# Retrieval-Augmented Generation: From Theory to Production

Generated with Plumora

# Table of Contents

# Chapter 1: Introduction to RAG: Why Retrieval-Augmented Generation Matters

Introduction to RAG: Why Retrieval-Augmented Generation Matters

Ask a large language model a question about your company's internal policies, or about something that happened last week, and one of two things tends to happen. Either it tells you it doesn't know, which is honest but useless, or it answers confidently and gets it wrong, which is far more dangerous. That second failure mode has a name: hallucination. It's not a bug that better prompting fixes, it's a direct consequence of how these models work. An LLM generates text by predicting the next most probable token given everything before it, based on patterns learned from its training data. It has no built-in mechanism for checking whether a claim is true, only for producing text that looks like the kind of text it was trained on. When it doesn't have the actual facts, it doesn't stop, it just keeps predicting plausible-sounding tokens.

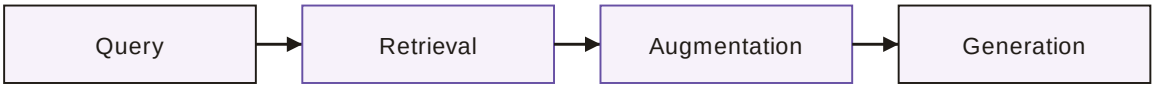
This connects to a second problem: staleness. Every model has a training cutoff, so it simply cannot know about anything that happened after that date. But the deeper issue for business applications isn't current events, it's that the model was never trained on your data at all. Your internal wikis, your support tickets, your product specs, your contracts: none of that was in the training set, and no amount of waiting for a newer model release fixes that, because it's private.

You might think the fix is just to paste everything into the prompt. Modern models do offer huge context windows now, some over a million tokens. But this runs into practical limits fast. Cost and latency scale with the number of tokens you send, so stuffing a hundred documents into every request gets expensive and slow at scale. There's also a well-documented effect sometimes called "lost in the middle," where models are demonstrably worse at using information buried in the center of a very long context than information near the beginning or end. A bigger context window doesn't guarantee the model actually uses what's in it.

So there are two tempting alternatives to RAG worth naming directly, because you'll be asked to justify not using them. The first is fine-tuning: retraining the model's weights on your own data. This can genuinely teach a model a new style or format, but it's a poor tool for injecting factual knowledge. Fine-tuned facts get blended into the weights statistically, which means the model can still misremember them, and there's no way to point at a source and say "this is where that answer came from." It's also expensive and slow to update: every time your documents change, you'd need to retrain. The second alternative is long-context stuffing, which we just covered: it scales poorly and doesn't guarantee attention where you need it.

RAG takes a different approach entirely. Instead of trying to bake knowledge into the model or cram it all into one giant prompt, it keeps the model frozen and fetches only the relevant slice of knowledge at the moment a question is asked, then hands that slice to the model as context. The model's job shrinks to something it's actually good at: reading a small set of relevant passages and synthesizing an answer, rather than trying to recall facts from memory.

That gives us the three stages that name this whole field, and it's worth being precise about what each one does.




Retrieval is the search step: given the user's query, find the documents or passages most likely to contain the answer, out of a much larger corpus. Augmentation is the assembly step: take those retrieved passages and construct a prompt that gives the model the context it needs, usually with some instructions about how to use that context. Generation is the familiar step: the LLM reads the augmented prompt and produces an answer.

Notice that this order isn't incidental, it's load-bearing. Augmentation literally cannot happen before retrieval, because it needs retrieved content to assemble the prompt around. And generation is entirely conditioned on what augmentation handed it. That means retrieval quality is a hard ceiling on the whole system. If retrieval pulls back the wrong passages, no amount of prompting cleverness or model capability in the generation step will produce a correct, grounded answer. You can swap in the best LLM available and it will confidently synthesize a wrong answer from wrong context. This is why the bulk of this course, and the bulk of real RAG engineering effort in production, lives in the retrieval side of the pipeline rather than the generation side.

This architecture also happens to solve problems that matter specifically for business deployment, beyond just answer quality. Because retrieval operates over a live document store rather than frozen model weights, updating your knowledge base is as simple as adding a document, no retraining required. Because generation is grounded in specific retrieved passages, you can cite sources back to the user, which matters enormously for trust and for compliance in regulated industries. And because retrieval is a distinct step, you can enforce access control there: a user only ever sees passages retrieved from documents they're permitted to read, before generation ever happens. None of that is available to you with fine-tuning or with a single giant undifferentiated context dump.

Over the rest of this course we'll build out each stage of this pipeline in depth: how retrieval actually finds "relevant" passages in the first place, how to store and search millions of them efficiently, how to assemble context that doesn't overwhelm the model, and how to evaluate whether the whole system is actually working. That starts next lesson with the piece that makes retrieval possible at all: how text gets turned into vectors, and how "relevance" becomes a number you can compute.

# Chapter 2: Embeddings: Math and Theory

Embeddings: Math and Theory

Last lesson ended with a promise: we'd explain how text becomes vectors, and how "relevance" turns into a number a machine can compute. That's the entire subject of this lesson, and it's worth taking slowly, because everything downstream in this course, chunking strategies, index selection, re-ranking, evaluation, is really just people arguing about how to get this one step right.

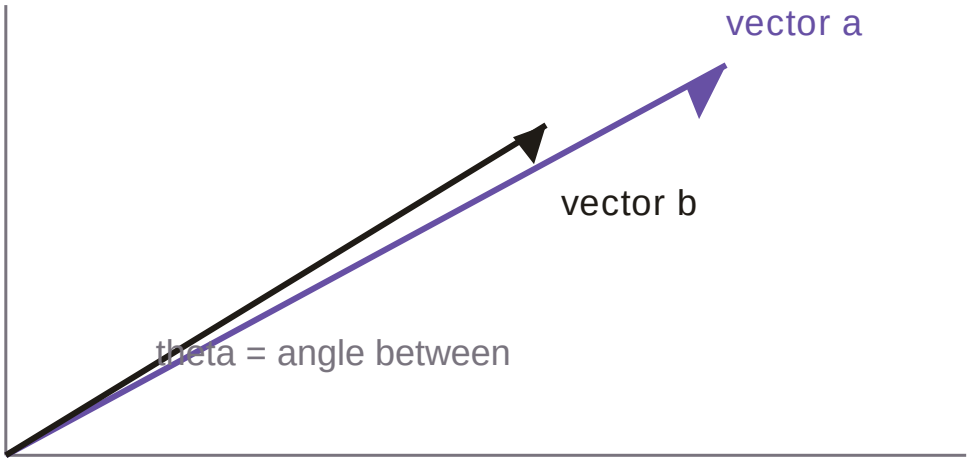
An embedding model is a function that takes a piece of text and outputs a vector, a list of numbers, living in a space with some fixed number of dimensions, commonly 384, 768, or 1536 depending on the model. You can think of this as a point in space. What makes it useful isn't the number of dimensions itself, it's that the model has been trained so that texts with similar meaning land near each other in that space, and texts with unrelated meaning land far apart. Nothing about this mapping is hand-designed. No dimension corresponds to a human-readable concept like "sentiment" or "topic." Meaning is encoded in the geometry: in the relative positions and directions of points, not in any single coordinate.

Where does that geometry come from? These models are trained with a contrastive objective. You give the network pairs of text known to be related, a question and its correct answer, two paraphrases of the same sentence, and pairs known to be unrelated. Training pushes related pairs closer together in the vector space and pushes unrelated pairs apart. Do this over billions of pairs and the network discovers a geometry where semantic similarity is expressed as spatial proximity. A simplified version of that objective looks like this:

$$\mathcal{L} = -\log \frac{e^{\text{sim}(a,p)}}{e^{\text{sim}(a,p)} + \sum_{n} e^{\text{sim}(a,n)}}$$

Here a is an anchor text, p is a positive match, and the n terms are negatives. Minimizing this loss means maximizing similarity to the positive relative to all the negatives. You don't need to derive this yourself to use embeddings well, but it's worth internalizing: the model was explicitly optimized around a similarity function, so understanding that function is understanding the model's whole reason for existing.

