters most.

The numerical representation of a piece of text is called an **embedding**.

Embeddings are lossy representations. They preserve semantic structure while discarding the exact wording of the original text. Once text has been embedded, the original passage cannot be reconstructed from the vector alone.

That tradeoff is intentional.

Two passages describing the same underlying concept can end up close together in vector space even when their vocabulary differs entirely. That property is what allows semantic retrieval systems to generalize beyond exact string matching.

This chapter explains how text becomes vectors and why mathematical operations on those vectors approximate human judgments about semantic similarity.

You will generate embeddings using both local and API-based models, compute similarity scores directly, visualize embedding spaces with UMAP, and build a brute-force semantic search system from scratch.

By the end of the chapter, you will have a working retrieval system driven entirely by vector similarity, along with a clear understanding of both its strengths and its limitations.

2.1 From words to vectors

The idea that meaning can be represented numerically is older than modern LLMs.

It begins with a foundational idea from linguistics known as the distributional hypothesis: words that appear in similar contexts tend to have similar meanings. If we observe that "*dog*" and "*puppy*" frequently occur near words such as "*leash*," "*bark*," "*pet*," and "*walk*," we can infer a semantic relationship between them even without explicit definitions.

Early embedding methods such as Word2Vec [1] operationalized this idea directly. These models trained neural networks to predict words from their surrounding context, or predict surrounding words from a target word. During training, the network learned numerical representations for each term, and those learned representations became the embeddings.

Each word was assigned a single fixed vector in space.

That design introduced an important limitation: the embedding for "*bank*" was identical whether the sentence referred to a financial institution or the side of a river. These representations are therefore called **static embeddings** because the vector does not change across contexts.

[1] T. Mikolov et al., "Efficient Estimation of Word Representations in Vector Space," 2013. https://arxiv.org/abs/1301.3781

Modern embedding models take a different approach.

Transformer-based architectures generate **contextual embeddings**, where the representation of a word depends on the surrounding text. The vector for "*bank*" changes depending on whether the passage discusses finance or geography.

More importantly, modern embedding models typically operate at the passage level rather than the individual-word level. A sentence, paragraph, or longer document is converted into a single vector intended to capture the semantic meaning of the entire passage.

This distinction is critical for retrieval systems.

Retrieval is not fundamentally about matching words. It is about matching meaning. The goal is not to compare a query term-by-term against a document, but to compare the semantic content of the query against the semantic content of stored passages.

An embedding places a piece of text at a point in a high-dimensional vector space.

Semantically related texts occupy nearby regions of that space. Unrelated texts appear farther apart.

For example:

- "*How to reset my device*" and "*Restoring factory settings*" tend to cluster together.
- "*How to reset my device*" and "*The history of the Roman Empire*" do not.

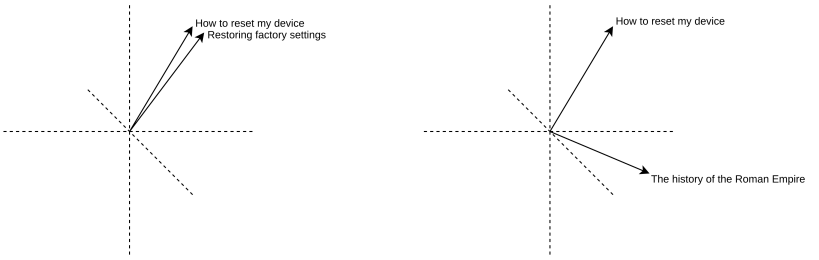


Figure 3. Semantically similar texts land near each other in vector space, while unrelated texts land far apart.

These vector spaces are not two-dimensional or three-dimensional. Modern embedding models commonly produce vectors with hundreds or thousands of dimensions, such as 384, 768, or 1,536.

Humans cannot directly visualize spaces of that dimensionality, but the underlying geometry still behaves consistently. Distance, clustering, and directional relationships remain mathematically meaningful, and those geometric relationships correspond surprisingly well to semantic similarity.

2.2 The bi-encoder architecture

Before generating embeddings, it is useful to understand the model architecture that produces them.

Most embedding models used for retrieval are **bi-encoders**. In a bi-encoder architecture, queries and documents are encoded independently using the same model, producing a separate vector representation for each input. Similarity is then computed by comparing those vectors geometrically.

This independence is what makes large-scale retrieval practical.

