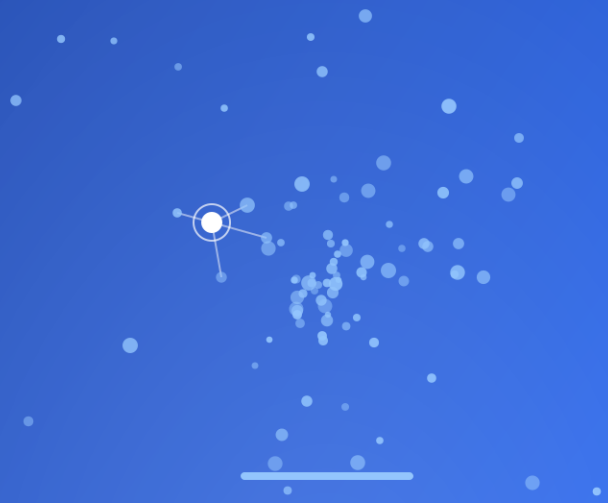


# Search Semantically

## Large Language Models & Semantic Search

A beginner-to-expert, hands-on course

**Retrieve** → **Re-rank** → **Generate**

**14 chapters • runnable notebooks • open-source**

Dr. Mahdi Habibi · v1.0.0

# Contents

# Preface

**Search Semantically** is a hands-on course on how modern search works — from classic keyword matching to embeddings, dense retrieval, re-ranking, evaluation, and Retrieval-Augmented Generation (RAG). Every chapter pairs intuitive explanations (analogies, diagrams, worked examples) with runnable code, so you learn the idea *and* how to build it.

The toolkit is open-source and key-free (sentence-transformers, FAISS, Chroma, rank-bm25), so everything runs locally. Difficulty is marked per chapter: Beginner, Intermediate, Advanced.

By the end you will be able to: explain how keyword search works and why it falls short; turn text into meaning-vectors with embeddings; build a semantic search engine; scale it with a vector database; sharpen it with re-ranking and hybrid search; evaluate it properly; and plug it into an LLM to build a grounded question-answering system.

Each chapter ends with *Check your understanding* questions; **model answers are collected in the Answers appendix** near the end of this book. This is the companion text to the code repository of the same name; each chapter maps to a folder with a notebook you can run.

# Chapter 0 — Introduction

> **Level:** Beginner - **Status:** Complete **Prerequisites:** a little Python. No ML background needed. **You'll build:** a mental model of the whole search pipeline.

---

## At a glance

This chapter sets the stage. By the end you'll understand *what* semantic search is, *why* LLMs changed search, and *how* the rest of the course fits together — so every later chapter has a place to land.

---

## The motivation

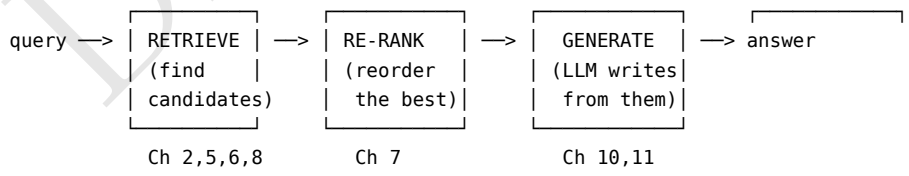
Imagine you run a website with thousands of articles — picture a small Wikipedia, or a store with a big product catalog. People need to *find* things. For decades the answer was **keyword search**: match the words in the query to the words in the documents.

That works until someone searches for *"how do I stop my laptop from dying so fast"* and your best article is titled *"Improving battery life on portable computers."* Not a single important word matches — yet it's exactly the right answer. **Keyword search matches words; people search by meaning.** Closing that gap is what this course is about.

---

## Intuition first: the four-step pipeline

Everything we build fits one simple pipeline. Keep this picture in your head:




- **Retrieve** — quickly pull a handful of likely-relevant documents from the whole corpus.
- **Re-rank** *(optional)* — carefully reorder those candidates so the best is on top.
- **Generate** *(optional)* — hand the top results to an LLM so it can write a direct answer grounded in them (this is **RAG**).

Some systems stop after Retrieve (a classic search box). Others go all the way to Generate (an AI assistant that answers questions). You'll build each stage and understand the trade-offs.

---

## What "semantic" adds

The single most important upgrade is in the **Retrieve** step. Instead of matching words, we convert text into **embeddings** — vectors of numbers that capture *meaning* — and find documents whose vectors are closest to the query's vector. "Battery life" and "stop my laptop dying" land near each other in this space even with no shared words. That's **semantic search**, and Chapters 3–6 build it up from scratch.

---

## How to use this repo

1. Do the setup once.
2. Read chapters in order if you're new; jump around using the roadmap if you're experienced.
3. Read each chapter's README, then run its `notebooks/` to see the idea in code.
4. Keep the glossary open in a tab.

---

## Hands-on

Notebook: `notebooks/00_pipeline_tour.ipynb` — a 5-minute end-to-end demo: stand up a search system in three lines with the course's `SearchStack`, ask it questions in plain English, and watch it answer by *meaning* — so you see the destination before learning to build each part.

---

## Key terms

Large Language Model (LLM), embedding, retrieval, ranking, semantic search, RAG. *(See GLOSSARY.)*

---

## Check your understanding

1. In one sentence, why does keyword search miss relevant results?
2. What does the "Retrieve" step do, and how does "semantic" change it?
3. Which pipeline stages are optional, and when would you skip them?

---

## References


- DeepLearning.AI x Cohere — *Large Language Models with Semantic Search* (course inspiration).
- Links to primary sources are added as we write later chapters.

# Chapter 1 — Foundations of Information Retrieval

> **Level:** Beginner - **Status:** Complete **Prerequisites:** Chapter 0. **You'll build:** the vocabulary and the mental model that the whole course stands on.

---

## At a glance

Before we write a single ranking formula, we need to agree on *what search is* and *what makes it hard*. This chapter defines the handful of words we'll use in every later chapter (query, document, corpus, relevance, retrieval, ranking), lays out the shape of a search system, and draws the single most important distinction in the course: **matching words vs. matching meaning.**

---

## The motivation: a user with a question

Search always starts the same way: a person has something they want to know — an **information need** — and somewhere out there is text that could satisfy it. The catch is that the person doesn't hand you their need directly. They hand you a *query*: a short, imperfect, often ambiguous stand-in for what they actually mean.

> Information need: *"I have a pounding pain on one side of my head — what is it?"* Query they type: *"strong pain in the side of the head"*

Your job is to bridge the gap between that messy query and the documents that truly answer it. Everything in this course is a better and better bridge.

---

## The core vocabulary

These five words appear constantly. Learn them once here.

| Term | Meaning | Example (this course's corpus) |
|---|---|---|
| **Information need** | What the user actually wants to know | "Why does grass look green?" |
| **Query** | The text the user submits | `what color is the grass` |

| Term | Meaning | Example (this course's corpus) |
|---|---|---|
| **Document** | One retrievable unit of text | The passage titled *"The color of grass"* |
| **Corpus** | The whole collection you search | Our 16 sample passages (later, millions) |
| **Relevance** | How well a document satisfies the need | The grass passage is relevant; the whale one isn't |

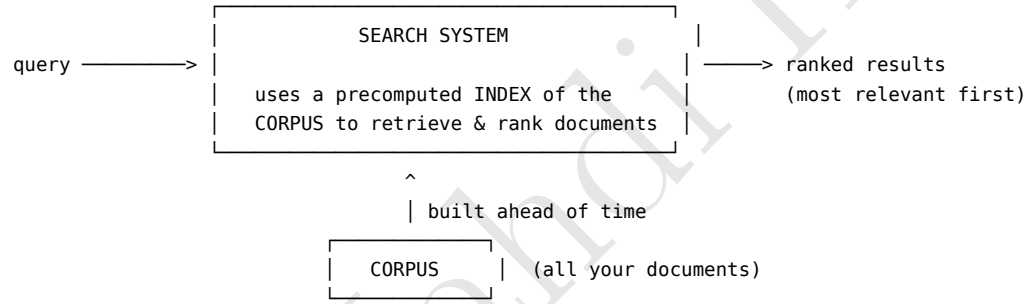
Two more describe the *actions* a search system takes:

- **Retrieval** — pulling a set of candidate documents out of the corpus for a query.
- **Ranking** — ordering those candidates so the most relevant sit at the top.

---

## The shape of a search system

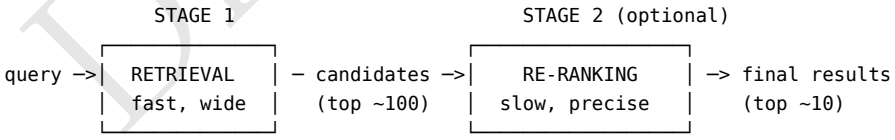
At the highest level, every search system looks like this:




The key idea hiding in that picture: **the corpus is processed *before* any query arrives.** The system builds a data structure called an **index** in advance, so that at query time it can answer in *milliseconds* instead of re-reading every document. (We build a concrete index — the *inverted index* — in Chapter 2.)

### Search is usually multi-stage

Real systems rarely do everything in one shot. They split the work into stages:




- **Stage 1 — Retrieval** casts a wide net *cheaply*. It has to look at the whole corpus, so it must be fast. Classic keyword retrieval (BM25, Chapter 2) lives here, and so does semantic dense retrieval (Chapter 5).
- **Stage 2 — Re-ranking** takes the small candidate set and reorders it *carefully* using a more expensive, more accurate model, and can fold in extra signals like popularity or freshness (Chapter 7).

You'll meet both stages repeatedly. Keep the pattern in mind: **retrieve wide and cheap, then re-rank narrow and precise.**

## The one distinction that matters most: words vs. meaning

Here is the crux of the entire course, in one example.

A naïve search system answers a query by counting **shared words** between the query and each document. Take the query `what color is the grass` against a tiny archive:

```
Query:  what color is the grass

D0  "the grass is green ..."          shared: the, grass, is    ->  3   [yes]  best
D1  "the sky is blue ..."             shared: the, is           ->  2
D2  "the capital of canada is ..."    shared: the, is           ->  2
D3  "a whale is a mammal ..."         shared: is                ->  1
```

Word-counting gets this right — D0 wins — because the answer literally repeats the query's words. This is the whole idea behind **keyword (lexical) search**, and it's remarkably effective and fast. It powered the web for decades.