That brings us to how we actually measure "close." Cosine similarity is the default choice in most RAG systems. It measures the angle between two vectors, ignoring their length entirely.




```
import numpy as np

def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

def dot_product(a, b):
    return np.dot(a, b)

def euclidean_distance(a, b):
    return np.linalg.norm(a - b)
```

Three functions, three ways of asking the same question. Cosine similarity, shown above, normalizes out magnitude entirely, so a long, verbose chunk of text and a short one that mean the same thing will still score close to 1.0. Dot product skips the normalization step and just multiplies and sums corresponding components. That makes it faster to compute, and it's identical to cosine similarity when both vectors already have unit length, which is exactly why many embedding models, including OpenAI's, output normalized vectors by default: it lets a vector database use plain dot product for speed while still getting cosine's behavior. Euclidean distance measures straight-line distance between the two points rather than the angle between them.

These three aren't independent facts to memorize, they're mathematically linked for normalized vectors. If a and b both have unit length, then:

$$\|a - b\|^2 = 2 - 2\cos(\theta)$$

Smaller Euclidean distance corresponds exactly to larger cosine similarity in that case, so ranking results by one gives you the same order as ranking by the other. This is why you'll see vector databases offer "cosine," "dot product," and "L2" as index options and it barely matters for

normalized embeddings, but it matters enormously the moment your vectors aren't normalized, because then Euclidean distance starts reacting to magnitude in ways cosine similarity deliberately ignores. Getting this wrong is a real, common source of retrieval bugs: picking L2 distance against un-normalized vectors from a model that was trained assuming cosine comparison will quietly degrade your results with no error message anywhere.

Now, which embedding model should actually produce these vectors? This is a real tradeoff decision, not a solved question. OpenAI's text-embedding-3 models are strong general-purpose performers, hosted, easy to call over an API, and require no infrastructure of your own, but every embedding call leaves your network and costs money per token. Cohere's embedding models are comparable, with particular strength in multilingual and retrieval-tuned variants. Open-source sentence-transformers models, like the all-MiniLM or BGE families, run entirely on your own hardware, cost nothing per call, keep data in-house for compliance-sensitive use cases, and can be fine-tuned on your domain's language, but you take on the operational burden of hosting and maintaining them, and smaller open models sometimes trail the best hosted ones on general benchmarks like MTEB.

```
# Hosted model: one API call, no infrastructure to manage
from openai import OpenAI
client = OpenAI()
resp = client.embeddings.create(model="text-embedding-3-small", input="revenue grew
 12%")
vector = resp.data[0].embedding  # 1536 dimensions

# Open-source model: runs locally, no per-call cost
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")
vector = model.encode("revenue grew 12%")  # 384 dimensions
```

Notice the dimension counts differ, 1536 versus 384. Higher dimensionality generally means more capacity to represent nuance, but it also means more storage per vector and slower similarity computation at scale, so it's not simply "bigger is better." Choosing an embedding model means weighing quality benchmarks against cost, latency, data control, and the dimensionality you're willing to store and search across potentially millions of chunks.

That last part, searching across millions of vectors, is precisely where we're headed next. Now that you can compute similarity between any two vectors, the real engineering problem appears: how do you find the nearest neighbors to a query vector out of millions of candidates without comparing it to every single one? That's the subject of vector databases and indexing, starting next lesson.

# Chapter 3: Vector Databases and Indexing Strategies

Lesson 3: Vector Databases and Indexing Strategies

Last lesson left you with a working tool: given any two vectors, you can compute how similar they are. That's necessary, but it isn't sufficient. In a real RAG system you don't have two vectors, you have millions of them, one per chunk of every document you've ingested. When a query comes in, you need the top handful of chunks whose vectors are closest to the query vector. Compute cosine similarity against every single vector in your corpus, and you've just described a brute-force linear scan. It works. It also doesn't scale.

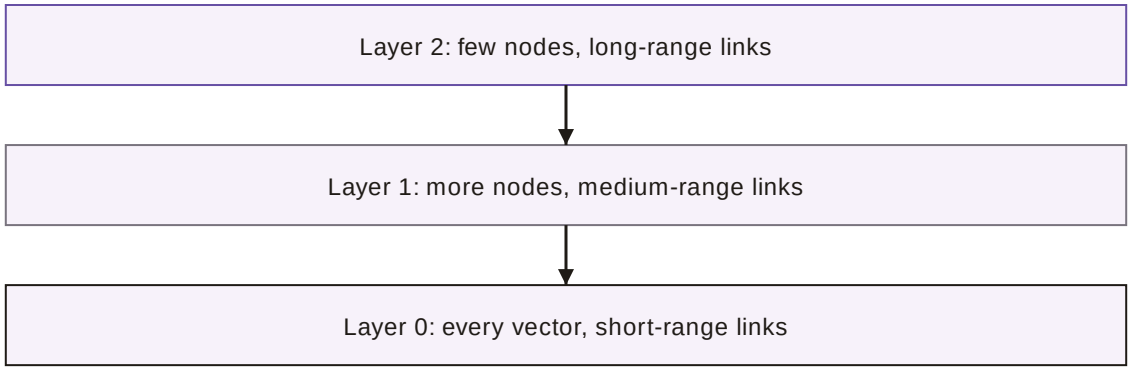
Do the arithmetic. Ten million chunks, 1536 dimensions each, computing a dot product for every one of them on every single query. That's billions of floating-point operations per query, and you need to do this in tens of milliseconds, not seconds, because there's a user waiting on the other end and an LLM call still to come after retrieval. Brute-force nearest neighbor search is exact, but exact doesn't matter if it's too slow to ship.

This is where approximate nearest neighbor search, ANN, enters the picture. The core idea is a trade: give up a small, controllable amount of accuracy in exchange for a massive amount of speed. Instead of comparing your query to every vector, you organize the vectors ahead of time into a structure that lets you skip most of them entirely, visiting only the candidates likely to be close. You'll occasionally miss the true nearest neighbor in favor of the second or third best, but with a well-tuned index that miss rate is small, and the speedup is often several orders of magnitude. Every vector database you'll encounter is, underneath, a wrapper around one of a small number of ANN indexing algorithms. Understanding those algorithms is what lets you tune the tradeoff instead of just accepting defaults.

The two you'll run into constantly are IVF and HNSW.

IVF, inverted file index, works by clustering. During index construction, you run something like k-means over your vectors to produce a set of cluster centroids, and every vector gets assigned to its nearest centroid. This partitions your space into cells, like a Voronoi diagram. At query time, instead of scanning everything, you first find the few centroids closest to your query vector, then only scan the vectors assigned to those cells. You've turned a scan of millions into a scan of a few thousand. The tuning knob is `nprobe`, the number of cells you check: check more cells and you get better recall at the cost of speed, check fewer and you get speed at the cost of occasionally missing a good match that happened to land in a cell you skipped.

HNSW, hierarchical navigable small world, takes a different approach: it builds a multi-layer graph. Picture a graph where each vector is a node, connected to its approximate neighbors. The bottom layer is dense, containing every vector. Each layer above it is a sparser subset, with longer-range connections that let you jump across the space quickly.




A search starts at the top, sparse layer, greedily walks toward the query vector, then drops down a layer and refines, repeating until it reaches the bottom. It's the same intuition as searching a sorted structure: coarse jumps first, fine-grained steps last. HNSW tends to give excellent recall and query speed, at the cost of higher memory usage and slower index-build time, since maintaining that graph structure isn't free. IVF tends to use less memory and build faster, but often needs more careful tuning to match HNSW's recall. Neither is universally "better"; they sit at different points on the same tradeoff curve.

Now, where do you actually run this? You have real choices, and they split roughly into managed services and libraries or self-hosted databases.

Pinecone is a fully managed, cloud-hosted vector database. You send it vectors, it handles indexing, scaling, and replication, and you never think about the underlying algorithm unless you want to. That convenience costs money per stored vector and per query, and your data leaves your infrastructure.