During indexing, the system embeds every document once and stores the resulting vectors. At query time, only the user query needs to be embedded. Retrieval then becomes a vector search problem over the precomputed document embeddings.

Without this separation, retrieval would require recomputing representations for every query-document pair at search time, which would be computationally infeasible at scale.

An alternative architecture, called a **cross-encoder**, processes the query and document together as a single input sequence. Because the model jointly attends to both texts, cross-encoders typically produce more accurate relevance judgments than bi-encoders.

The tradeoff is performance.

Cross-encoders cannot precompute document embeddings. The model must run once for every query-document pair, making exhaustive retrieval prohibitively expensive over large corpora.

For that reason, cross-encoders are rarely used for primary retrieval. Instead, they are commonly used for *reranking*: rescoring a small set of candidate

documents returned by a faster retriever. We will revisit this architecture in Chapter 9.

For now, the important idea is simple:

Bi-encoders make retrieval efficient because queries and documents can be embedded separately. That separation enables the fundamental retrieval pattern used throughout modern RAG systems:

embed once, search many times.

2.3 Generating embeddings locally

It is time to generate embeddings.

At this stage, the model produces nothing more than vectors of floating-point numbers. The challenge is determining whether those vectors actually capture semantic meaning in a useful way. This section focuses on generating embeddings. The following sections will introduce techniques for inspecting and evaluating them.

We will use the `sentence-transformers`[2] library, which provides a lightweight Python interface for running embedding models locally. In this context, locally means the model executes entirely on your own hardware: no API calls, no external services, and no data leaving the machine.

Throughout this chapter, we will work with a small corpus defined in `ch02-embeddings/corpus.py`: thirty short passages drawn from the fictional Acme Corp knowledge base. The corpus contains five examples each from Operations, Engineering, IT & Security, Product, Compliance, and HR & Benefits.

The examples are intentionally chosen to create meaningful semantic clusters while also exposing the limits of embedding-based similarity. All subsequent scripts in this chapter import from the same corpus file.

[2] https://sbert.net

```
$ python ch02-embeddings/ch02.1-generate_embeddings_locally.py

Type: <class 'numpy.ndarray'>
Shape: (30, 384)
Dtype: float32
First embedding (first 10 values): [-0.10466396 -0.05740074 -0.08202061
↪  -0.0030316  -0.00774839 -0.06115615
 -0.0388694   0.03082308  0.04396805  0.02269078]
```

Let us unpack the output.

The result is a NumPy array with shape (30, 384). We passed 30 documents into the `all-MiniLM-L6-v2` [3] embedding model and received 30 vectors in return, each with 384 dimensions.

The dimensionality comes from the model architecture, not from the input text itself. Every passage embedded by `all-MiniLM-L6-v2` produces a 384-dimensional vector regardless of whether the input contains two words or several paragraphs.

`all-MiniLM-L6-v2` is an excellent model for learning retrieval pipelines, but its maximum context window is only 256 tokens.

Every embedding model has a fixed input limit, and most truncate inputs silently once that limit is exceeded. Tokens beyond the maximum length are discarded without warning.

As a result, the embedding may represent only the beginning of a document rather than the document as a whole.

Always verify a model's context window before embedding production data. Section 2.7 discusses embedding model selection in more detail.

The vector values themselves are stored as `float32` numbers. An embedding is simply a sequence of floating-point values arranged in high-dimensional space.

Importantly, individual dimensions are not human-interpretable. A single coordinate does not correspond to concepts such as "*about databases*" or "*related to security*." Semantic meaning is distributed across the entire vector.

One final detail is critical:

[3] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Embeddings generated by different models are not interchangeable.

Each embedding model defines its own vector space. Dimension 42 in one model has no meaningful relationship to dimension 42 in another, even if both models produce vectors of identical length.

Mixing embeddings from different models inside the same index creates a subtle but severe failure mode. The retrieval system will continue returning results, but the similarity calculations will no longer be meaningful.

For that reason, production systems must use the same embedding model for both document indexing and query embedding.

2.4 Generating embeddings via API

Not all embedding models run locally.

Many high-performing embedding systems are exposed only through hosted APIs. One of the most widely used examples is OpenAI's `text-embedding-3-small`. Instead of executing the model on local hardware, we send text to the API provider and receive embeddings in return.