But now flip the example. Same idea, different phrasing:

```
Query:  strong pain in the side of the head

D2  "A migraine produces a sharp, throbbing sensation localized to one temple ..."
       shared meaningful words with the query: (basically none)   ->  ~0  [no]  missed!
```

That document is *exactly* the right answer — but it shares almost no words with the query. "Temple" means "side of the head"; "sensation" means "pain"; "migraine" is the "strong" thing. A word-counter is blind to all of it. This is the **vocabulary mismatch problem**, and it's the fundamental limitation of keyword search.

> **Matching words is not the same as matching meaning.** Keyword search matches words. The rest of this course is the story of teaching machines to match *meaning* — using language models and embeddings (Chapters 3–6) — and of combining both worlds (Chapter 8).

---

## Hands-on

Notebook: `notebooks/01_words_vs_meaning.ipynb`

You'll load the course corpus and write a five-line "search" that just counts shared words. You'll watch it nail the *grass* query and completely miss the *headache* query — feeling the vocabulary-mismatch problem for yourself. That failure is the motivation for everything that follows.

---

## Going deeper

- **Relevance is not binary.** In practice relevance is graded (perfect / good / marginal / irrelevant) and *subjective* — two annotators may disagree. Chapter 9 turns this into measurable metrics.
- **Precision vs. recall tension.** A system can favor returning only sure-thing results (high precision) or casting a wide net to miss nothing (high recall). Different applications sit at different points on this trade-off.

- **The query is a lossy encoding of the need.** Much of modern search (query rewriting, expansion, HyDE — Chapter 12) is really about reconstructing the underlying need from a thin query.

---

## Pitfalls & gotchas

- **Confusing retrieval with ranking.** Retrieval decides *which* documents are candidates; ranking decides their *order*. A perfect ranker can't save you if retrieval never surfaced the right document.
- **Assuming more shared words = more relevant.** Common words ("the", "is") are shared by almost everything and signal nothing. Chapter 2's IDF fixes exactly this.
- **Forgetting the index is precomputed.** Beginners imagine search scans every document live. It doesn't — the heavy lifting happens at indexing time.

---

## Key terms

information need, query, document, corpus, relevance, retrieval, ranking, index, vocabulary mismatch, retrieval stage, re-ranking stage. *(See GLOSSARY.)*

---

## Check your understanding

1. Explain the difference between an *information need* and a *query* in your own words.
2. Why does a search system build an index *before* any query arrives?
3. Name the two classic stages of a search system and what each optimizes for.
4. Give your own example of two texts that mean the same thing but share no keywords.
5. Why is counting shared words a poor measure of relevance when the words are common?

---

## References


- Manning, Raghavan & Schütze, *Introduction to Information Retrieval* (free online) — the classic textbook for these foundations.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 1 (inspiration).

# Chapter 2 — Keyword / Lexical Search

**Level:** Beginner - **Status:** Complete **Prerequisites:** Chapter 1. **You'll build:** a working keyword search engine — inverted index, TF-IDF, and BM25 — from scratch, then see exactly where it breaks.

---

## At a glance

Keyword search (a.k.a. **lexical search**) matches the literal words in a query against the words in your documents. It's fast, transparent, needs no machine learning, and still powers a huge share of search today. Understanding it deeply pays off twice: it's a strong baseline you will keep using (in hybrid search, Chapter 8), and its failure modes are the exact reason embeddings were invented (Chapters 3–6).

We'll build three things, each better than the last:

1. **Word-overlap search** — the naïve "count shared words" idea, to see the intuition.
2. **TF-IDF** — weight words by how *informative* they are.
3. **BM25** — the refined, industry-standard scoring function.

And we'll do it all over our own sample corpus, no external services.

---

## The motivation

Think of any search box you used today — Spotify, YouTube, a shop, your company wiki. Behind most of them, at least at the first stage, is keyword search. The task: given a query, score every document by how well its words match, and return the top few. The genius is doing this over millions of documents in *milliseconds*. Two ideas make that possible — a clever data structure (the inverted index) and a good scoring function (BM25). Let's build both.

---

## Step 0 — Preparing text: tokenization

Before matching, we normalize text into **tokens** (roughly, words):

```
"What color is the Grass?"  ──tokenize──>  ["what", "color", "is", "the", "grass"]
```

Typical steps: lowercase everything, split on non-letters, and often drop **stop words** (ultra-common words like *the, is, of* that carry little meaning) and apply **stemming** (reduce *running*

-> *run* so variants match). Our helper `src/corpus.py` (at the repo root) does a simple version you can read in full.

> Why this matters: without normalization, "Grass" != "grass" and "running" != "run", and your matches silently fall apart.

---

## Step 1 — The inverted index: how search is *fast*

Imagine you had to answer a query by re-reading all 10 million documents every time. Hopeless. Instead we build, **once, ahead of time**, an **inverted index**: a table mapping each word to the list of documents that contain it (and how often).

```
FORWARD view (what we start with)        INVERTED INDEX (what we build)
─────────────────────────────────        ──────────────────────────────────
D0: "grass is green"                     term       ->  postings (doc: count)
D1: "the sky is blue"                    ──────────────────────────────────
D4: "the blue whale ..."                 "grass"    ->  {D0:1}
                                         "green"    ->  {D0:1}
                                         "blue"     ->  {D1:1, D4:1}
                                         "whale"    ->  {D4:1}
                                         "sky"      ->  {D1:1}
```

Now a query like `blue` is answered by a single lookup — "blue -> {D1, D4}" — instead of a full scan. This is *the* trick behind instant search. The word -> documents direction is why it's called "inverted" (a normal index goes document -> words).

---

## Step 2 — Scoring: not all shared words are equal

The naïve approach counts shared words. But we saw the flaw in Chapter 1: the words *the* and *is* are shared by nearly every document and tell us nothing. We need to weight words by how *informative* they are. That's **TF-IDF**.

### Term Frequency (TF)

> **Intuition:** a word that appears many times in a document probably matters to that document.

TF = how often term $t$ appears in document $d$. If "whale" appears 3 times in a passage, that passage is likely about whales.

### Inverse Document Frequency (IDF)

> **Intuition:** a word that appears in *few* documents is more distinctive and useful for telling documents apart. A word in *every* document is useless.

$$\mathrm{IDF}(t) = \log \frac{N}{\mathrm{df}(t)}$$

where $N$ is the number of documents and $\mathrm{df}(t)$ is how many documents contain $t$.

### TF-IDF = TF x IDF

Multiply them: frequent-in-this-doc **and** rare-across-the-corpus = high score.

### Worked example (do the numbers yourself)

Take a 5-document toy corpus and the query `what color is the grass`:

```
D0 "the grass is green"          D1 "the sky is blue"       D2 "the capital of canada is ottawa"
D3 "a whale is a mammal"         D4 "tomorrow is saturday"
N = 5 documents
```

Compute IDF for the query's content words (log base $e$):

| term | df (docs containing it) | IDF = ln(N/df) |
|---|---|---|
| is | 5 | ln(5/5) = **0.00** |
| the | 3 (D0, D1, D2) | ln(5/3) ~= **0.51** |
| color | 0 | — (not in corpus) |
| grass | 1 (D0) | ln(5/1) ~= **1.61** |

See what happened? **"is" contributes nothing** (it's everywhere), **"grass" contributes the most** (it's rare and distinctive). Now score D0 by summing TF-IDF over query terms present:

```
score(D0) = TF(grass)-IDF(grass) + TF(the)-IDF(the) + TF(is)-IDF(is)
          =    1 - 1.61          +   1 - 0.51       +   1 - 0.00
          =  2.12    <-  highest, and driven almost entirely by "grass"
```

Every other document only shares "the"/"is", so they score <= 0.51. TF-IDF gets the ranking right *for the right reason*: the rare, meaningful word did the work.

---

## Step 3 — BM25: the workhorse

TF-IDF has two weaknesses that **BM25** fixes, and both come from common sense:

1. **Term saturation.** Seeing "whale" 10 times isn't 10x more relevant than seeing it once — there are diminishing returns. TF-IDF grows linearly forever; BM25 flattens out.
2. **Document length.** A long document naturally repeats words more, unfairly inflating raw TF. BM25 normalizes by length, comparing a document to the *average* document length.

The BM25 score of document $d$ for query $q$:

$$\text{BM25}(q,d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t,d)\,(k_1+1)}{f(t,d) + k_1\left(1 - b + b\,\frac{|d|}{\text{avgdl}}\right)}$$

Don't be scared — read it piece by piece:

- $f(t,d)$ — how many times term $t$ appears in $d$ (the TF).
- $|d|$ — length of $d$; avgdl — average document length in the corpus.
- $k_1$ (typically ~1.2–2.0) — controls **saturation**: how quickly extra occurrences stop helping.
- $b$ (typically ~0.75) — controls **length normalization**: 0 = ignore length, 1 = fully normalize.
- The IDF term is BM25's own variant, but the intuition (rare = valuable) is identical.

The whole fraction is just "TF, but with saturation and length-normalization built in." That's BM25. It has been the default first-stage ranker for decades because it's fast, robust, and needs no training.

---

## Hands-on

Notebook: `notebooks/02_keyword_search.ipynb`

In order, you will:

1. Load the corpus and tokenize it.
2. **Build an inverted index** and answer a query with a single lookup.
3. Implement **word-overlap** scoring and rank the *grass* query.
4. Implement **TF-IDF** from scratch and reproduce the worked example's numbers.
5. Implement **BM25** from scratch, then cross-check against the `rank-bm25` library.
6. Run the **failure demo**: the *"strong pain in the side of the head"* query that BM25 misses even though the migraine passage answers it — motivating embeddings in Chapter 3.

---

## Going deeper

- **Why the "+0.5" terms in BM25's IDF?** BM25 uses $\text{IDF}(t) = \ln\left(1 + \frac{N - \text{df}(t) + 0.5}{\text{df}(t) + 0.5}\right)$. The smoothing keeps IDF from blowing up or going negative for very common terms.
- **Tuning $k_1$ and $b$.** These are corpus-dependent. Short, uniform documents want low $b$; long, varied documents want $b$ near 0.75. Chapter 9 shows how to tune them by *measuring*.
- **BM25F and fielded search.** Real documents have fields (title, body, tags). BM25F weights matches in important fields (like the title) more heavily.
- **Still a strong baseline.** On many benchmarks, well-tuned BM25 is surprisingly competitive with — and complementary to — neural retrieval, which is exactly why hybrid search (Chapter 8) combines them.