Weaviate and Milvus are open-source vector databases you can self-host or consume as a managed offering. Both support HNSW natively, both handle metadata filtering alongside vector search, and both are built for production scale with sharding and replication. The tradeoff versus Pinecone is operational: you gain control and avoid per-call fees, but you take on running and scaling the service yourself.

pgvector is different in spirit: it's an extension that adds vector similarity search directly into PostgreSQL. If you already run Postgres, this is enormously appealing, your embeddings live next to your relational data, in the same transactional database, with the same backup and access-control

story you already have. It supports both IVF and HNSW indexes. The tradeoff is that it's a general-purpose database with vector search bolted on, not a system purpose-built for billion-scale vector workloads, so at very large scale it can trail specialized stores on latency.

FAISS, from Meta, isn't a database at all, it's a library. It gives you the indexing algorithms themselves, IVF, HNSW, and several others, to embed directly inside your own application. There's no server, no persistence layer, no access control, you build that yourself. FAISS is the right choice when you want maximum control, are comfortable managing your own infrastructure, or are prototyping and want to understand the indexing mechanics directly.

Let's actually build one, using FAISS, so the theory has a concrete home.

```
import faiss
import numpy as np

dimension = 768
index = faiss.IndexHNSWFlat(dimension, 32)  # 32 = neighbors per node
index.hnsw.efConstruction = 200
index.hnsw.efSearch = 50

vectors = np.random.random((10000, dimension)).astype("float32")
index.add(vectors)

query = np.random.random((1, dimension)).astype("float32")
distances, indices = index.search(query, k=5)
print(indices)
```

The `32` controls how many graph connections each node keeps, more connections mean better recall and more memory. `efConstruction` controls how thorough the search is while building the graph, and `efSearch` controls how thorough it is at query time, this is your live recall-versus-speed dial, the same tradeoff we discussed above, now exposed as a parameter you can tune.

Here's the same idea, but living inside Postgres via pgvector, which is often the pragmatic choice if your business already runs relational data:

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE documents (
    id bigserial PRIMARY KEY,
    content text,
    embedding vector(768)
);

CREATE INDEX ON documents
    USING hnsw (embedding vector_cosine_ops);
```

```
SELECT content
FROM documents
ORDER BY embedding <=> '[0.12, 0.03, ...]'
LIMIT 5;
```

The `<=>` operator is pgvector's cosine distance operator, and the `USING hnsw` clause tells Postgres to build the same kind of graph index we just discussed, rather than scanning the table row by row.

So how do you actually choose, given all of this? Think in three dimensions: scale, latency, and operational cost. If you're under a few hundred thousand chunks and already run Postgres, pgvector is often the least-friction answer. If you need serious scale with minimal operations burden and don't mind paying for it, Pinecone. If you need scale, self-hosting, and control over cost at high volume, Weaviate or Milvus. If you're building something custom, embedding search inside your own service, or just want to feel the indexing mechanics under your hands, FAISS.

None of these choices are permanent or exclusive, either, teams commonly prototype with FAISS and migrate to a managed store once they understand their real query patterns. What matters is that you now understand what's happening underneath whichever name is on the box: vectors clustered or graphed ahead of time, so that a query only has to touch a small, well-chosen slice of your data. Next lesson, we go one level up the pipeline, to the question of what actually goes into those vectors in the first place: how you chunk documents, how much overlap to use, and how retrieval quality depends on decisions made before a single embedding is ever computed.

# Chapter 4: The Retrieval Pipeline: Chunking, Preprocessing, and Query Strategies

Lesson 4: The Retrieval Pipeline: Chunking, Preprocessing, and Query Strategies

Everything we've covered so far, embeddings, distance metrics, ANN indexes, assumes you already have a set of text chunks worth embedding. That assumption is doing more work than it looks like. In practice, the single biggest lever on retrieval quality isn't the embedding model or the index type, it's how you cut your documents into pieces in the first place. Get chunking wrong and no vector database on earth will save you, because you'll be searching over pieces of text that don't correspond to coherent ideas.

Let's start with the naive approach: fixed-size chunking. You pick a chunk size, say 500 characters or 200 tokens, and slice the document into consecutive pieces of that length. It's simple and fast, but it has an obvious flaw: it doesn't know where sentences end, where paragraphs break, or where one idea stops and another begins. You can easily end up with a chunk that starts mid-sentence and ends mid-thought, which produces an embedding that represents neither idea well.

The fix most teams reach for first is overlap. Instead of slicing at 0 to 500, 500 to 1000, and so on, you slide the window with some overlap, say 500 characters with 100 characters of overlap, so consecutive chunks share some text. This reduces the odds that an important sentence gets split exactly at a chunk boundary and orphaned from its context. Overlap isn't free, though, it increases storage and the number of vectors you need to embed and index, and too much overlap just means redundant retrieval results competing for your top-k slots.

A better approach is structure-aware, recursive chunking: split on the largest natural boundary available (say, double newlines for paragraphs), and only fall back to a smaller boundary (single newlines, then sentences, then raw characters) if a piece is still too large. This tends to respect the actual structure of the document instead of imposing an arbitrary grid over it.

```
def recursive_chunk(text, max_size=500, overlap=50, separators=None):
    separators = separators or ["\n\n", "\n", ". ", " "]
    if len(text) <= max_size:
        return [text]

    sep = next((s for s in separators if s in text), None)
    if sep is None:
        # No separator found, fall back to a hard character split.
        return [text[i:i + max_size] for i in range(0, len(text), max_size - overla
p)]
```

```
    pieces, current = [], ""
    for part in text.split(sep):
        candidate = current + sep + part if current else part
        if len(candidate) <= max_size:
            current = candidate
        else:
            if current:
                pieces.append(current)
            current = part
    if current:
        pieces.append(current)

    # Recurse on any piece that's still too big, using a finer separator.
    result = []
    for piece in pieces:
        if len(piece) > max_size:
            result.extend(recursive_chunk(piece, max_size, overlap, separators[1:])
)
        else:
            result.append(piece)
    return result
```

Notice the recursion here is doing the real work: it tries paragraph breaks first, and only degrades to sentence or word splits when it has to. This is exactly the strategy behind LangChain's `RecursiveCharacterTextSplitter` and similar utilities in LlamaIndex, we're just seeing the mechanics directly. The overlap parameter still applies within each fallback level, so you keep the continuity benefit without fighting the document's own structure.

One more theoretical point worth internalizing: chunk size is a tradeoff between precision and recall. Small chunks give you precise, focused embeddings, easier to match tightly to a specific query, but they lose surrounding context and multiply your vector count. Large chunks preserve context but dilute the embedding, a 2000-character chunk covering three different subtopics produces a vector that's a blurry average of all three, matching none of them well. There's no universal right answer, it depends on your documents and your queries, which is exactly why you'll build an evaluation harness in Lesson 8 rather than guessing.

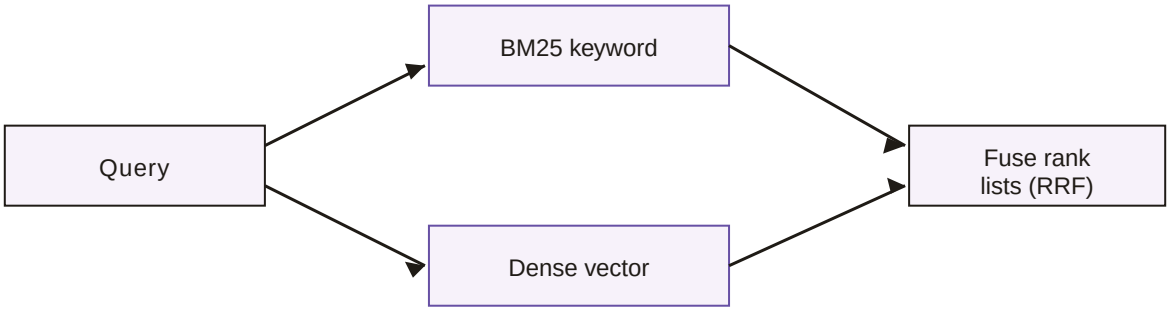
Chunking isn't only about splitting text, it's also about attaching metadata. Every chunk should carry structured fields alongside its text and vector: source document, section title, page number, author, date, access level. This metadata isn't for the embedding model, it's for filtering at query time. If a user asks about "the 2024 policy changes," you want to filter to chunks with `date >= 2024` before or during the similarity search, not rely on the embedding alone to somehow encode recency.