The following script embeds the same 30-document corpus using `text-embedding-3-small`.

```
$ python ch02-embeddings/ch02.2-generate_embeddings_api.py

Type: <class 'numpy.ndarray'>
Shape: (30, 1536)
Dtype: float64
First embedding (first 10 values): [ 0.00283813  0.03424072  0.08551025
↪  0.02151489  0.03915405  0.03039551
 -0.01983643  0.02828979  0.05639648 -0.03414917]
```

The corpus is unchanged, but the embeddings are different.

Each vector now contains 1,536 dimensions rather than 384 because the dimensionality is determined by the embedding model itself. Different models define different vector spaces with different embedding sizes, training objectives, and retrieval characteristics.

Higher dimensionality does not automatically imply higher retrieval quality. Larger vectors increase storage requirements, memory usage, and similarity

computation costs, and in many applications smaller models perform competitively despite lower dimensionality.

Choosing between local and API-based embedding models involves practical engineering tradeoffs.

Local models (sentence-transformers):

- No per-token API costs
- Data remains entirely within your infrastructure
- Full control over model versioning and reproducibility
- Requires local CPU or GPU resources
- Smaller local models may underperform state-of-the-art hosted models

API models (OpenAI, Voyage AI, Cohere):

- Often provide stronger general-purpose retrieval quality
- No infrastructure management required
- Usage-based pricing introduces ongoing operational cost
- Data is transmitted to an external provider
- Model updates or deprecations can change embedding behavior over time

Neither approach is universally correct.

Some organizations cannot send sensitive data to external providers because of privacy, compliance, or regulatory constraints. Others prefer managed APIs to avoid operating GPU infrastructure internally.

In practice, embedding model selection is rarely just a machine learning decision. It is also an operational, financial, and organizational one.

2.5 Cosine similarity and distance metrics

We now have embeddings. The next step is determining how to measure similarity between them.

Modern retrieval systems commonly rely on three related metrics:

- cosine similarity
- dot product

- Euclidean distance (L2 distance)

Cosine similarity measures the angle between two vectors rather than their absolute distance. If two vectors point in the same direction, their cosine similarity is `1`. If they are orthogonal, the similarity is `0`. If they point in opposite directions, the similarity is `-1`.

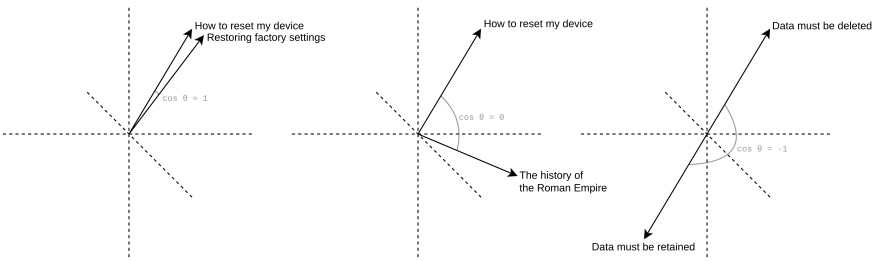


Figure 4. Three cases of cosine similarity: same direction, perpendicular, and opposite.

The formula is straightforward:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Figure 5. The cosine similarity formula: dot product of A and B divided by the product of their norms.

The numerator is the *dot product* of vectors A and B: multiply corresponding elements and sum the results. The denominator divides by the magnitudes, or *norms*, of the two vectors.

This normalization step is important because it removes the effect of vector length. Cosine similarity measures directional alignment rather than absolute magnitude.

Let us compute the metric directly in Python.

```
$ python ch02-embeddings/ch02.3-cosine_similarity_manual.py

Dot product:        0.2475
Norm of A:          1.0000
Norm of B:          1.0000
Cosine similarity:  0.2475
```

The script compares two Operations-related passages: one describing the SLA uptime guarantee and another describing the enterprise refund policy. It computes the dot product, vector norms, and cosine similarity explicitly rather than relying on a library helper function.

Notice that both vector norms are exactly `1.0`.

This occurs because sentence-transformers normalizes embeddings by default. Once vectors are normalized, the denominator of the cosine similarity equation becomes `1 × 1`, meaning cosine similarity reduces to the dot product itself.

This optimization is intentional.

Many embedding models normalize vectors specifically so similarity search can use dot products directly. Eliminating the normalization step improves retrieval efficiency, which becomes important when comparing a query against millions or billions of vectors.