---

## Pitfalls & gotchas

- **Mismatched tokenization.** You must tokenize the query the *same way* as the documents, or matches vanish. Same lowercasing, same stemming, same stop-word list.
- **Over-aggressive stop-word removal.** Dropping "not" or "no" can flip meaning; dropping "IT" can erase a real term. Be careful.
- **Raw counts without length normalization.** Long documents will dominate. Use BM25, not plain TF, in practice.
- **Expecting keyword search to understand synonyms.** It fundamentally can't — that's the point of the failure demo, and the reason the course continues.

---

## Key terms

lexical search, tokenization, stop words, stemming, inverted index, postings, term frequency (TF), document frequency (df), inverse document frequency (IDF), TF-IDF, BM25, term saturation, length normalization. *(See GLOSSARY.)*

---

## Check your understanding

1. Why is it called an *inverted* index, and what problem does it solve?
2. In the worked example, why does the word "is" contribute nothing to the score?
3. What two problems does BM25 fix compared to plain TF-IDF, and which parameter controls each?
4. Predict: for the query "blue", will D1 (sky) or D4 (whale) score higher, and what would change your answer? (Hint: think TF and length.)
5. Give a query where keyword search would fail on our corpus, and explain why.

---

## References


- Robertson & Zaragoza, *The Probabilistic Relevance Framework: BM25 and Beyond* (2009).
- Manning, Raghavan & Schütze, *Introduction to Information Retrieval*, chapters on term weighting and the vector space model.
- `rank-bm25` — the small Python library we cross-check against.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 1 (inspiration).

# Chapter 3 — From Text to Vectors

> **Level:** Beginner - **Status:** Complete **Prerequisites:** Chapter 2. **You'll build:** intuition and code for turning text into numbers, plus the similarity measures every semantic system depends on.

---

## At a glance

Computers compare numbers, not words. To search by *meaning*, we first have to turn text into **vectors** — lists of numbers — in a way where "close in meaning" becomes "close in space." This chapter walks the ladder from the crudest representations (one-hot, bag-of-words) up to the idea of a **dense embedding**, and teaches the tool that makes "close" precise: **cosine similarity.** Chapter 4 then shows how a real model produces those vectors.

---

## The motivation

In Chapter 2 keyword search failed on *"strong pain in the side of the head"* because it only sees words, not meaning. To fix that we need a representation where *temple* and *side of the head* end up near each other. That representation is a vector. So the question of this chapter is simple: **how do we convert text into numbers, and how do we measure whether two pieces of text are "close"?**

---

## Rung 1 — One-hot encoding (words as isolated symbols)

The simplest way to give a word a number is to assign it a slot. Build a vocabulary, then represent each word as a vector that is all zeros except a single 1 in its slot.

```
vocabulary: [apple, banana, car, castle, joy]

apple   ->  [1, 0, 0, 0, 0]
banana  ->  [0, 1, 0, 0, 0]
car     ->  [0, 0, 1, 0, 0]
```

It works, but look at what it *can't* do. Every pair of different words is **exactly equally far apart**. "apple" is no closer to "banana" than it is to "car." One-hot vectors encode *identity* but zero *meaning*. They're also **sparse** and huge — one dimension per word in the language (millions).

---

## Rung 2 — Bag-of-words (documents as word counts)

To represent a *document*, add up the one-hot vectors of its words — i.e., just count how often each vocabulary word appears. Order is thrown away (hence "bag").

```
vocab: [apple, fruit, is, an, red]

"an apple is a fruit"   ->  [1, 1, 1, 1, 0]
"apple is red"          ->  [1, 0, 1, 0, 1]
```

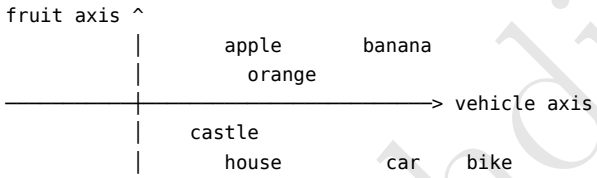
This is exactly what TF-IDF and BM25 operate on under the hood. It's great for keyword matching — but it inherits one-hot's blindness: two documents that say the same thing with *different words* share no dimensions, so they look unrelated. Synonyms are invisible. **Meaning still isn't captured.**

---

## Rung 3 — Dense vectors (meaning as location)

The breakthrough: instead of one dimension per word, use a *few hundred* dimensions, and let a model choose the numbers so that **similar meanings land near each other.** These are **dense** vectors (every dimension carries information, few zeros) — the **embeddings** we build in Chapter 4.

Picture a 2-D version of the idea (real ones have hundreds of dimensions):




Now "apple" sits next to "banana" and "orange" — close in space because close in meaning — and far from "car." *That's* what we couldn't do before. Given only the neighborhood, you could guess a new word's location from its meaning. (This is precisely the "where would you put *apple*?" intuition from the lesson.)

---

## Measuring "closeness"

Once text is vectors, we need a number for how similar two vectors are. Three common measures:

### Dot product

Multiply matching components and sum: $\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i$. Larger = more aligned, but it grows with vector *length*, so long vectors can dominate.

### Cosine similarity (the default for text)

The cosine of the angle between two vectors — it measures **direction, ignoring length**:

$$\text{cosine}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}$$

- **1.0** -> same direction (as similar as possible)
- **0.0** -> perpendicular (unrelated)

- **-1.0** -> opposite

We use cosine because in text the *direction* of a vector encodes meaning, while its magnitude often just reflects length or word count — which we usually don't want to matter.

### Euclidean distance

Straight-line distance $\|\mathbf{a} - \mathbf{b}\|$. Smaller = closer. For **normalized** vectors (scaled to length 1), Euclidean distance and cosine similarity rank neighbors identically — which is why many systems normalize first and then just use dot product.

### Worked mini-example

```
a = [1, 0]   ("apple", pointing along the fruit axis)
b = [2, 0]   ("banana", same direction, longer)
c = [0, 1]   ("car", pointing along the vehicle axis)

dot(a, b)    = 1·2 + 0·0 = 2
cosine(a, b) = 2 / (1 · 2) = 1.0      <-  identical meaning, despite different lengths
cosine(a, c) = 0 / (1 · 1) = 0.0      <-  unrelated
```

Notice cosine says apple~=banana are a perfect match (same direction) even though their raw vectors differ in length — exactly the behavior we want.

---

## Hands-on

Notebook: `notebooks/03_text_to_vectors.ipynb`

You will:

1. Build **one-hot** and **bag-of-words** vectors by hand and see why they miss meaning.
2. Implement **cosine similarity** and **Euclidean distance** in a few lines of numpy.
3. Play with a **toy 2-D word map** (fruits / vehicles / buildings / sports) and confirm that *apple* really does land among the fruits by nearest-neighbor.
4. See that normalizing vectors makes cosine and Euclidean agree on the ranking.

Everything here is pure numpy + matplotlib — no models yet, so it runs instantly.

---

## Going deeper

- **The curse of dimensionality.** In very high dimensions, distances between random points concentrate — most pairs look equally far. Good embeddings fight this by placing *meaningful* structure in the space; it's why learned embeddings beat random projections.
- **Why not just use bag-of-words with TF-IDF weights as "vectors"?** You can — that's the classic *vector space model*, and it's still sparse and meaning-blind to synonyms. Dense embeddings differ by being *learned* to capture semantics.
- **Distance metric choice matters.** Cosine is standard for text embeddings; some models are trained for dot product or Euclidean. Always match the metric the model was trained with (Chapter 4).

---

## Pitfalls & gotchas

- **Comparing un-normalized vectors with dot product** can let a long document beat a more relevant short one. Normalize, or use cosine.
- **Mixing metrics.** If a model was trained with cosine, don't rank with raw Euclidean on un-normalized vectors and expect the same neighbors.
- **Thinking dimensions are human-interpretable.** In real embeddings, no single dimension means "fruitiness." Meaning is distributed across all of them.

---

## Key terms

vector, one-hot encoding, sparse vector, bag-of-words, dense vector, embedding, dot product, cosine similarity, Euclidean distance, normalization, vector space. *(See GLOSSARY.)*

---

## Check your understanding

1. Why are all distinct one-hot vectors equally far apart, and why is that a problem?
2. What information does bag-of-words throw away, and what does it keep?
3. In one sentence, what does cosine similarity measure that the dot product doesn't isolate?
4. Two vectors point the same way but one is twice as long. What is their cosine similarity?
5. When would Euclidean distance and cosine similarity give the *same* ranking of neighbors?

---

## References


- Jurafsky & Martin, *Speech and Language Processing*, chapter on vector semantics (free draft).
- Manning, Raghavan & Schütze, *Introduction to Information Retrieval*, the vector space model.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 2 (inspiration).

# Chapter 4 — Embeddings Deep Dive

> **Level:** Beginner -> Intermediate - **Status:** Complete **Prerequisites:** Chapter 3 (vectors & cosine similarity). **You'll build:** hands-on intuition for real embedding models, and the exact demo that makes semantic search "click."

---

## At a glance

Chapter 3 argued that if we could place text in a space where *meaning = location*, search would work by meaning. This chapter is where that becomes real. An **embedding model** is a neural network trained to do exactly that: feed it text, get back a dense vector whose neighbors are semantically similar. We'll use an open-source model, embed our own sentences, and reproduce the striking result from the lesson — **every question's nearest neighbor is its own answer.** That single observation *is* the seed of dense retrieval (Chapter 5).

---

## The motivation

We keep hitting the same wall: *temple* should be near *side of the head*, but keyword and bag-of-words methods can't see it. An embedding model learns this from massive amounts of text. By the end of this chapter, "how do I stop my laptop dying" will sit near "improving battery life" in vector space — no shared words required.

---

## What is an embedding model, really?

An **embedding model** maps a piece of text to a fixed-length dense vector (its **embedding**). It's trained so that texts humans consider similar get vectors that are close (high cosine similarity), and dissimilar texts get vectors that are far apart.

```
"an apple is a fruit"  ──> [ 0.02, -0.11, 0.34, ... ]   (e.g. 384 numbers)
"a banana is a fruit"  ──> [ 0.03, -0.09, 0.31, ... ]    <-  close to the apple vector
"a car has four wheels"──> [-0.21,  0.40, 0.05, ... ]    <-  far from both
```

You don't design the numbers; the model learned them. No single dimension means "fruitiness" — meaning is spread across all of them.