```
chunk_record = {
    "id": "doc42-chunk7",
```

```
    "text": "Employees may carry over up to 10 unused vacation days...",
    "embedding": [0.013, -0.081, ...],
    "metadata": {
        "source": "hr_handbook_2024.pdf",
        "section": "Time Off Policy",
        "page": 14,
        "date": "2024-01-15",
    },
}
```

Most vector databases, Pinecone, Weaviate, pgvector, all support filtered search: apply a `WHERE`-style predicate on metadata fields and only run the ANN search over the surviving subset. This is often more valuable in production than any amount of embedding model tuning, because it directly encodes business logic, don't show finance documents to interns, don't return last year's pricing sheet, that a similarity score alone can never capture.

Now, semantic similarity search has a real weakness: it's bad at exact matches. If a user searches for an error code like "ERR_429" or a specific product SKU, a dense embedding model may not place that query especially close to the chunk containing it, because the model reasons about meaning, not tokens. Keyword search, the old-fashioned kind, handles this perfectly. This is why production RAG systems increasingly use hybrid search: run a sparse keyword method like BM25 alongside your dense vector search, then combine the two ranked lists.




The fusion step is the interesting part. A common, robust method is Reciprocal Rank Fusion, which ignores the raw scores from each method (which live on incompatible scales) and instead uses each document's rank position:

$$\mathrm{RRF}(d) = \sum_{i} \frac{1}{k + \mathrm{rank}_i(d)}$$

Here rank_i(d) is the position of document d in ranked list i, and k is a small constant (commonly 60) that softens the impact of any single very-high rank. A document that shows up near the top of

both the keyword and semantic lists gets a high fused score even if the underlying similarity numbers were never comparable to begin with.

```
def reciprocal_rank_fusion(rank_lists, k=60):
    scores = {}
    for ranked_ids in rank_lists:
        for rank, doc_id in enumerate(ranked_ids):
            scores[doc_id] = scores.get(doc_id, 0) + 1 / (k + rank + 1)
    return sorted(scores.items(), key=lambda kv: kv[1], reverse=True)
```

Each `ranked_ids` list is just the document IDs from one retrieval method, ordered best-first. RRF sidesteps the score-normalization problem entirely, it only cares about ordering, which is what makes it a practical default for combining any two retrieval signals, not just BM25 and dense vectors.

There's one more stage worth naming here, even though we'll build it properly in Lesson 7: re-ranking. Hybrid retrieval typically pulls back a generous candidate set, say the top 50, using fast methods. A re-ranker, usually a cross-encoder that looks at the query and each candidate chunk together rather than as separately embedded vectors, then re-scores that smaller set with much higher precision, at the cost of being too slow to run over your entire corpus. The pattern is: cast a wide, cheap net first, then apply an expensive, accurate pass only to the survivors.

Put together, your retrieval pipeline now has real shape: documents get split by a structure-aware chunker with sensible overlap, each chunk is stored with metadata for filtering, a query hits both a keyword index and a vector index, the two result lists get fused, and (optionally) a re-ranker sharpens the final ordering. Next lesson, we take whatever chunks survive this pipeline and deal with the other half of RAG: assembling them into a prompt an LLM can actually use, without blowing past its context window or letting it hallucinate past what the retrieved text actually supports.

# Chapter 5: Generation: Prompt Engineering and Context Assembly

Lesson 5: Generation: Prompt Engineering and Context Assembly

By the end of Lesson 4, you had a ranked list of chunks sitting in memory: fused, maybe re-ranked, ready to use. But a list of chunks is not an answer. The generation stage is where those chunks get turned into a prompt, handed to an LLM, and turned into text a user actually reads. This is also where a surprising amount of visible RAG quality gets decided. You can build an excellent retriever and still ship a system that hallucinates constantly, simply because the prompt around your retrieved chunks doesn't force the model to stay grounded in them.

Let's start with a constraint you can't ignore: the context window. Every LLM call has a hard token budget, and that budget has to cover everything, not just your retrieved chunks. You need room for the system prompt, the retrieved context, any conversation history, the user's query, and the model's output. If you stuff in ten chunks and the model doesn't have room left to generate a full answer, you get a truncated, unhelpful response. So the practical task is: count tokens, decide how the budget is split, and only include as many chunks as actually fit.

```
import tiktoken

def fit_chunks_to_budget(chunks, query, system_prompt, model="gpt-4o", max_tokens=8
000, reserve_for_answer=1000):
    enc = tiktoken.encoding_for_model(model)
    used = len(enc.encode(system_prompt)) + len(enc.encode(query)) + reserve_for_an
swer
    budget = max_tokens - used
    selected = []
    for chunk in chunks:  # already ranked best-first
        cost = len(enc.encode(chunk["text"]))
        if cost > budget:
            break
        selected.append(chunk)
        budget -= cost
    return selected
```

Notice the loop stops as soon as a chunk won't fit rather than skipping it and trying the next one. That keeps the ordering intact: since chunks arrive best-first from retrieval, you always want the highest-ranked ones in the window, not whichever happens to be smallest. This is also why chunk size from Lesson 4 matters again here: smaller chunks let you fit more distinct pieces of evidence into the same budget, at the cost of each piece carrying less surrounding context.

Once you know which chunks fit, you need a template that turns them into a prompt the model will actually follow. A reliable structure has three parts: a system prompt that sets the model's role and grounding rules, a context section listing the retrieved chunks with identifiers, and the user's query at the end.


The key design decision inside that context section is how you label each chunk. If you just concatenate raw text, the model has no way to tell you which chunk supported which claim, and no way to signal that a claim isn't supported at all. Give each chunk an identifier, like its source document and position, and instruct the model to cite that identifier inline whenever it uses the chunk.

```
def build_prompt(chunks, query):
    system_prompt = (
        "Answer the user's question using only the numbered context below. "
        "Cite the source number in brackets, like [2], after every claim you make.
"
        "If the context does not contain the answer, say you don't know."
    )
    context_block = "\n\n".join(
        f"[{i+1}] Source: {c['source']}\n{c['text']}" for i, c in enumerate(chunks)
    )
    user_prompt = f"Context:\n{context_block}\n\nQuestion: {query}"
    return system_prompt, user_prompt

def call_llm(system_prompt, user_prompt, client, model="gpt-4o"):
    response = client.chat.completions.create(
        model=model,
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt},
        ],
        temperature=0,
    )
```