Normalization behavior depends on the embedding model.

`sentence-transformers` normalizes vectors by default, but many embedding APIs do not. Before using dot product similarity directly, verify the vector norm with:

`np.linalg.norm(embedding)`

The result should be close to `1.0`.

If embeddings are not normalized, normalize them manually before indexing or comparison:

`embedding / np.linalg.norm(embedding)`

Failing to normalize embeddings does not produce invalid similarity scores, but it does make scores inconsistent across vectors with different magnitudes. This can introduce subtle retrieval errors that are difficult to diagnose later.

Now compare similarity scores across different document categories.

The next script computes dot product similarity between embeddings drawn from different parts of the corpus.

```
$ python ch02-embeddings/ch02.4-cross_topic_dot_product_similarity.py

Ops vs Ops:           0.2475
IT vs IT:             0.2679
Ops vs IT:            0.1199
Ops vs Compliance:    0.1478
```

Documents within the same semantic category produce higher similarity scores than documents from unrelated categories. The embedding space is beginning to reflect conceptual structure.

Notice that Operations versus Compliance (`0.1478`) scores slightly higher than Operations versus IT (`0.1199`). That relationship is plausible because Operations and Compliance both contain language related to policies, business rules, and data governance.

Now compare those same document pairs using **Euclidean distance**.

Euclidean distance measures the straight-line distance between points in vector space.

```
$ python ch02-embeddings/ch02.5-cross_topic_euclidean_distance_similarity.py

Ops vs Ops:           1.2268
IT vs IT:             1.2100
Ops vs IT:            1.3267
Ops vs Compliance:    1.3055
```

For normalized embeddings, cosine similarity and Euclidean distance produce equivalent rankings. Pairs with high cosine similarity will also have low Euclidean distance, and vice versa.

In practice, normalized embedding systems usually rely on dot product similarity because it is computationally efficient and preserves the same ordering.

For unnormalized embeddings, cosine similarity is generally the safer default because it accounts for vector magnitude automatically.

Euclidean distance is less common in embedding retrieval systems, but it remains useful in domains where absolute geometric distance is itself meaningful.

2.6 Visualizing embedding space with UMAP

Similarity scores are useful, but they are difficult to interpret in isolation.

Our corpus contains 30 documents represented as points in a 384-dimensional vector space. To inspect the structure of that space, we need a way to project those points into two dimensions without discarding the relationships that matter.

We cannot simply keep two dimensions and throw away the remaining 382. That would discard almost all of the information in the embeddings.

Instead, we use dimensionality reduction: a family of techniques that project high-dimensional data into a lower-dimensional space while preserving selected geometric relationships.

UMAP (Uniform Manifold Approximation and Projection) is commonly used for this purpose. It preserves local structure well, meaning points that are close together in the original high-dimensional space tend to remain close together in the two-dimensional projection.

This makes UMAP useful for inspecting clusters.

Now project the embeddings and plot the result.

```
$ python ch02-embeddings/ch02.6-visualize_umap.py
```

The script reduces the 384-dimensional embeddings to two dimensions, colors each point by topic, annotates representative documents, and saves the scatter plot.