---

## Three granularities: word, sentence, document

Embeddings exist at different scales, and the course cares mostly about the middle one:

| Level | What it represents | Typical use |
|---|---|---|
| **Word** | one word's meaning | word analogies, classic NLP |
| **Sentence / passage** | the meaning of a whole sentence or paragraph | **semantic search, RAG** <- our focus |
| **Document** | a long text's overall meaning | clustering, topic maps |

We use **sentence/passage embeddings** because search matches queries against passages.

---

## A short history (so the names make sense)

- **word2vec / GloVe (2013–2014)** — learn *one fixed vector per word*. Famous for vector arithmetic: `king - man + woman ~= queen`. Limitation: "bank" (river) and "bank" (money) get the *same* vector, since there's no context.
- **Contextual embeddings — BERT (2018)** — the vector for a word now *depends on the sentence* around it, so the two "banks" differ. But BERT out of the box isn't tuned for comparing whole sentences by cosine similarity.
- **Sentence embeddings — Sentence-BERT / `sentence-transformers` (2019+)** — fine-tuned so a single vector per sentence is directly comparable with cosine similarity. This is what we use, and what powers modern semantic search.

---

## The model we use

We use **`all-MiniLM-L6-v2`** from the open-source `sentence-transformers` library:

- Free, small (~90 MB), runs on CPU, no API key.
- Outputs **384-dimensional** vectors.
- Fast enough to embed thousands of passages in seconds.

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")
vecs = model.encode(["an apple is a fruit", "a banana is a fruit"])
# vecs.shape -> (2, 384)
```

> Different models output different dimensionalities (384, 768, 1536, …). Bigger isn't always better — it's a trade-off of quality vs. speed vs. memory, which we revisit in Chapters 6 & 9.

---

## The "aha" demo: questions find their answers

This is the heart of the lesson. Take question/answer pairs, embed *all* of them, and for each **question** find its nearest neighbor among the *other* sentences. The nearest one is its own **answer** — even though question and answer share few words.

```
Q: "what color is the sky?"      ->  nearest: "the sky is blue."
Q: "what is an apple?"           ->  nearest: "an apple is a fruit."
Q: "where does the bear live?"   ->  nearest: "the bear lives in the woods."
```

Sit with this: matching *meaning* just turned a pile of sentences into a working question- answering lookup. **Searching for the nearest embedding to a query = searching by meaning.** That's dense retrieval, and Chapter 5 scales it to a real corpus.

---

## Properties worth knowing

- **Dimensionality** — fixed per model (384 for ours). Every text, short or long, becomes the same size vector.
- **Normalization** — many workflows L2-normalize embeddings so cosine similarity = dot product (faster, and what most vector databases expect).
- **Semantic arithmetic** — with *word* vectors, `king - man + woman ~= queen`; a fun way to see that directions in the space carry meaning. (Sentence vectors don't do arithmetic as cleanly, but neighborhoods are very meaningful.)

---

## Hands-on

Notebook: `notebooks/04_embeddings.ipynb`

You will:

1. Load `all-MiniLM-L6-v2` and embed a few sentences; inspect the shape (384) and a slice of the numbers.
2. Confirm that paraphrases ("hello, how are you?" vs "hi, how's it going?") get high cosine similarity while unrelated sentences don't.
3. Run the **question -> answer nearest-neighbor demo** on our own Q/A pairs.
4. Embed our course corpus, find nearest neighbors for a query, and confirm the *migraine* passage that keyword search missed (Chapter 2) is now the top hit.
5. **Cluster** the corpus by meaning and **project to 2-D** (PCA) to see topics group together.

> Note: this notebook downloads a small model on first run, so it needs internet the first time (it then works offline). The pure-analysis helpers (cosine matrix, nearest neighbors, clustering, PCA) are written so they're easy to test even without the model.

---

## Going deeper

- **How are these models trained?** Mostly *contrastive learning*: pull known-similar pairs (e.g., a question and its answer, or a sentence and its paraphrase) together and push random pairs apart. The Q -> A demo works because the model was trained on exactly this kind of signal.
- **Bi-encoder vs. cross-encoder.** The model here is a **bi-encoder**: it embeds each text independently, so vectors can be precomputed and reused — essential for fast search (Chapter 5). Cross-encoders (Chapter 7) are more accurate but can't precompute.

- **Symmetric vs. asymmetric search.** Some models are tuned for querydocument (asymmetric, e.g. "msmarco" models) vs. sentencesentence (symmetric). Picking the right one matters.
- **Pooling.** A sentence embedding is usually a pooled summary (mean of token vectors) from a transformer; different pooling changes the vectors.

---

## Pitfalls & gotchas

- **Different model = different space.** You cannot compare vectors from two different models. Embed the corpus and the query with the *same* model.
- **Forgetting to re-embed after a model change.** Swap the model -> you must rebuild the whole index.
- **Ignoring the model's max input length.** Long passages get truncated; that's a big reason we *chunk* documents (Chapter 11).
- **Assuming bigger dimensions are always better.** They cost memory and speed; measure (Chapter 9) before paying for them.

---

## Key terms

embedding, embedding model, dense vector, dimensionality, word2vec, GloVe, contextual embeddings, sentence embeddings, bi-encoder, normalization, contrastive learning, pooling. *(See GLOSSARY.)*

---

## Check your understanding

1. In one sentence, what is an embedding model trained to do?
2. Why does the word "bank" get one vector in word2vec but different vectors in BERT?
3. Why can a bi-encoder's document vectors be precomputed, and why does that matter for search?
4. Explain *why* a question's nearest embedding tends to be its own answer.
5. You switch from a 384-dim model to a 768-dim one. What must you redo, and why?

---

## References


- Mikolov et al., *Efficient Estimation of Word Representations in Vector Space* (word2vec, 2013).
- Devlin et al., *BERT* (2018).
- Reimers & Gurevych, *Sentence-BERT* (2019) — the basis for `sentence-transformers`.
- `sentence-transformers` documentation (open-source library used here).
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 2 (inspiration).

# Chapter 5 — Dense Retrieval & Semantic Search

> **Level:** Beginner -> Intermediate - **Status:** Complete **Prerequisites:** Chapter 4 (embeddings & cosine similarity). **You'll build:** a complete semantic search engine you query in natural language, and a head-to-head against BM25.

---

## At a glance

This is the payoff of Part III. We take the embeddings from Chapter 4 and turn them into an actual **search engine**: embed every document once, embed the query, return the documents whose vectors are nearest. That's **dense retrieval** — retrieval by *meaning*. You'll build it end to end over our own corpus, then race it against the BM25 engine from Chapter 2 to see exactly where each wins.

---

## The motivation

We've been circling this since Chapter 1: the *"strong pain in the side of the head"* query whose answer (the migraine passage) says *"sharp sensation localized to one temple."* Keyword search can't bridge that gap. In Chapter 4 we saw embeddings put those two phrases near each other in vector space. Now we make that the basis of retrieval — so a user can ask in their own words and still find the right passage.

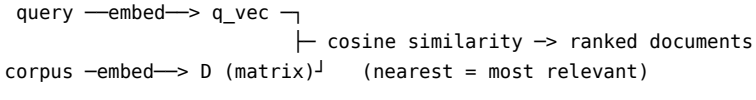
---

## The core algorithm (three steps)

Dense retrieval is beautifully simple:

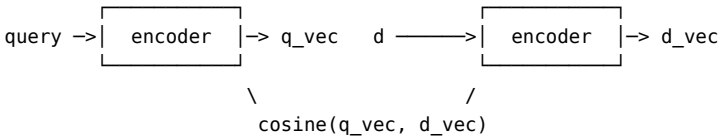
```
INDEXING (once, offline)                SEARCH (per query)
------------------------                ------------------------
for each document d:                    1. embed the query -> q_vec
    d_vec = embed(d)                     2. score every d: cosine(q_vec, d_vec)
store all d_vec in a matrix              3. return the top-k highest-scoring docs
```

That's it. The "intelligence" lives entirely in the embedding model; retrieval itself is just "find the nearest vectors." Contrast this with BM25, where the intelligence was in the scoring formula and the model was just word counts.



## The bi-encoder: why this scales

The model we use is a **bi-encoder** (two-tower): it embeds the query and each document *independently*.




Why does "independently" matter so much? Because document vectors don't depend on the query, you can **precompute them all once** and reuse them for every future query. At search time you only embed the (short) query and do fast vector math. This is what makes dense retrieval practical over millions of documents — and it's the crucial difference from the **cross-encoder** re-ranker in Chapter 7, which is more accurate but must re-run the model for every query–document pair.

## Building it over our corpus

In the notebook we implement a small, reusable `SemanticSearch` class:

1. **Index:** embed all 16 passages -> a `(16, 384)` matrix, L2-normalized so cosine = dot product.
2. **Search:** embed the query, take the dot product with the matrix, `argsort` for the top-k.
3. **Query in plain English:** `search("why is grass green?")`, `search("a large ocean animal")`, `search("strong pain in the side of the head")`.

The migraine query — our running example — now returns the migraine passage as the **top** result, finally solving the problem we posed in Chapter 1.

## Dense vs. BM25: a fair race

Neither method is strictly better; they fail differently. We run both on the same queries and compare:

| Query type | BM25 (keyword) | Dense (semantic) |
|---|---|---|
| Paraphrase / synonyms ("pain in the head" -> "temple") | [no] misses | [yes] finds |
| Conceptual ("a large ocean animal" -> "blue whale") | [no] often misses | [yes] finds |
| Exact rare term / code / name ("all-MiniLM-L6-v2") | [yes] nails it | [!] can drift |

| Query type | BM25 (keyword) | Dense (semantic) |
|---|---|---|
| Typo-free keyword overlap | [yes] strong | [yes] usually fine |

The takeaway that sets up Chapter 8: **dense retrieval wins on meaning; keyword retrieval wins on exactness.** Real systems often want both — which is why hybrid search exists.

---

## Hands-on

Notebook: `notebooks/05_dense_retrieval.ipynb`

You will:

1. Build a reusable `SemanticSearch` class (index + search).
2. Query the corpus in natural language and inspect scores.
3. Confirm the migraine query is finally solved.
4. Run **dense vs. BM25** side by side on several queries and interpret the differences.
5. Probe a failure case (an exact rare token) that motivates hybrid search.

> First run downloads the small embedding model (needs internet once). The retrieval logic (normalize, dot product, top-k) is written to be testable even without the model.

---

## Going deeper

- **Exact vs. approximate search.** Our corpus is tiny, so we compute cosine against *every* document (brute force, exact). At scale that's too slow — Chapter 6 introduces approximate nearest-neighbor indexes.
- **Asymmetric retrieval.** For question -> passage search, models fine-tuned on query/passage pairs (e.g. MS MARCO models) often beat symmetric sentence models. Match the model to the task.
- **Score calibration.** Cosine scores aren't probabilities and aren't comparable across models or even across queries. Use them to *rank*, not as absolute confidence.
- **Dense vs. sparse-neural (SPLADE).** There's a middle ground: learned *sparse* representations that keep an inverted index but add semantics. Worth knowing the landscape exists.

---

## Pitfalls & gotchas

- **Embedding query and corpus with different models** (or different pooling/normalization) -> garbage results. Keep them identical.
- **Forgetting to normalize** when you use dot product as the similarity. Either normalize and use dot product, or use full cosine.
- **Assuming semantic always beats keyword.** It doesn't for exact identifiers, product codes, or rare names — the exact case keyword search was built for.
- **Stale index.** Add or edit documents? You must embed and add them to the matrix, or they're invisible to search.

## Key terms

dense retrieval, semantic search, bi-encoder, two-tower, embedding index, nearest neighbor search, brute-force search, top-k, asymmetric search. *(See GLOSSARY.)*

## Check your understanding

1. State the three steps of dense retrieval in your own words.
2. Why can a bi-encoder precompute document vectors, and why is that essential for scale?
3. Give one query where BM25 beats dense retrieval, and explain why.
4. Why do we L2-normalize the embeddings before taking dot products?
5. Our search is "brute force." What does that mean, and when does it stop being good enough?

## References


- Karpukhin et al., *Dense Passage Retrieval for Open-Domain Question Answering* (DPR, 2020).
- Reimers & Gurevych, *Sentence-BERT* (2019).
- `sentence-transformers` documentation (semantic search utilities).
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 3 (inspiration).

# Chapter 6 — Vector Databases & Approximate Nearest Neighbor (ANN)

**Level:** Intermediate - **Status:** Complete **Prerequisites:** Chapter 5 (dense retrieval). **You'll build:** a fast, persistent vector index and a felt understanding of the speed-vs-accuracy trade-off behind every vector database.

---

## At a glance

In Chapter 5 we scored the query against *every* document (brute force). That's exact and fine for 16 passages — but hopeless for 16 million. This chapter is about making nearest-neighbor search *fast at scale*: the **approximate nearest neighbor (ANN)** idea, the index types that implement it (**HNSW**, IVF, PQ), and the **vector databases** (FAISS, Chroma) that package it up with persistence and metadata filtering.

---

## The motivation

Semantic search has a scaling problem hiding in plain sight. Brute-force search compares the query to all $N$ documents — that's $O(N)$ per query. At a million vectors of 384 dimensions, every single search touches hundreds of millions of numbers. Users expect answers in **milliseconds**. Something has to give. The insight: we don't actually need the *exact* nearest neighbors — we need the *right documents*, fast. Trading a tiny bit of accuracy for a huge speed-up is the whole game.

---

## Brute force, and why it breaks