```
    return response.choices[0].message.content
```

A few things worth noticing here. Temperature is set to 0, because grounded factual answers benefit from determinism, not creative variance. The instruction to say "I don't know" is doing real work: without it, most models will confidently answer from parametric memory even when the retrieved context is irrelevant, which defeats the entire purpose of RAG. And citing by bracketed number rather than asking the model to quote source names verbatim keeps the output easy to parse and link back to your original documents in a UI.

It's worth being honest about what citations do and don't buy you. A citation tells you the model referenced a chunk, it does not prove the model's claim is actually supported by that chunk's content. Models can cite a source and still misstate what it says. This is called faithfulness, and checking it properly requires comparing generated claims against retrieved text systematically, which is exactly what we build in Lesson 8 when we construct an evaluation harness. For now, treat citations as a strong nudge toward grounded behavior and a debugging aid, not a guarantee.

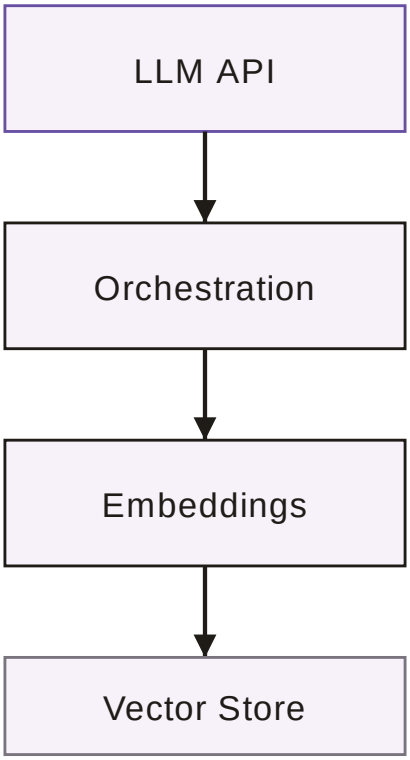
Put together, generation now has real shape: fit the highest-ranked chunks into your token budget, wrap them in a template that numbers them as sources, instruct the model to cite and to abstain when the context is insufficient, and call the model at low temperature. Next lesson, we step back from the mechanics and survey the broader ecosystem, the LLM APIs, orchestration frameworks, and vector stores you'd actually choose between to build all of this in a real stack.

# Chapter 6: Full RAG Tech Stack Landscape

Full RAG Tech Stack Landscape

For the last five lessons you've been building RAG by hand: writing your own cosine similarity function, standing up your own index, chunking documents with your own code, assembling prompts token by token. That was deliberate. You needed to see what's actually happening at each stage before you started trusting a library to do it for you. Starting today, we zoom out. In production, almost nobody builds every layer from scratch, and knowing the landscape of what's available, and what each choice costs you, is as important as knowing the math underneath it.

Think of a RAG system as four independent decisions stacked on top of each other: which LLM generates the answer, which framework (if any) orchestrates the pipeline, which model produces your embeddings, and which database stores and searches them. Each decision is largely separable from the others, which is exactly why the ecosystem has fragmented into dozens of vendors and open-source projects instead of one obvious winner.




Start with the LLM API layer, since it's the one your users actually see. OpenAI's GPT-4o family and Anthropic's Claude family are the two dominant managed options: strong instruction following, wide context windows, reliable function calling, and no infrastructure to manage. You pay per token and

you send your data to a third party, which matters if you're in a regulated industry. The open-source alternative, models like Llama 3 or Mistral, gets you full control over weights and data residency, but you either self-host, which means owning GPU infrastructure and inference optimization, or you use an inference provider like Together AI, Fireworks, or Groq, which gives you an API-shaped interface over an open-weight model. The decision usually comes down to three questions: does your data ever leave your network, what's your tolerance for per-token cost at scale, and do you need a capability, like very long context or strong multi-step reasoning, that only the frontier managed models currently deliver well.

Next is orchestration. LangChain, LlamaIndex, and Haystack all do roughly the same job: they wrap document loading, chunking, retrieval, and prompt assembly into reusable abstractions, so you can build a basic RAG pipeline in a dozen lines instead of the few hundred you've written across Lessons 4 and 5.

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.chains import RetrievalQA

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = FAISS.load_local("my_index", embeddings)
llm = ChatOpenAI(model="gpt-4o", temperature=0)

qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(search_kwargs={"k": 5}),
)
answer = qa_chain.invoke({"query": "What is our refund policy?"})
```

That's the same retrieval-then-generate flow you built manually, just compressed behind a `RetrievalQA` object. The convenience is real, especially for prototyping and for wiring up the dozens of document loaders and vector store integrations these frameworks maintain. The cost is also real: when something goes wrong deep inside a chain, you're debugging someone else's abstraction instead of your own code, and the prompt templates hidden inside these libraries aren't always the ones you'd choose for grounding and citation behavior. A common pattern in production teams is to prototype with a framework and then "unwrap" the critical path, the retrieval and prompt assembly logic, into plain code once the system stabilizes, keeping the framework only for the parts that genuinely save time, like document loaders. LlamaIndex leans more heavily into retrieval-specific tooling and indexing strategies; Haystack leans into pipeline composition for search-heavy applications. LangChain has the widest ecosystem of integrations, which is often the deciding factor.

The embedding layer is a decision you already have the vocabulary for from Lesson 2. OpenAI's `text-embedding-3` models and Cohere's `embed-v3` are strong managed defaults with good multilingual support. Open-source sentence transformers, like BGE or E5 variants hosted on Hugging Face, let you self-host, fine-tune on your domain's vocabulary, and avoid per-call embedding costs, at the price of hosting a model server yourself. A detail worth flagging here: your embedding model and your LLM don't have to come from the same vendor, and often shouldn't be chosen for that reason. Embedding quality for your specific domain matters more than brand consistency.

The vector store layer you already surveyed in Lesson 3: Pinecone and Weaviate as managed services, Milvus and pgvector for self-hosted control, FAISS for embedded, no-server setups. Nothing new to add here except that this choice interacts with the others less than you'd expect. A FAISS index doesn't care whether your embeddings came from OpenAI or a self-hosted model, and Pinecone doesn't care whether your LLM is Claude or Llama.

Here's a decision matrix for three common business scenarios, showing how these four layers tend to travel together in practice:

1. Fast-moving startup MVP: OpenAI GPT-4o, LangChain or LlamaIndex for speed, OpenAI embeddings, Pinecone. Optimized for time to first working prototype, not cost.
2. Cost-sensitive high-volume product: a smaller open-weight LLM via Together or Groq, thin custom orchestration code instead of a framework, open-source BGE embeddings self-hosted, pgvector riding on infrastructure you already run.
3. Regulated enterprise (healthcare, finance, legal): self-hosted Llama or a private-cloud model deployment, custom orchestration for auditability, self-hosted embeddings so no document content ever leaves the network, and Milvus or pgvector inside the company's own infrastructure boundary.

None of these combinations is "correct" in the abstract. They're correct relative to a constraint: speed to market, cost per query at your expected volume, or where your data is legally allowed to go. When you evaluate a real business use case, work through the four layers in that order, LLM, orchestration, embeddings, store, and for each one ask what constraint is binding hardest. That's the matrix, and it's the same one you'll apply in Lesson 9 when we build a complete application end to end.

Next lesson, we go deeper rather than wider: cross-encoder re-ranking, query rewriting, multi-hop retrieval, and agentic RAG, where the model itself decides when and what to retrieve.

# Chapter 7: Advanced RAG Techniques: Re-ranking, Query Rewriting, and Agentic RAG

Lesson 7: Advanced RAG Techniques: Re-ranking, Query Rewriting, and Agentic RAG

Everything up to this point has assumed a simple pipeline: embed the query, run approximate nearest neighbor search, take the top-k chunks, stuff them into a prompt. That pipeline works, and it'll get a prototype most of the way to useful. But it has three specific failure modes that show up the moment you put a RAG system in front of real users, and this lesson is about fixing each one directly.

The first failure mode: your top-k results are often good but not optimally ordered, and sometimes a highly relevant chunk sits at rank 8 instead of rank 2, outside the window you actually send to the LLM. The second: user queries are frequently vague, colloquial, or bundle multiple questions into one sentence, and a single embedding of that raw query doesn't retrieve well. The third: some questions genuinely require multiple retrieval steps, because the answer to sub-question A determines what you need to search for in sub-question B. Let's take these in order.

Cross-encoder re-ranking

Recall from Lesson 2 that your embedding model is a bi-encoder: it encodes the query and the document separately, into fixed vectors, and then you compare those vectors with cosine similarity. This is fast, because you can pre-compute document embeddings once and just embed the query at search time. But it's also lossy, because the model never actually looks at the query and document together. It has to compress all the information relevant to matching into a single vector ahead of time, without knowing what it'll be matched against.

A cross-encoder does the opposite: it takes the query and a candidate document as a single joint input and outputs a relevance score directly. This is much more accurate, because the model can attend across both texts simultaneously and pick up on interactions a bi-encoder would blur together. It's also much slower, because you can't precompute anything, you need one forward pass per query-document pair. That tradeoff is exactly why re-ranking is a second stage, not a replacement for vector search: use the cheap bi-encoder to pull, say, the top 50 candidates from millions of documents, then use the expensive cross-encoder to re-score just those 50 and take the top 5.