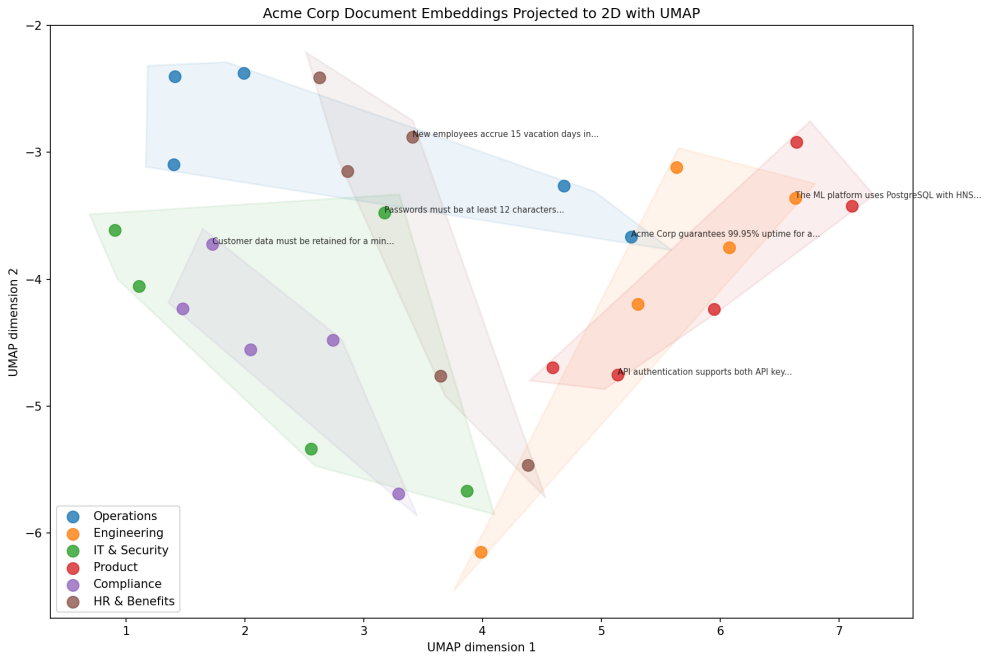


Figure 6. Acme Corp document embeddings projected to 2D using UMAP. Documents from the same category cluster together; overlapping clusters reveal semantic similarity across topics.

The resulting plot should show documents from the same category clustering together. Operations forms one group, Engineering another, and HR & Benefits, Product, IT & Security, and Compliance each form their own regions.

The model has recovered topical structure without receiving the category labels as input.

The most interesting cases appear near cluster boundaries. Compliance and IT & Security may overlap because both discuss data protection, access controls, and regulatory requirements. Engineering and Product may appear nearby because both contain language about technical platform behavior.

These boundary cases are useful. They show that embeddings capture semantic similarity, not organizational taxonomy.

For example, a Compliance sentence about "*access controls that restrict ePHI to authorized persons*" may land near an IT & Security sentence about "*access controls follow the principle of least privilege*," even though the two passages belong to different departments.

UMAP plots are diagnostic tools, not maps of absolute truth.

UMAP preserves local structure better than global structure. Nearby points usually indicate meaningful similarity, but large distances between clusters should not be over-interpreted. If two clusters appear far apart in the plot, that does not necessarily mean they are maximally distant in the original 384-dimensional space.

Within-cluster relationships are usually more informative than between-cluster distances.

UMAP is also stochastic. Running the same script multiple times may produce different layouts. The same clusters should still appear, but their position, rotation, and orientation on the canvas can change.

For reproducible plots, set a fixed random seed with `random_state=42` in the `umap.UMAP()` constructor.

2.7 Choosing an embedding model

We have used `all-MiniLM-L6-v2` for local embeddings and `text-embedding-3-small` through an API. Both worked on the test corpus. In production, however, embedding model choice has a direct effect on retrieval quality, latency, cost, and maintainability.

The decision is best made in three passes.

The first pass eliminates models that cannot meet your constraints. The second compares the remaining candidates on retrieval quality. The third determines whether the winner is practical to operate at scale.

2.7.1 Tier 1: Hard constraints

Check these before benchmarking. If a model fails any of these constraints, remove it from consideration regardless of its leaderboard score.

Context length. Every embedding model has a maximum input length. `all-MiniLM-L6-v2` accepts short inputs, while models such as `text-embedding-3-large` and `voyage-3-large` support much longer chunks. If a chunk exceeds the limit, it may be truncated, which means relevant information can disappear before retrieval ever begins. For long documents or large chunks, context length alone can eliminate many smaller models.

Language coverage. Match the model to the languages in your corpus. `all-MiniLM-L6-v2` is best suited to English. Multilingual models such as `multilingual-e5-large`, `BGE-M3`, and Cohere's multilingual embeddings represent many languages in a shared vector space, allowing queries and documents in different languages to match. For English-only systems, however, an English-specialized model often performs better than a multilingual model of comparable size.

Privacy. Some data cannot leave your infrastructure. Medical records, financial data, proprietary code, and regulated personal data may rule out API-based embedding models entirely. In those cases, local models are not a convenience; they are a requirement.

Domain specialization. General-purpose models often underperform on domain-specific retrieval. Code, law, finance, and biomedical text all have specialized vocabulary and matching patterns. If a strong domain-specific embedding model exists for your use case, start there. It may outperform a larger general model at lower cost.