```
brute force:  for each of N documents, compute similarity to the query, keep the top-k
              cost ∝ N x dimensions       <-  grows linearly with corpus size
```

Double the corpus, double the work — *every query, forever*. Exact, but it doesn't scale. ANN changes the shape of that cost.

---

## The ANN idea: don't look at everything

Approximate nearest neighbor algorithms **pre-organize** the vectors so a query only has to examine a small, promising subset instead of the whole corpus. You give up the guarantee of finding the *exact* closest vectors; in return you get sublinear search that's often 10–1000x faster while still returning the true top results the vast majority of the time.

We measure that "vast majority" with **recall@k**: of the true top-k neighbors (from brute force), how many did the approximate index actually return? Recall of 0.95 means you're getting 95% of the exact answers at a fraction of the cost — usually a great deal.

### Two families to know

**HNSW (Hierarchical Navigable Small World)** — a *graph*. Each vector is a node linked to its near neighbors, with a hierarchy of layers (like express lanes on a highway). A search greedily hops from node to node, zooming in on the query's neighborhood. Fast, high recall, memory-hungry, and the default in most vector databases.

```
HNSW: start at the top layer (few, long links), descend layer by layer,
      at each layer walk greedily toward the query, refine at the bottom.

layer 2:   * ──────────────── *          (coarse, long hops)
layer 1:   * ─── * ─── * ─── *          (medium)
layer 0:   *-*-*-*-*-*-*-*-*-*-*-*       (dense, short hops — the full graph)
```

**IVF (Inverted File)** — *clustering*. Partition the space into buckets (via k-means); at query time, only search the few buckets nearest the query instead of all of them. Simple and effective.

**Product Quantization (PQ)** — *compression*. Squash each vector into a compact code so millions fit in RAM and distance approximations are cheap. Often combined with IVF (`IVF+PQ`).

You don't need to implement these — you need to know they exist and what knob each offers (recall vs. speed vs. memory).

---

## Vector databases: the index plus everything else

An ANN index alone isn't a product. A **vector database** wraps it with the things real apps need:

- **Persistence** — save the index to disk and reload it; don't re-embed on every restart.
- **Metadata & filtering** — store fields alongside vectors ("language = en", "year >= 2020") and filter during search.
- **CRUD** — add, update, delete vectors as your corpus changes.
- **Scaling & serving** — batching, sharding, an API.

We use two open-source options:

- **FAISS** — a fast in-memory index library (from Meta). Great for learning and for large static indexes. We use `IndexFlatIP` (exact) and `IndexHNSWFlat` (approximate).
- **Chroma** — a lightweight vector *database* with persistence and metadata filtering, a gentle on-ramp to production patterns.

---

## Hands-on

Notebook: `notebooks/06_vector_databases.ipynb`

You will:

1. Build an **exact** FAISS index (`IndexFlatIP`) and confirm it matches Chapter 5's brute force.
2. Build an **HNSW** index over a larger set of synthetic vectors.
3. **Benchmark**: measure query time and **recall@10** of HNSW vs. exact search — and *feel* the trade-off (big speed-up, tiny recall loss).
4. Use **Chroma** to persist embeddings with metadata and run a filtered semantic query ("only English passages").
5. Reflect on choosing an index for your corpus size.

> The FAISS benchmark runs fully offline with synthetic vectors (no model needed). The Chroma section uses the embedding model, so it runs on your machine after the first model download.

---

## Going deeper

- **The three-way trade-off.** Every ANN index balances **recall**, **speed (QPS)**, and **memory**. You can't max all three; you tune for your constraints. HNSW's `M` and `efSearch`, IVF's `nprobe` are the knobs.
- **Build time vs. query time.** HNSW is slower to *build* and heavier in RAM but very fast to *query*. IVF is cheaper to build. For frequently-rebuilt indexes this matters.
- **Filtering is hard.** Combining metadata filters with ANN ("nearest vectors *where* year=2024") can wreck recall if done naively; mature vector DBs implement careful pre/post-filtering.
- **Disk-based & billion-scale.** Techniques like DiskANN push ANN beyond RAM. The principles are the same; the engineering is deeper.
- **Benchmarks.** ann-benchmarks.com compares libraries on the recall/speed frontier — useful when choosing.

---

## Pitfalls & gotchas

- **Using the wrong metric.** FAISS `IndexFlatIP` maximizes inner product — normalize your vectors so it equals cosine, or use `IndexFlatL2` intentionally.
- **Judging quality by speed alone.** A blazing index with 0.6 recall is silently dropping relevant results. Always measure recall against exact search.
- **Rebuilding vs. updating.** Some indexes don't support deletes well; plan how your corpus changes.
- **Over-engineering small corpora.** Under ~105 vectors, exact search is often fast enough — don't reach for ANN until you need it.

---

## Key terms

approximate nearest neighbor (ANN), recall@k, brute-force search, HNSW, IVF, product quantization (PQ), vector database, FAISS, Chroma, metadata filtering, quantization. *(See GLOS-*

*SARY.)*

---

## Check your understanding

1. Why does brute-force search stop being viable as the corpus grows?
2. What exactly does ANN approximate, and how do we measure the cost of that approximation?
3. Give a one-sentence intuition for how HNSW avoids scanning every vector.
4. What does a vector *database* add on top of a raw ANN *index*?
5. You have 5,000 vectors. Should you reach for HNSW? Why or why not?

---

## References


- Malkov & Yashunin, *Efficient and robust approximate nearest neighbor search using HNSW graphs* (2016).
- Johnson, Douze & Jégou, *Billion-scale similarity search with GPUs* (FAISS, 2017).
- FAISS and Chroma documentation (open-source tools used here).
- ann-benchmarks.com — recall/speed comparisons across libraries.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 3 (Dense Retrieval; ANN/vector-DB portion) (inspiration).

# Chapter 7 — Re-ranking

> **Level:** Intermediate - **Status:** Complete **Prerequisites:** Chapter 5 (dense retrieval). Chapter 6 helpful. **You'll build:** a two-stage retrieve-then-rerank pipeline and measure the quality jump.

---

## At a glance

First-stage retrieval (BM25 or dense) is built for *speed*: it scans the whole corpus, so it has to be cheap. That speed comes at a cost in precision — the top result isn't always the *best* result. **Re-ranking** adds a second, slower, much more accurate model that re-scores only the handful of candidates the first stage returned. This "retrieve wide and cheap, then re-rank narrow and precise" pattern is how nearly every production search and RAG system gets both speed and quality.

---

## The motivation

Recall the bi-encoder from Chapter 5: it embeds the query and each document *separately*, then compares vectors. That separation is what makes it fast (precompute all document vectors) — but it's also a limitation. The model never gets to read the query and a document *together*, so it can miss subtle relevance signals. The result: the right document is usually somewhere in the top 20–100, but not always at rank 1. Re-ranking fixes the ordering of that small set.

```
Stage 1 (retrieval)          Stage 2 (re-ranking)
top-50 candidates,           read query+doc TOGETHER,
fast but approximate    ->   re-score each of the 50   ->   precise top-5
(bi-encoder / BM25)          (cross-encoder)
```

---

## Bi-encoder vs. cross-encoder — the key distinction

This is the heart of the chapter.