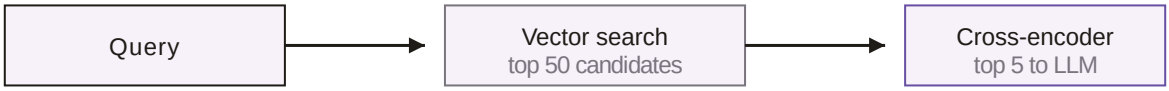
```
from sentence_transformers import CrossEncoder

reranker = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")

def rerank(query, candidates, top_k=5):
```

```
    pairs = [(query, doc) for doc in candidates]
    scores = reranker.predict(pairs)
    ranked = sorted(zip(candidates, scores), key=lambda x: x[1], reverse=True)
    return [doc for doc, score in ranked[:top_k]]
```

The `pairs` list is the key detail here: each candidate is bundled with the query into a tuple, and `predict` scores every pair jointly rather than comparing pre-computed vectors. Notice that this function takes candidates that already came from your vector database, it doesn't replace that step. In production, a common pattern is to retrieve 50, re-rank down to 5, and only send those 5 to the LLM. That extra stage routinely fixes cases where the "obviously right" chunk was buried at rank 12 in the raw vector search.




Query rewriting and decomposition

The second failure mode is upstream of retrieval entirely: the query itself. A user typing "what changed in the new policy" gives your embedding model very little semantic content to work with compared to the document chunks it's competing against, which are dense with specifics. The fix is to have an LLM rewrite the query before you embed it, expanding it into something closer to what the answer would actually contain.

One well-known version of this is HyDE (Hypothetical Document Embeddings): instead of embedding the raw query, you ask an LLM to generate a plausible hypothetical answer, and you embed that instead. The intuition is that a hypothetical answer looks more like a real document chunk than a short question does, so it lands closer to relevant chunks in vector space. A simpler and often just as effective version is query decomposition: break a compound question into its constituent sub-questions and retrieve for each separately.

```
def decompose_query(question, llm_call):
    prompt = f"""Break the following question into 1-3 simpler,
self-contained sub-questions needed to answer it fully.
Return one sub-question per line, no numbering.

Question: {question}"""
    response = llm_call(prompt)
    return [q.strip() for q in response.split("\n") if q.strip()]
```

Given something like "How does our refund policy differ between EU and US customers, and has it changed since 2023?", this produces two or three focused sub-questions, each of which retrieves far better on its own than the compound original would. You then run retrieval for each sub-question independently and merge the results, typically deduplicating by chunk ID before re-ranking the combined set.

Multi-hop retrieval and agentic RAG

Decomposition handles questions where the sub-questions are independent. Multi-hop retrieval handles the harder case, where sub-question B depends on the answer to sub-question A, so you can't generate both queries up front. Consider "What's the capital of the country that acquired the company mentioned in this document?" You need to retrieve and resolve "which company," then use that answer to formulate the next retrieval. This requires a loop: retrieve, read, decide whether you have enough information, and if not, formulate the next query based on what you just learned.

This loop is the essence of agentic RAG. Instead of a fixed pipeline, the LLM itself decides, at each step, whether to retrieve again, what to retrieve, and when it has enough context to answer. Architecturally this usually looks like a ReAct-style loop: the model produces a thought, optionally calls a retrieval tool, observes the result, and repeats until it decides to answer. The tradeoff versus the fixed pipelines from earlier lessons is straightforward: agentic RAG handles genuinely multi-step questions that a single retrieval pass cannot, at the cost of more LLM calls, higher latency, and a new failure mode where the agent retrieves unnecessarily or loops longer than it should. It's a tool for the subset of queries that actually need it, not a wholesale replacement for the simpler pipeline from Lesson 4.

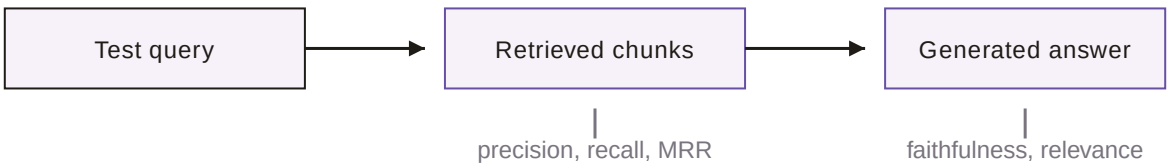
That's the toolbox: re-ranking to fix ordering, rewriting and decomposition to fix bad queries, multi-hop and agentic loops to fix questions that need iterative reasoning. What you don't yet have is a way to measure whether any of this actually made your system better, versus just more expensive. That's Lesson 8: evaluation, where we put numbers on retrieval quality and answer faithfulness so these techniques earn their place in production rather than being added on faith.

# Chapter 8: Evaluation and Debugging RAG Systems

Lesson 8: Evaluation and Debugging RAG Systems

Every technique from the last three lessons, re-ranking, query rewriting, agentic loops, made some claim about improving quality. But "this feels better" is not an evaluation strategy, and it definitely isn't something you can put in front of a stakeholder who's asking whether the system is ready to ship. This lesson is about replacing that feeling with numbers, and using those numbers to figure out exactly where a RAG pipeline is breaking.

The first thing to accept is that RAG has two independent failure surfaces, and they need two independent sets of metrics. Retrieval can fail by returning the wrong chunks, and generation can fail by writing a bad answer even from the right chunks. If you only measure the final answer, you can't tell which of these happened, and you'll end up tuning the wrong stage. A system with perfect retrieval and a sloppy prompt looks identical, from the outside, to a system with great prompting and terrible retrieval. Both produce a wrong answer. Only separate measurement tells them apart.




Start with retrieval, because if retrieval is broken nothing downstream can save you. To measure it, you need a golden dataset: a set of queries, each paired with the document chunks that would actually answer them. Building this by hand is real work, but there's no shortcut around it. Once you have it, three metrics do most of the work.

Precision at k asks: of the k chunks you retrieved, how many were actually relevant?

$$\text{Precision@}k = \frac{|\text{relevant} \cap \text{retrieved top } k|}{k}$$

Recall at k asks the opposite question: of all the relevant chunks that exist, how many did you actually surface in your top k?

$$\text{Recall@}k = \frac{|\text{relevant} \cap \text{retrieved top } k|}{|\text{relevant}|}$$

These pull against each other. You can boost recall by retrieving more chunks, but that drags precision down as you include more irrelevant ones. Which one you optimize for depends on your generation step: if your LLM is good at ignoring irrelevant context, favor recall; if irrelevant chunks derail it, favor precision and lean on the re-ranking from Lesson 7.

Mean Reciprocal Rank adds something precision and recall miss entirely: where in the ranking the first relevant chunk lands. If the right chunk is always retrieved but buried at position eight, precision and recall at k=5 won't catch that, but MRR will.

$$\text{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\text{rank}_i}$$

where rank_i is the position of the first relevant chunk for query i. A score close to 1.0 means the right chunk is usually first; a score near 0.2 means users are typically getting it around position five.

Here's a minimal implementation you can run against your own golden set:

```
def precision_at_k(retrieved, relevant, k):
    top_k = retrieved[:k]
    hits = sum(1 for doc in top_k if doc in relevant)
    return hits / k

def recall_at_k(retrieved, relevant, k):
    top_k = retrieved[:k]
    hits = sum(1 for doc in top_k if doc in relevant)
    return hits / len(relevant)

def reciprocal_rank(retrieved, relevant):
    for i, doc in enumerate(retrieved, start=1):
        if doc in relevant:
            return 1 / i
    return 0.0
```

Each function takes a list of retrieved document IDs, ranked by score, and a set of known-relevant IDs for that query. Run these over your whole test set and average them, and you have an objective baseline you can compare before and after every change to chunking, embeddings, or re-ranking.

Retrieval metrics only tell you whether the right material showed up. They say nothing about what the LLM did with it, and that's where the actual hallucination risk lives, since a model can be handed perfect context and still write something it isn't supported by. This is where faithfulness comes

in. Faithfulness measures whether every claim in the generated answer is actually backed by the retrieved context. The standard approach is to have an LLM decompose the answer into individual factual claims, then check each claim against the retrieved chunks: is this supported, contradicted, or absent? The faithfulness score is the fraction of claims that are supported.