2.7.2 Tier 2: Retrieval quality

Once the hard constraints have narrowed the field, compare the remaining models on retrieval performance.

Benchmark performance. Public benchmarks such as MTEB [4] are useful for shortlisting candidates, especially under the Retrieval task. But they are not a substitute for testing on your own data. A model that performs well on general benchmarks may rank differently on customer support tickets, legal contracts, source code, or internal runbooks.

When quality matters, embed a representative sample of your own documents with two or three candidate models and compare retrieval results directly.

Asymmetric query and document encoding. Some strong open models, including the E5 and BGE families, expect different prefixes for queries and passages. For example, the input may need to be prepended with `query:` or `passage:` depending on its role.

Forgetting these prefixes is a common production bug. The system still runs, but retrieval quality quietly drops.

[4] https://github.com/embeddings-benchmark/mteb

Fine-tunability. Open embedding models can be fine-tuned on your own (`query, relevant document`) pairs with tools such as sentence-transformers. For narrow domains with labeled examples, a fine-tuned smaller model can outperform a larger general model.

Some API providers also offer embedding fine-tuning. Others do not. If you expect to adapt the model to your domain, check this before committing.

2.7.3 Tier 3: Production costs

These factors may not decide which model is best, but they determine whether you can afford to run it.

Dimensionality. Dimensionality is the length of the output vector. Higher-dimensional vectors can capture more nuance, but they require more storage and are slower to search.

For example, a corpus of 100,000 documents stored as 1,536-dimensional float32 vectors requires roughly 586 MB for vectors alone. The same corpus at 384 dimensions requires roughly 147 MB.

At small scale, this difference barely matters. At 100 million documents, it becomes an infrastructure decision.

Matryoshka dimensionality. Some embedding models are trained so that their vectors degrade gracefully when truncated. Instead of storing the full vector, you can keep the first 512 or 256 dimensions and retain much of the retrieval quality.

This turns dimensionality from a fixed choice into an operational dial: higher dimensions for quality, lower dimensions for cost and speed.

Cost. Local models do not charge per token, but they require CPU or GPU capacity. API models charge per token embedded. That cost can become significant when re-embedding a large corpus after a schema change, model upgrade, or chunking change.

Deprecation risk. API models can be retired. When that happens, you may need to re-embed the entire corpus with a replacement model. At small scale, this is inconvenient. At large scale, it is a migration project.

Local models give you more control over upgrade timing, but you also take responsibility for maintaining the embedding stack yourself.

Do not choose an embedding model based on name recognition.

Start by eliminating models that fail your hard constraints. Then benchmark the survivors on your own data. Only after that should you optimize for cost, dimensionality, and operational risk.

Chapter 11 gives us the harness to do this systematically.

2.8 Similarity search from scratch

We now have everything needed to build a working search system.

The process is simple: embed the query with the same model used for the documents, compute cosine similarity between the query embedding and each document embedding, and return the highest-scoring results.

This is brute-force nearest neighbor search. It compares the query against every document. There is no index, no specialized data structure, and no approximation.

For 30 documents, brute force is instant. For 3 million documents, it becomes impractical. But as a teaching tool, it is the clearest possible demonstration of semantic search.

The search module lives in `ch02-embeddings/ch02.7-queries.py`. It loads the saved embeddings, defines a `search` function, encodes the query, computes dot-product similarity against each document embedding, and returns the top `K` results ranked by score.