```
BI-ENCODER (Chapter 5, first stage)
-----------------------------------
query ->[encoder]-> q_vec
doc   ->[encoder]-> d_vec
score = cosine(q_vec, d_vec)

+ document vectors precomputed once
+ scoring is just vector math (fast)
- never sees query & doc together
```

```
CROSS-ENCODER (this chapter, re-ranker)
---------------------------------------
query ┐
      ├-> [ single model reads BOTH ] -> score
doc   ┘        (a relevance number)

+ reads query+doc jointly -> far more accurate
- must run the model for EVERY query-doc pair (slow)
- cannot precompute -> impossible over a full corpus
```

- A **bi-encoder** is like matching people by their dating-profile vectors — fast, but each profile was written without knowing who's asking.
- A **cross-encoder** is like a recruiter reading the job description and a résumé *side by side* — much better judgment, but you can't do it for a million résumés per query.

So we use each where it shines: bi-encoder to *retrieve* many candidates cheaply, cross-encoder to *re-rank* the few precisely.

---

## Why not just use the cross-encoder for everything?

Because it can't precompute. To find the best of $N$ documents, a cross-encoder must run the model $N$ times **per query** — reading each (query, document) pair fresh. At a million documents that's a million forward passes for a single search: hopelessly slow. The bi-encoder narrows the field to ~50 first; the cross-encoder then does its expensive, accurate work on just those 50. Best of both worlds.

---

## The model we use

An open-source cross-encoder trained on MS MARCO (a large query -> passage relevance dataset):

```
from sentence_transformers import CrossEncoder
reranker = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
scores = reranker.predict([(query, doc1), (query, doc2), ...])   # higher = more relevant
```

It outputs a relevance score for each (query, document) pair. We sort candidates by that score.

---

## Hands-on

Notebook: `notebooks/07_reranking.ipynb`

You will:

1. Retrieve top-k candidates with the Chapter 5 dense retriever (first stage).
2. Load a **cross-encoder** and re-score those candidates (second stage).
3. Watch the ordering improve — a relevant passage that sat at rank 4 jumps to rank 1.
4. Wrap it as a reusable `retrieve_then_rerank(query)` function.
5. Discuss the cost: how big should the first-stage candidate set be? (The recall/latency knob.)

> The cross-encoder downloads on first run (internet once). The pipeline logic (take top-k, score pairs, re-sort) is written to be testable with a mock scorer.

---

## Going deeper

- **Choosing the candidate depth (top-k).** Re-rank too few and you might exclude the best document (first-stage recall caps your ceiling); re-rank too many and latency grows. Typical values: retrieve 50–200, re-rank to 5–10. Tune by measuring (Chapter 9).

- **Re-ranking is bounded by retrieval recall.** A re-ranker can only reorder what it's given. If the right document never made the top-50, no re-ranker can save you — improve first-stage recall first.
- **Extra signals.** Production re-rankers often blend the semantic score with popularity, freshness, click data, or business rules — re-ranking is the natural place to fold these in.
- **LLM re-rankers.** You can prompt an LLM to rank passages (listwise). Powerful but slower and costlier than a dedicated cross-encoder; useful when quality matters more than latency.
- **Latency budgeting.** The cross-encoder is the expensive step; batch the pairs and cap k to stay within your latency target.

---

## Pitfalls & gotchas

- **Re-ranking the whole corpus.** Never. Only re-rank the small first-stage candidate set.
- **Mismatched task.** Use a cross-encoder trained for query -> passage relevance (e.g. MS MARCO), not a sentence-similarity model, for search re-ranking.
- **Ignoring first-stage recall.** Spending all your effort on the re-ranker while the retriever misses relevant docs is optimizing the wrong stage.
- **Comparing cross-encoder scores across queries.** Like cosine scores, they rank within a query; they're not calibrated probabilities across queries.

---

## Key terms

re-ranking, cross-encoder, bi-encoder, two-stage retrieval, candidate set, first-stage recall, MS MARCO, listwise ranking. *(See GLOSSARY.)*

---

## Check your understanding

1. In one sentence, what can a cross-encoder do that a bi-encoder can't — and what does it cost?
2. Why can't we just use the cross-encoder as the first-stage retriever?
3. What limits how much a re-ranker can improve results, no matter how good it is?
4. You retrieve top-10 and re-rank, but quality is still poor. Name two different fixes and when each applies.
5. Where in a two-stage pipeline would you add a "prefer recent documents" rule, and why there?

---

## References


- Nogueira & Cho, *Passage Re-ranking with BERT* (2019) — the cross-encoder re-ranking idea.
- Reimers & Gurevych, *Sentence-BERT* (2019) — bi- vs. cross-encoder trade-offs.
- `sentence-transformers` CrossEncoder documentation and MS MARCO models.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson 4 (inspiration).

# Chapter 8 — Hybrid Search

> **Level:** Intermediate - **Status:** Complete **Prerequisites:** Chapters 2 (BM25) and 5 (dense retrieval). **You'll build:** a hybrid retriever that fuses keyword and semantic search, and see it beat either one alone.

---

## At a glance

We've now built two retrievers that fail in *different* ways. BM25 nails exact words but is blind to meaning; dense retrieval matches meaning but can miss exact terms, names, and codes. The obvious move: **use both and combine them.** That's **hybrid search**, and because the two methods are complementary, the fused result is often better than either alone. The trick is *how* to combine two different kinds of scores — and the cleanest answer is **Reciprocal Rank Fusion (RRF)**.

---

## The motivation

A quick table you've seen building across the course:

| Query | BM25 (keyword) | Dense (semantic) |
|---|---|---|
| "strong pain in the side of the head" | [no] misses (no shared words) | [yes] finds migraine passage |
| "a large ocean animal" | [no] misses | [yes] finds blue whale |
| "Lucene" (exact library name) | [yes] exact hit | [!] may drift |
| "SKU-4471-B" (a product code) | [yes] exact hit | [no] meaningless to the model |

Notice the [no] 's and [yes] 's are in *different rows*. Where one fails, the other tends to succeed. Hybrid search is about capturing both columns of [yes] at once.

---

## The hard part: two scores that don't mix

You can't just add a BM25 score and a cosine score. They live on different scales:

- **BM25** scores are unbounded positives (e.g., 0 to ~15), depending on term rarity and length.
- **Cosine** scores sit in roughly [-1, 1].

Adding them lets BM25's larger numbers dominate arbitrarily. Two ways to fix this:

### Option A — Score normalization + weighted sum

Rescale each score to [0, 1] (e.g., min-max), then combine with a weight:

```
hybrid = alpha - normalize(dense_score) + (1 - alpha) - normalize(bm25_score)
```

Simple, but fragile: min-max normalization is sensitive to outliers, and the best alpha varies by query and corpus.

### Option B — Reciprocal Rank Fusion (RRF) <- preferred

Ignore the raw scores entirely; use only each document's **rank** in each list. For a document $d$:

$$\mathrm{RRF}(d) = \sum_{r \in \text{rankers}} \frac{1}{k + \mathrm{rank}_r(d)}$$

where `rank` is 1-based and $k$ is a small constant (commonly 60). Sum a document's reciprocal ranks across all retrievers; sort by the total.

Why RRF is the default:

- **Scale-free.** It never compares a BM25 number to a cosine number — only positions.
- **Robust.** Outlier scores can't dominate; a document ranked highly by *both* retrievers wins.
- **Trivially simple.** A few lines of code, no tuning of alpha, no normalization headaches.

### RRF worked example

Query returns these rankings (k = 60):

```
BM25 : [D10, D4,  D2]        Dense: [D2,  D4,  D7]
        #1   #2   #3                 #1   #2   #3

RRF scores (k = 60):
D2  : 1/(60+3) [BM25 #3] + 1/(60+1) [Dense #1] = 0.01587 + 0.01639 = 0.03227    <-  highest
D4  : 1/(60+2) [BM25 #2] + 1/(60+2) [Dense #2] = 0.01613 + 0.01613 = 0.03226    <-  a hair behind
D10 : 1/(60+1) [BM25 #1] + 0        (not in Dense)                 = 0.01639
D7  : 0        (not in BM25) + 1/(60+3) [Dense #3]                 = 0.01587
```

D2 and D4 rise to the top because *both* retrievers liked them — exactly the consensus behavior we want, and neither needed to be #1 in a single list to win. The documents only one retriever found (D10, D7) rank well below them. Note D2 edges out D4 despite D4 being more evenly placed: one #1 finish (1/61) outweighs two #2 finishes here. RRF rewards strong positions and, especially, agreement across retrievers.

---

## Hands-on

Notebook: `notebooks/08_hybrid_search.ipynb`

You will:

1. Reuse BM25 (Chapter 2) and the dense retriever (Chapter 5) to produce two ranked lists.
2. Implement **Reciprocal Rank Fusion** in a few lines and fuse the lists.
3. Compare BM25-only vs. dense-only vs. hybrid on queries designed to expose each method's weakness (a synonym query and an exact-token query).
4. See hybrid stay strong on *both*, where each single method fails on one.
5. (Optional) Add the Chapter 7 re-ranker on top of the fused candidates — the full modern stack.

> The RRF logic is pure Python and runs anywhere. The dense list needs the embedding model (runs on your machine); the fusion is verified independently with fixed rankings.

---

## Going deeper

- **Tuning k in RRF.** Larger $k$ flattens the contribution of top ranks (more democratic across the list); smaller $k$ rewards being #1 heavily. 60 is a robust default from the original paper.
- **Weighted RRF.** You can weight retrievers: $\Sigma\ w_r\ /(k + \text{rank}_r)$. Useful when you trust one retriever more, but adds a knob to tune.
- **Hybrid + re-rank is the standard modern stack:** BM25 U dense -> RRF fuse -> cross-encoder re-rank -> (LLM for RAG). Chapters 2, 5, 7, 8, and 10 assembled.
- **Sparse-neural hybrids.** Learned sparse models (SPLADE) blur the keyword/semantic line by putting semantics *into* an inverted index; another route to "best of both."
- **Fusion vs. single strong model.** Sometimes one well-tuned retriever suffices. Measure (Chapter 9) before adding fusion complexity.

---

## Pitfalls & gotchas

- **Summing raw scores from different scales.** The classic mistake; use RRF or normalize carefully.
- **Fusing lists of different lengths inconsistently.** Decide how to treat documents missing from one list (they simply contribute 0 from that retriever in RRF).
- **Assuming hybrid always wins.** For a corpus of pure natural-language prose with no codes or names, dense alone may already be enough — hybrid adds most when queries mix intent and exact terms.
- **Double-counting near-duplicate retrievers.** Fusing two very similar dense models adds little; diversity (lexical + semantic) is what makes fusion pay off.

---

## Key terms

hybrid search, score normalization, Reciprocal Rank Fusion (RRF), rank fusion, weighted fusion, complementary retrievers, SPLADE. *(See GLOSSARY.)*

---

## Check your understanding

1. Why can't you simply add a BM25 score and a cosine score together?
2. Explain, in one sentence, what RRF uses instead of raw scores and why that helps.
3. In the worked example, why do D2 and D4 outrank D10 and D7?
4. Give a query where hybrid clearly beats both BM25-only and dense-only, and say why.
5. Where does re-ranking (Chapter 7) fit relative to fusion in the full stack?

---

## References


- Cormack, Clarke & Büttcher, *Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods* (2009) — the RRF paper.
- Robertson & Zaragoza, *The Probabilistic Relevance Framework: BM25 and Beyond* (2009).
- Formal et al., *SPLADE* (2021) — learned sparse retrieval.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search* (inspiration).

# Chapter 9 — Evaluating Search

> **Level:** Intermediate - **Status:** Complete **Prerequisites:** Chapters 2, 5, 7, 8 (the retrievers we'll score). **You'll build:** an evaluation harness and a leaderboard comparing every retriever you've built.

---

## At a glance

We've built keyword, dense, hybrid, and re-ranked retrievers, and *claimed* each is better than the last. This chapter makes those claims **measurable**. You'll create a small labeled test set and implement the standard IR metrics — **precision@k, recall@k, MRR, nDCG, MAP** — from scratch, then use them to turn "this feels better" into a number.

> The golden rule of this chapter: **you cannot improve what you do not measure.** Every tuning decision in earlier chapters (k1 and b in BM25, efSearch in HNSW, alpha in hybrid, candidate depth in re-ranking) should be made by measuring, not guessing.

---

## The motivation

Imagine two search systems. On one query System A looks better; on another System B does. Which should you ship? Eyeballing a few queries doesn't scale and isn't reproducible. We need (1) a set of queries with known correct answers, and (2) numbers that summarize ranking quality across all of them. That's an **evaluation set** plus **metrics**.

---

## Step 1 — Relevance judgments (the ground truth)

Evaluation starts with a labeled set: queries paired with the documents a human judged **relevant**. We ship a tiny one in `data/eval_queries.json` — six queries over our 16-passage corpus, each with its relevant document ids.