Answer relevance is a different question: assuming the answer is faithful, does it actually address what the user asked? A faithful answer can still ramble, address a related-but-different question, or bury the actual answer under caveats. One common technique generates several hypothetical questions that the answer would be a good response to, embeds them, and compares their similarity to the original query. Low similarity means the answer drifted.

Computing these by hand, decomposing claims, generating hypothetical questions, is exactly the kind of repetitive, well-defined LLM task you don't want to reimplement yourself, which is why RAGAS exists. It packages faithfulness, answer relevancy, context precision, and context recall into a single evaluation call over a dataset of questions, contexts, and answers.

```
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy, context_precision, contex
t_recall
from datasets import Dataset

data = Dataset.from_dict({
    "question": questions,
    "answer": generated_answers,
    "contexts": retrieved_contexts,
    "ground_truth": reference_answers,
})

results = evaluate(
    data,
    metrics=[faithfulness, answer_relevancy, context_precision, context_recall],
)
print(results)
```

The `contexts` field is a list of retrieved chunks per question, and `ground_truth` is your reference answer, used for context recall specifically. RAGAS uses an LLM under the hood to do the claim-checking and question-generation work described above, so treat this as another dependency with its own cost and latency, not a free lunch.

The real value of running retrieval and generation metrics together is diagnosis. Low context recall with high faithfulness means retrieval is the bottleneck, the model is being faithful to context that's missing the answer. High context recall with low faithfulness means retrieval did its job and the model is drifting or fabricating on top of good material, a prompting or grounding problem from

Lesson 5. High faithfulness with low answer relevance means the model is accurately reporting facts that don't answer the question, often a sign the query needs rewriting per Lesson 7. This is the entire point of separating the two metric families: the failure pattern tells you which lesson's toolbox to reach into.

To make this operational, wrap it in a harness that runs automatically whenever you change something in the pipeline:

```
def run_eval(rag_pipeline, test_set):
    rows = []
    for item in test_set:
        contexts = rag_pipeline.retrieve(item["question"])
        answer = rag_pipeline.generate(item["question"], contexts)
        rows.append({
            "question": item["question"],
            "answer": answer,
            "contexts": [c.text for c in contexts],
            "ground_truth": item["reference_answer"],
        })
    return Dataset.from_list(rows)
```

Run this against your golden set before and after any change, whether it's a new chunk size, a different embedding model, or an added re-ranking step, and compare the metric deltas directly. That turns every tuning decision from the last four lessons into an experiment with a measurable outcome instead of a guess.

That's the measurement half of production readiness solved: you now have numbers for both retrieval and generation, and a harness that reruns them automatically. Lesson 9 puts everything together, embeddings, indexing, chunking, generation, advanced retrieval, and this evaluation harness, into one deployable application built around a concrete business scenario.

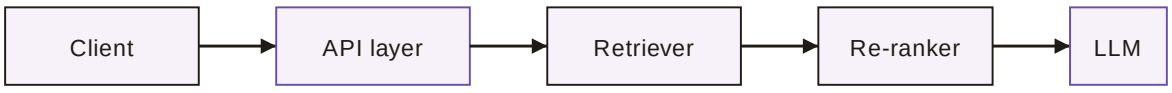
# Chapter 9: Building an End-to-End RAG Application for Business Use Cases

Building an End-to-End RAG Application for Business Use Cases

Every lesson so far has handed you one piece of the machine: how embeddings work, how vector indexes retrieve neighbors, how chunking shapes what gets retrieved, how prompts get assembled, how re-ranking and query rewriting sharpen results, how to measure whether any of it is actually working. This lesson stops adding pieces and starts wiring them together. We'll build a real service: an internal document Q&A assistant that a company could point at its policy handbook, engineering wiki, or support knowledge base and expose to employees through an API.

The business scenario matters because it forces real decisions. "Build a RAG system" is vague. "Build a system where an employee asks a question in Slack and gets an answer grounded in our internal docs, with a citation, in under three seconds" is a spec. That specificity is what turns a notebook full of experiments into an application with a shape.

Start with the architecture, because every implementation choice downstream flows from it. A production RAG service splits cleanly into two pipelines that run on different schedules: an ingestion pipeline that runs whenever documents change, and a query pipeline that runs on every user request. Conflating these two is the most common design mistake people make when they move from a demo notebook to a real service, because a notebook runs both in the same script and it's easy to forget they have completely different latency and reliability requirements.



Ingestion is offline and can afford to be slow and careful: parse documents, chunk them, embed them, upsert into the vector store, and record metadata like source and last-updated timestamp. The query pipeline, by contrast, has to be fast and defensive: embed the query, retrieve, re-rank, assemble a prompt, call the LLM, and return an answer with citations, all within a latency budget the business actually cares about. Designing these as separate services (or at least separate modules with separate deployment schedules) is what lets you scale, monitor, and debug them independently, a theme lesson 10 will push much further.

Let's build the ingestion side first, since nothing works without it.

```
# ingest.py
from pathlib import Path
from embeddings import embed_batch
from chunking import chunk_document
from vector_store import upsert

def ingest_directory(path: str, source_tag: str):
    for file in Path(path).glob("**/*.md"):
        text = file.read_text()
        chunks = chunk_document(text, chunk_size=500, overlap=50)
        vectors = embed_batch([c.text for c in chunks])
        records = [
            {
                "id": f"{file.stem}-{i}",
                "vector": vec,
                "text": chunk.text,
                "metadata": {"source": str(file), "tag": source_tag},
            }
            for i, (chunk, vec) in enumerate(zip(chunks, vectors))
        ]
        upsert(records)
```

Nothing here is new; you built chunk_document in lesson 4, embed_batch draws on lesson 2's embedding models, and upsert is the vector database interface from lesson 3. What's new is treating this as a standalone job you can re-run on a schedule or trigger from a webhook when someone edits a doc, rather than a step buried inside a request handler.

Now the query pipeline, exposed as an API. FastAPI is a reasonable default here because it gives you request validation and async support with very little ceremony, and it plays well with the async LLM and vector-store clients you've been using throughout the course.

```
# main.py
from fastapi import FastAPI
from pydantic import BaseModel
from retriever import retrieve
from reranker import rerank
```

```
from generator import generate_answer

app = FastAPI()

class QueryRequest(BaseModel):
    question: str
    top_k: int = 5

class QueryResponse(BaseModel):
    answer: str
    sources: list[str]

@app.post("/ask", response_model=QueryResponse)
async def ask(req: QueryRequest):
    candidates = await retrieve(req.question, top_k=20)
    top_chunks = rerank(req.question, candidates, keep=req.top_k)
    answer, sources = await generate_answer(req.question, top_chunks)
    return QueryResponse(answer=answer, sources=sources)
```

Notice the shape of this handler: it doesn't contain any retrieval logic, chunking logic, or prompt logic itself. It just orchestrates calls to modules you've already built in earlier lessons. That's deliberate. An end-to-end RAG application isn't one big clever function; it's a thin coordination layer sitting on top of well-tested, independently swappable components. If you want to try a different re-ranker or swap OpenAI embeddings for an open-source sentence transformer, you change one module and the handler doesn't notice.

The generate_answer function is where retrieval and generation actually meet, and it's worth looking at because citation handling lives here.

```
# generator.py
from llm_client import chat_completion

PROMPT_TEMPLATE = """Answer the question using only the context below.
Cite each fact with the source number in brackets, like [1].

Context:
{context}

Question: {question}
"""

async def generate_answer(question: str, chunks: list[dict]):
    context = "\n\n".join(
        f"[{i+1}] {c['text']} (source: {c['metadata']['source']})"
        for i, c in enumerate(chunks)
    )
    prompt = PROMPT_TEMPLATE.format(context=context, question=question)
    response = await chat_completion(prompt)
    sources = [c["metadata"]["source"] for c in chunks]
    return response, sources
```

This is the same context-assembly technique from lesson 5, but notice it now returns sources alongside the answer, because in a real internal tool, "trust me" isn't good enough; an employee reading the answer needs to see which policy document or wiki page it came from, and be able to click through and verify it.