Now test it with several queries. The script runs six queries through the search function and prints ranked results with similarity scores.

```
$ python ch02-embeddings/ch02.7-queries.py

Query: How do I get my money back?
--------------------------------------------------------------------------------
  1. [0.3654] Enterprise customers may request a full refund within 45 days of purchase.
  2. [0.2487] Upon account termination all customer data including backups is deleted within 90 days.
  3. [0.1446] Expense reports over $500 require approval from both the direct manager and department head.
  4. [0.0904] Customer data must be retained for a minimum of 7 years from the date of the last transaction.
  5. [0.0705] Access controls follow the principle of least privilege with quarterly reviews by system owners.


Query: What is the SLA uptime guarantee?
--------------------------------------------------------------------------------
  1. [0.6413] Acme Corp guarantees 99.95% uptime for all API endpoints under the enterprise SLA.
  2. [0.2987] Enterprise provisioning from signed contract to go-live takes 10 to 15 business days.
  3. [0.2794] Database backups target a recovery point objective of 1 hour and recovery time of 4 hours.
  4. [0.2709] Customer data must be retained for a minimum of 7 years from the date of the last transaction.
  5. [0.2629] New employees accrue 15 vacation days in their first year, increasing to 20 days after year two.


Query: How does the deployment pipeline work?
--------------------------------------------------------------------------------
  1. [0.3989] All production deployments use a blue-green model with canary rollout at 5% traffic.
  2. [0.2133] Standard shipping takes 5 to 7 business days from the date the order is processed.
  3. [0.2015] Enterprise provisioning from signed contract to go-live takes 10 to 15 business days.
  4. [0.1780] Webhooks deliver real-time event notifications to customer endpoints via HTTP POST.
  5. [0.1734] Account administrators manage users, configure SSO, and generate API keys in the admin console.


Query: What are the password requirements?
--------------------------------------------------------------------------------
```

```
  1. [0.5848] Passwords must be at least 12 characters and rotated every 90
  ↪  days per Section 4.2.1.
  2. [0.3207] Account administrators manage users, configure SSO, and generate
  ↪  API keys in the admin console.
  3. [0.2994] Access controls follow the principle of least privilege with
  ↪  quarterly reviews by system owners.
  4. [0.2989] The HIPAA Security Rule requires access controls that restrict
  ↪  ePHI to authorized persons.
  5. [0.2251] API authentication supports both API keys and OAuth 2.0
  ↪  authorization code flow.


Query: How do I authenticate API requests?
------------------------------------------------------------------------------
↪  ---
  1. [0.5979] API authentication supports both API keys and OAuth 2.0
  ↪  authorization code flow.
  2. [0.4772] Account administrators manage users, configure SSO, and generate
  ↪  API keys in the admin console.
  3. [0.3116] Acme Corp guarantees 99.95% uptime for all API endpoints under
  ↪  the enterprise SLA.
  4. [0.2947] Webhooks deliver real-time event notifications to customer
  ↪  endpoints via HTTP POST.
  5. [0.1475] Passwords must be at least 12 characters and rotated every 90
  ↪  days per Section 4.2.1.


Query: How many vacation days do new employees get?
------------------------------------------------------------------------------
↪  ---
  1. [0.8410] New employees accrue 15 vacation days in their first year,
  ↪  increasing to 20 days after year two.
  2. [0.3565] Remote work requires manager approval with a minimum of 2 days
  ↪  per week in the office.
  3. [0.3092] Passwords must be at least 12 characters and rotated every 90
  ↪  days per Section 4.2.1.
  4. [0.3021] Birthing parents receive 16 weeks of paid parental leave after 1
  ↪  year of employment.
  5. [0.2121] Customer data must be retained for a minimum of 7 years from the
  ↪  date of the last transaction.
```

All six queries return the correct top-ranked result.

The refund query finds the refund sentence even though the wording differs from the document. This is the semantic-only case the previous chapter was designed to demonstrate. The SLA query finds the uptime guarantee. The deployment query finds the blue-green deployment policy. The password query surfaces the relevant IT security rule.

The scores tell a second story.

The refund query scores only `0.3654`, much lower than the others, because the query and document share little vocabulary. The model has to infer that "*get my money back*" means "*request a refund.*" By contrast, "*How many vacation days do new employees get?*" scores `0.8410` because the query and matching document use nearly the same words.

Both results are correct. The score difference reflects how hard the embedding model had to work to bridge the wording gap.

Do not overinterpret absolute scores.

A top score of 0.36 is not necessarily weak if the next result is 0.25. What matters is the separation between the top result and the alternatives. Confident retrieval has a clear winner. Ambiguous retrieval has several candidates clustered at similar scores.

We will return to this distinction in Chapter 9, where reranking uses these gaps to decide whether to return one result or several.

Now try queries that are harder for embeddings.