```
{"query": "a large animal that lives in the ocean", "relevant_ids": [4, 5]}
```

We use **binary** relevance (relevant or not) for simplicity. Real sets are bigger and often use **graded** relevance (perfect / good / marginal), which nDCG can exploit. Building good judgments is the hard, human part of evaluation — the metrics are the easy part.

---

## Step 2 — The metrics, by intuition

Each metric answers a different question. Implemented from scratch in `src/metrics.py`.

### Precision@k — "of what I showed, how much was good?"

Fraction of the top-k results that are relevant. `precision@5 = 0.6` -> 3 of the top 5 are relevant. Cares about the top; ignores relevant docs you missed.

### Recall@k — "of what's good, how much did I find?"

Fraction of all relevant documents that appear in the top-k. `recall@5 = 0.5` -> you surfaced half of the relevant docs. Precision and recall trade off against each other.

### MRR (Mean Reciprocal Rank) — "how high was the *first* right answer?"

For each query take `1 / (rank of the first relevant result)`, then average over queries. First relevant at rank 1 -> 1.0; rank 2 -> 0.5; rank 5 -> 0.2. Perfect when you only care about the single best answer (e.g., "I feel lucky" search, or the top passage for RAG).

### nDCG (Normalized Discounted Cumulative Gain) — "are the best results near the top?"

Rewards putting relevant documents *high*, with a logarithmic discount for lower positions, then normalizes by the ideal ordering so the score sits in [0, 1]. The go-to metric when *order* matters across the whole list.

```
DCG   = Σ  gain(rank) / log2(rank + 1)          (a hit at rank 1 is worth more than at rank 5)
nDCG  = DCG / DCG(ideal ranking)                 (1.0 = perfect order)
```

### MAP (Mean Average Precision) — "overall precision across all relevant docs"

Average precision computes precision each time a relevant doc appears, averaged; MAP averages that over queries. A single robust number that accounts for both rank and completeness.

### Which to use?

- One correct answer matters most -> **MRR**.
- Ordering of many relevant docs matters -> **nDCG**.
- Overall precision/recall balance -> **MAP** / **precision@k** / **recall@k**. Report several; they illuminate different failure modes.

### A worked nDCG number

For ranked `[D0, D1, D2, D3]` with relevant `{D1, D3}` (hits at ranks 2 and 4):

```
DCG@4  = 1/log2(3) + 1/log2(5) = 0.6309 + 0.4307 = 1.0616
ideal  = 1/log2(2) + 1/log2(3) = 1.0000 + 0.6309 = 1.6309   (both relevant docs up front)
nDCG@4 = 1.0616 / 1.6309 = 0.651
```

(These exact numbers are asserted in the notebook, so you can trust the implementation.)

---

## Step 3 — The leaderboard

The notebook evaluates BM25, dense, hybrid, and (optionally) re-ranked retrieval on the same test set and prints a table:

```
retriever      nDCG@5   MRR    recall@5
BM25             ...     ...     ...
Dense            ...     ...     ...
Hybrid           ...     ...     ...
Dense+Rerank     ...     ...     ...
```

Now the progression across the course stops being a story and becomes evidence.

---

## Hands-on

Notebook: `notebooks/09_evaluating_search.ipynb`

You will:

1. Load the labeled queries and confirm the metric implementations against hand-computed values.
2. Produce rankings from BM25, dense, and hybrid retrievers over the eval queries.
3. Compute precision@k, recall@k, MRR, nDCG, MAP and assemble a **leaderboard**.
4. Interpret the differences — and notice how a tiny test set makes numbers noisy (a lesson in itself).

> BM25 evaluation runs fully offline. Dense/hybrid rows use the embedding model (your machine). The metric functions are verified independently with fixed rankings.

---

## Going deeper

- **Offline vs. online.** Offline metrics (this chapter) use a fixed labeled set. Online evaluation (**A/B testing**, interleaving) measures real user behavior — clicks, dwell time, conversions — which is the ultimate ground truth but slower and costlier to run.
- **Statistical significance.** With six queries, differences are noise. Real evaluations use hundreds/thousands of queries and significance tests (e.g., paired t-test) before declaring a winner.
- **Graded relevance & judgment cost.** nDCG shines with graded labels. Gathering judgments is expensive; pooling and tools like TREC-style qrels exist to manage it.
- **Metric gaming.** Optimizing one metric can hurt others (e.g., chasing recall floods results with marginal hits). Watch a basket of metrics.
- **Beyond relevance.** Production also weighs latency, diversity, freshness, and fairness — quality is multi-dimensional.

---

## Pitfalls & gotchas

- **Tuning on your test set.** If you tune hyperparameters against the same queries you report on, you overfit. Keep a separate dev/validation set.

- **Reading too much into a tiny set.** Six queries can't rank systems reliably; treat this chapter's numbers as illustrative, not definitive.
- **Comparing metrics at different k.** precision@5 and precision@10 aren't comparable; fix k.
- **Ignoring first-stage recall when evaluating re-rankers.** If retrieval missed the relevant doc, the re-ranker's ceiling is already capped — measure both stages.

---

## Key terms

relevance judgments, ground truth, precision@k, recall@k, MRR, nDCG, DCG, MAP, average precision, offline vs. online evaluation, A/B testing. *(See GLOSSARY.)*

---

## Check your understanding

1. Give the one-line question each of precision@k, recall@k, MRR, and nDCG answers.
2. Why does nDCG need to be *normalized*, and what by?
3. When is MRR the right metric to optimize, and when is it misleading?
4. Why are six queries not enough to declare one retriever better than another?
5. What's the danger of tuning k1, b, or alpha on the same queries you report metrics on?

---

## References


- Manning, Raghavan & Schütze, *Introduction to Information Retrieval*, evaluation chapter.
- Järvelin & Kekäläinen, *Cumulated Gain-based Evaluation of IR Techniques* (nDCG, 2002).
- TREC evaluation methodology and `trec_eval` (industry-standard tooling).
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search* (inspiration).

# Chapter 10 — RAG: Retrieval-Augmented Generation

**Level:** Intermediate - **Status:** Complete **Prerequisites:** Chapter 5 (dense retrieval). Chapters 6–9 helpful. **You'll build:** a grounded question-answering app over your own corpus.

---

## At a glance

So far our systems *retrieve* passages and hand them to the user. The final step is to let a **Large Language Model** read those passages and write a direct, natural-language **answer**. Done naïvely, LLMs hallucinate and go stale. **Retrieval-Augmented Generation (RAG)** fixes this by retrieving relevant context first and instructing the model to answer *from it* — turning search into an answer engine that is accurate, current, and citable. This chapter assembles everything from Chapters 1–9 into one pipeline.

---

## The motivation: why not just ask the LLM?

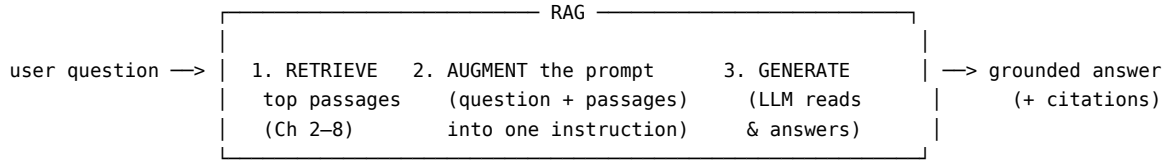
Ask an LLM a question directly and two problems surface:

1. **Hallucination** — it may produce a fluent, confident answer that is simply *wrong*, because it's generating plausible text, not looking anything up.
2. **Stale / missing knowledge** — it only knows what it saw in training. Your private documents, last week's news, and this quarter's numbers aren't in there.

RAG addresses both by **grounding** the model in retrieved evidence. The model no longer has to *remember* the answer — it just has to *read and summarize* the passages you give it.

---

## The RAG pattern




Three steps:

1. **Retrieve** — use any retriever we've built (dense, hybrid, + re-rank) to fetch the top few relevant passages for the question.
2. **Augment** — build a prompt that contains the question *and* those passages, with an instruction like *"Answer using only the context below; if the answer isn't there, say so."*
3. **Generate** — the LLM produces an answer grounded in the supplied context.

The retrieval half of this course is what makes the generation half trustworthy. **Better retrieval -> better answers.**

---

## Anatomy of a good RAG prompt

The prompt is where grounding is enforced. A solid template:

```
You are a helpful assistant. Answer the QUESTION using ONLY the CONTEXT below.
If the context does not contain the answer, say "I don't know based on the provided context."
Cite the passages you used by their [id].