One more integration point deserves attention: where evaluation fits into this running service, not just your offline test harness. You don't want to run RAGAS on every live request, that's too slow and too expensive, but you do want lightweight signals flowing continuously. A practical pattern is to log every request, its retrieved chunks, and its answer to a table, and run your lesson 8 evaluation harness against a sample of that log nightly.

```
# logging_hook.py
from datetime import datetime
from db import log_table

async def log_interaction(question, chunks, answer):
    log_table.insert({
        "timestamp": datetime.utcnow(),
        "question": question,
        "chunk_ids": [c["id"] for c in chunks],
        "answer": answer,
    })
```

Wire this into the /ask handler right after generate_answer returns, and you've turned your evaluation harness from something you run manually before a change into something that watches the system continuously in production, catching drift, like a document going stale or a new class of question the retriever handles poorly, before it becomes a user complaint.

Put together, ingest.py, main.py, generator.py, and logging_hook.py form a small but complete application: it ingests documents, answers questions with grounded citations, and instruments itself for ongoing measurement. Nothing about the architecture is exotic, and that's the point; a working RAG service is mostly disciplined plumbing between components you already understand deeply.

What it isn't yet is production-hardened. There's no authentication on that /ask endpoint, no rate limiting, no caching for repeated questions, no access control on which documents a given user is even allowed to retrieve from, and no answer to what happens when the LLM API times out under load. That's exactly where lesson 10 picks up: taking this working application and making it survive contact with real users, real traffic, and a real budget.

# Chapter 10: Production Considerations: Scaling, Security, and Cost Management

Lesson 10: Production Considerations: Scaling, Security, and Cost Management

Everything you built in lesson 9 works. It answers questions correctly, cites its sources, and logs enough data to evaluate itself. Ship it to ten internal users tomorrow and it'll probably be fine. Ship it to ten thousand, or expose it to customers who might ask it to reveal a document they're not supposed to see, and a different set of problems shows up. None of these problems are about retrieval quality anymore. They're about the system surviving contact with reality, and they tend to get discovered in production instead of code review if you don't plan for them now.

Let's start with scaling, because it clarifies where the actual bottlenecks live. Your FastAPI query handler is stateless, which means it scales the easy way: run more instances behind a load balancer, and each request is independent. That part is boring, in the good sense. The bottlenecks are elsewhere. Your vector database has to handle concurrent similarity searches, and if you're running HNSW with a large index, query latency under load depends heavily on how much of the index fits in memory and how many concurrent read threads your database allows. Your embedding provider and LLM provider both have rate limits measured in requests per minute and tokens per minute, and those limits are shared across every instance of your service. Ten replicas hammering the same OpenAI account will hit the same ceiling as one replica once total throughput exceeds it. The fix isn't more replicas; it's a request queue or client-side rate limiter that all instances respect together, plus retry logic with exponential backoff for the inevitable 429s.

Access control is the one that gets skipped most often, and it's the one that causes actual incidents. The naive approach is to retrieve normally and then filter out chunks the user isn't allowed to see before generation. Don't do that. By the time a chunk reaches your re-ranker, it's already been read into memory and scored, and a bug in the filtering step means a user sees a snippet of a document they had no business seeing. Permissions belong in the retrieval query itself, as a metadata filter enforced server-side, not as a post-hoc check on the client's behalf.

```
def retrieve_for_user(query_embedding, user, vector_store, k=10):
    allowed_ids = get_allowed_document_ids(user)  # from your auth service, not the
 request body
    return vector_store.query(
        vector=query_embedding,
        top_k=k,
        filter={"doc_id": {"$in": allowed_ids}},
    )
```

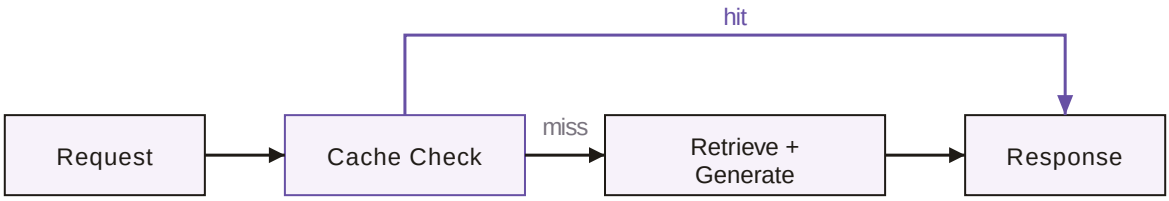
The key detail is where `allowed_ids` comes from. It's derived from the authenticated user's identity, looked up server-side, never trusted from a request parameter. If a client could pass its own list of allowed document IDs, access control would just be a suggestion. This also means your vector store needs metadata filtering that's actually efficient at scale, which is worth checking before you pick one back in lesson 3's territory: a filter that forces a full scan defeats the purpose of an approximate nearest neighbor index.

Monitoring in production is an extension of the evaluation harness from lesson 8, but with a different question in mind. Evaluation asks "is this answer good?" Monitoring asks "is the system healthy right now?" That means tracking latency percentiles, not averages, because a p99 of eight seconds hiding behind a p50 of half a second means some fraction of your users are having a bad time even though your dashboard looks green. It means tracking error rates per stage, retrieval failures, LLM timeouts, empty result sets, separately, because "the answer was wrong" and "the vector database timed out" need completely different responses. And it means tracking cost per request as a first-class metric alongside latency, because in a RAG system cost and quality are in constant tension, and you want to see that tradeoff on the same dashboard rather than discovering it in an invoice.

Caching is where you win back both latency and cost at once, and it works at two different layers. The cheap, obvious layer is exact-match caching: if the same query string comes in twice, return the cached answer. The more valuable layer is semantic caching: two different phrasings of the same question shouldn't both trigger a full retrieval-and-generation cycle.

```
def get_cached_answer(query, embed_fn, cache, threshold=0.95):
    query_vec = embed_fn(query)
    for cached_query, cached_vec, answer in cache.recent_entries():
        if cosine_similarity(query_vec, cached_vec) >= threshold:
            return answer
    return None
```

This checks the incoming query's embedding against recently answered queries, and if something is close enough, above a high similarity threshold like 0.95, it reuses the stored answer instead of running the full pipeline again. The threshold matters more than it looks: set it too low and you'll serve a stale, slightly-wrong answer to a question that actually needed fresh retrieval; set it too high and you barely save anything. Tune it against your evaluation set, the same way you'd tune a retrieval parameter. Also make sure cached answers respect access control: a semantic cache shared across users with different document permissions is its own leak waiting to happen, so key the cache by user or permission scope, not just by query text.




Cost management, at this point, is mostly bookkeeping applied consistently. Every LLM call has a token cost you can compute before you send it, since you know your context length and can estimate output length. Log that estimated cost alongside every request, and you can answer questions like "which query patterns are expensive" instead of just watching a monthly bill climb. A few concrete levers: cap the number of chunks you pass to generation rather than stuffing the full context window every time, since more context isn't free and doesn't always improve the answer; use a cheaper, smaller model for query rewriting and re-ranking, saving the expensive model for the final generation step; and batch embedding calls during ingestion rather than sending one document at a time, since most embedding APIs price and rate-limit in a way that rewards batching.

Here's the checklist this lesson leaves you with. Rate-limit and queue requests to external APIs across all replicas, not per instance. Enforce document access control inside the retrieval query, never as a filter after the fact. Track latency percentiles and error rates per pipeline stage, not just overall averages. Cache aggressively, both exact and semantic, scoped correctly to user permissions. Log estimated cost per request and review it as often as you review quality metrics. None of this is glamorous, and that's consistent with everything else in this course: a production RAG system is disciplined engineering applied to a pipeline you now understand end to end, from the embedding math in lesson 2 to the hardening checklist here in lesson 10.