The next script runs three deliberately problematic queries: a regulation not represented well in the corpus, a specific CVE identifier, and a negation query.

```
$ python ch02-embeddings/ch02.8-query_failing.py

Query: PDPA data residency rules for APAC
--------------------------------------------------------------------------------
  1. [0.3870] The HIPAA Security Rule requires access controls that restrict
     ePHI to authorized persons.
  2. [0.2328] Customer data must be retained for a minimum of 7 years from the
     date of the last transaction.
  3. [0.2190] Passwords must be at least 12 characters and rotated every 90
     days per Section 4.2.1.

Query: CVE-2024-1234
--------------------------------------------------------------------------------
  1. [0.2280] The annual performance review process includes quarterly
     check-ins and a 360-degree review.
  2. [0.2067] Passwords must be at least 12 characters and rotated every 90
     days per Section 4.2.1.
  3. [0.1432] Hybrid search combines BM25 keyword matching with semantic vector
     similarity for ranking.
```

```
14
15  Query: policies NOT related to compliance
16  ------------------------------------------------------------------------
    ↪  ---
17    1. [0.2677] The HIPAA Security Rule requires access controls that restrict
      ↪  ePHI to authorized persons.
18    2. [0.2258] Access controls follow the principle of least privilege with
      ↪  quarterly reviews by system owners.
19    3. [0.1770] EU customer personal data must be deleted or anonymized within 3
      ↪  years under GDPR rules.
```

These examples expose three weaknesses of embedding-based search.

The query "*PDPA data residency rules for* APAC" returns a HIPAA sentence as the top match. The model recognizes that the query is about data protection regulation, but it cannot reliably distinguish one regulatory framework from another when the target concept is missing or poorly represented in the corpus.

The query "CVE-2024-1234" is a specific identifier. The embedding model has no meaningful concept to attach to it. The result is effectively arbitrary: a performance review sentence appears first because the identifier does not map cleanly into semantic space.

The query "*policies* NOT *related to compliance*" returns compliance documents. This is a classic negation failure. The embedding contains the concept of compliance more strongly than the logical instruction to exclude it.

These failures are not implementation bugs. They are consequences of representing text as a single point in a continuous vector space.

Exact identifiers, rare terms, and logical operators often disappear during compression from text to vector. Chapter 8 addresses these limitations with hybrid search, which combines semantic similarity with keyword matching.

2.9 Summary

- Embeddings are lossy numerical representations of meaning. They cannot be decoded back into the original text, but they can be compared mathematically.
- Cosine similarity measures directional similarity rather than magnitude. For normalized vectors, cosine similarity and the dot product are equivalent.
- Embedding spaces are model-specific. Vectors produced by different embedding models are not compatible and should never be mixed in the same index.
- Bi-encoders embed queries and documents independently, which makes precomputing document embeddings and fast retrieval possible.
- Embeddings capture semantic similarity well for paraphrases and related concepts, but they struggle with exact identifiers, negation, rare terms, and some cross-format matching tasks.
- UMAP can project high-dimensional embeddings into two dimensions for visualization and exploratory analysis.
- Embedding model choice matters. Retrieval quality depends on factors such as dimensionality, context length, benchmark performance, language coverage, and domain specialization.
- Brute-force similarity search works by comparing a query embedding against every document embedding. It is exact, but it does not scale.

* * *

You can now turn text into vectors and retrieve semantically related documents through similarity search. But so far, we have worked with short, pre-prepared snippets. Real documents are messy: PDFs with dozens of pages, HTML with deeply nested sections, markdown files filled with code blocks and tables.

You cannot embed an entire 50-page report as a single vector and expect useful retrieval.

In the next chapter, we will tackle one of the highest-leverage decisions in a RAG pipeline: how to split documents into chunks that preserve meaning without destroying context.

Chapter 3. Chunking Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.1 The chunk size tradeoff

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.2 Fixed-size chunking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.3 Recursive character splitting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.4 Semantic chunking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.5 Document-structure-aware chunking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.6 Contextual chunking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.7 Comparing strategies: A retrieval test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

3.8 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

Chapter 4. Vector Storage and Indexing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.1 Exact vs. approximate nearest neighbor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.2 The speed-accuracy-memory tradeoff

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.3 Choosing a vector store

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.4 How HNSW works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.5 Building a FAISS index from scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.6 pgvector: Vectors in PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.7 Qdrant: A purpose-built vector database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.8 Tuning index parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.9 Putting it all together: the comparison benchmark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

4.10 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

Chapter 5. Building the Ingestion Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.1 The ingestion flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.2 Parsing real-world documents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.3 Text cleaning and normalization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.4 The full pipeline: Parse, clean, chunk, embed, store

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.5 Metadata extraction and storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.6 Idempotent re-ingestion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.7 Running the complete pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.

5.8 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/retrieval-augmented-generation.