CONTEXT:
[4] The blue whale is the largest animal known to have ever lived ...
[5] Although they live in the ocean, whales are mammals ...

QUESTION: what is the largest animal that has ever lived?

ANSWER:
```

Key ingredients:

- **A grounding instruction** ("use ONLY the context") to suppress hallucination.
- **A refusal clause** ("say I don't know") so the model declines instead of inventing.
- **The retrieved passages**, each tagged with an id for **citations**.
- **The user's question**, clearly delimited.

---

## The model we use

To keep the course open-source and key-free, the notebook defaults to a **small instruction-tuned model** (`google/flan-t5-base`) via Hugging Face `transformers`. It's tiny, runs on CPU, and is enough to demonstrate grounded answering. The notebook is written so you can **swap in a stronger model or a hosted API** by changing a single `generate()` function — the retrieval and prompt logic stay identical.

```python
from transformers import pipeline
llm = pipeline("text2text-generation", model="google/flan-t5-base")
answer = llm(prompt, max_new_tokens=128)[0]["generated_text"]
```

---

## Hands-on

Notebook: `notebooks/10_rag.ipynb`

You will:

1. Retrieve top passages for a question (reusing the Chapter 5 dense retriever).

2. **Assemble a grounded RAG prompt** from the question + passages (with ids for citations).
3. Generate an answer with a small open-source LLM (or your own `generate()`).
4. Demonstrate **grounding**: ask something the corpus *can't* answer and watch the model refuse instead of hallucinating.
5. Wrap it all in a single `rag_answer(question)` function — the capstone of the course so far.

> The retrieval + prompt-assembly logic runs and is tested without any model. The generation step downloads a small model on first run (internet once), or you can plug in an API.

---

## Going deeper

- **Grounding is not a guarantee.** Models can still ignore the context or over-generalize. Prompt design, and evaluating the *faithfulness* of answers to the context, matter.
- **Citations & attribution.** Returning the passage ids the answer used lets users verify claims — a major reason RAG beats a bare LLM for trust.
- **How many passages (context budget).** More context can help or hurt: too little misses the answer; too much dilutes it and risks the model latching onto a distractor ("lost in the middle"). Tune it.
- **Retrieval quality dominates.** Most RAG failures are *retrieval* failures — the answer wasn't in the passages. Improve the retriever (Chapters 6–8) before blaming the LLM.
- **RAG vs. fine-tuning.** RAG injects knowledge at query time (easy to update, citable); fine-tuning bakes it into weights (fast at inference, hard to update). They're complementary.

---

## Pitfalls & gotchas

- **No refusal clause.** Without "say I don't know," the model invents answers for unanswerable questions.
- **Dumping the whole corpus into the prompt.** Context windows are finite and models degrade with irrelevant text; retrieve, don't stuff.
- **Passages too big.** Long, unchunked passages waste context and bury the answer — motivating chunking (Chapter 11).
- **Blaming the LLM for retrieval bugs.** Always check *what was retrieved* before tuning the prompt or model.
- **No evaluation.** Answer quality should be measured (faithfulness, answer relevance), not eyeballed.

---

## Key terms

Large Language Model (LLM), hallucination, Retrieval-Augmented Generation (RAG), grounding, context window, prompt template, refusal, citation, faithfulness. *(See GLOSSARY.)*

---

## Check your understanding

1. Name the two problems with asking an LLM directly that RAG addresses, and how it addresses each.
2. List the three steps of RAG and what each contributes.
3. What two clauses in the prompt reduce hallucination, and how?
4. Why are most RAG failures actually *retrieval* failures?
5. When would you choose RAG over fine-tuning to give a model new knowledge?

---

## References


- Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (RAG, 2020).
- Liu et al., *Lost in the Middle: How Language Models Use Long Contexts* (2023).
- Hugging Face `transformers` documentation (open-source generation used here).
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search*, Lesson on generating answers (inspiration).

# Chapter 11 — Chunking & Production Pipelines

> **Level:** Intermediate -> Advanced - **Status:** Complete **Prerequisites:** Chapter 10 (RAG). **You'll build:** chunking strategies and a reusable ingestion pipeline — the engineering that turns a notebook demo into something real.

---

## At a glance

Our course corpus is 16 tidy one-paragraph passages. Real documents are long, messy, and plentiful: a 40-page PDF, a wiki with thousands of articles, a folder of transcripts. Before any of it can be searched, it has to be **loaded, cleaned, chunked, embedded, and indexed**. This chapter covers that ingestion pipeline — with **chunking** as the highest-leverage decision — plus the operational concerns (freshness, cost, debugging) that separate a demo from a system.

---

## Why chunk at all?

You can't embed a whole book as one vector. Three reasons chunking is mandatory:

1. **Model input limits.** Embedding models truncate long inputs (a few hundred tokens). Feed a whole document and everything past the limit is silently dropped.
2. **Retrieval precision.** One vector for a long document averages many topics into a blur. A user asking about page 30 shouldn't be matched to a vector dominated by pages 1–29. Smaller chunks = sharper matches.
3. **Answer quality (RAG).** You want to hand the LLM the *relevant paragraph*, not the whole document — it's cheaper and less distracting (recall "lost in the middle" from Chapter 10).

So: **split documents into chunks, embed each chunk, retrieve chunks.** The document id travels along as metadata so you can still cite the source.

---

## Chunking strategies

Implemented in `src/chunking.py`.

### Fixed-size (by words or tokens)

Cut every $N$ words with an **overlap** of $O$ words between neighbors.

```
size=40, overlap=10:
[ words 1–40 ][ words 31–70 ][ words 61–100 ]
              └─overlap─┘     └─overlap─┘
```

Simple and predictable. The **overlap** matters: without it, a sentence spanning a boundary gets cut in half and neither chunk contains the whole thought. Overlap re-includes the seam so context survives.

### Sentence-based

Group *whole sentences* (e.g., 3 at a time, overlapping by 1). Chunks read naturally because they never cut mid-sentence — usually better than fixed-size for prose.

### Semantic chunking

Split where the *topic* shifts (detected by drops in similarity between consecutive sentences). Smartest but most complex; great for heterogeneous documents.

### Choosing size & overlap

- **Too small:** chunks lack context; the answer is split across several chunks.
- **Too large:** chunks blur multiple topics; retrieval precision drops; more tokens per LLM call.
- **Rule of thumb:** start ~200–500 tokens with ~10–20% overlap, then **tune by measuring** (Chapter 9). There is no universal best — it depends on your documents and queries.

---

## The ingestion pipeline

Every production RAG system has the same backbone:

```
LOAD  ──>  CLEAN  ──>  CHUNK  ──>  EMBED  ──>  INDEX
(read      (strip      (split      (Ch 4)      (Ch 6 vector DB,
 files)     boilerplate, into                   with chunk text +
            normalize)  chunks)                 source metadata)
```

- **Load** — read PDFs, HTML, Markdown, transcripts into raw text.
- **Clean** — strip navigation/boilerplate, fix encoding, normalize whitespace.
- **Chunk** — apply a strategy above; attach metadata (`doc_id`, `title`, position).
- **Embed** — encode each chunk (batch for speed).
- **Index** — store vectors + chunk text + metadata in a vector database (Chapter 6).

The notebook builds a small, configurable `IngestionPipeline` class that runs this end to end and lets you swap the chunker and embedder.

---

## Operating it (the unglamorous but essential part)

- **Batching.** Embed chunks in batches, not one at a time — often 10–100x faster.
- **Caching.** Don't re-embed unchanged documents; hash content and skip if seen.
- **Freshness / re-indexing.** Documents change. Decide on full rebuilds vs. incremental upserts/deletes, and how often.
- **Cost & latency.** Embedding and generation dominate cost. Measure tokens, batch, and cache.

- **Monitoring.** Track retrieval hit rate, latency, and answer quality over time — regressions are invisible without it.

---

## Debugging a RAG system

When answers are bad, localize the failure — don't randomly tweak the prompt:

```
Bad answer?
  ├─ Was the right chunk retrieved?  ──no──>  RETRIEVAL problem
  │      ->  check chunking (too big/small?), embedding model, query, k, hybrid/rerank
  └─ Right chunk retrieved but answer still wrong?  ──> GENERATION problem
         ->  check the prompt (grounding/refusal), context budget, the model
```

The single most common root cause is **retrieval, not generation** — and often the fix is *chunking*. Print what was retrieved before touching anything else.

---

## Hands-on

Notebook: `notebooks/11_chunking_and_pipelines.ipynb`

You will:

1. Apply fixed-size (with overlap) and sentence-based chunking to a long document and compare.
2. See *why* overlap matters with a boundary-straddling sentence.
3. Build a configurable `IngestionPipeline` (load -> clean -> chunk -> embed -> index).
4. Show that chunking a long document improves retrieval precision over embedding it whole.
5. Walk the retrieval-vs-generation debugging checklist on a deliberately broken example.

> Chunking logic and the pipeline scaffold run offline. The embed/index steps use the model + Chroma (your machine).

---

## Going deeper

- **Token vs. word chunking.** Embed limits are in *tokens*; word counts approximate. For precise control, chunk by tokens with the model's tokenizer (e.g., `tiktoken`).
- **Metadata-rich chunks.** Store section headers, page numbers, timestamps — they power filtering (Chapter 6) and better citations.
- **Parent-document / small-to-big retrieval.** Retrieve on small chunks for precision but hand the LLM the surrounding *parent* passage for context. A popular, effective pattern.
- **Deduplication & near-duplicates.** Large corpora have repeats; dedupe to avoid burning context and skewing retrieval.
- **Incremental indexing at scale.** Upserts, tombstones for deletes, and background re-embedding keep a live index fresh without full rebuilds.

---

## Pitfalls & gotchas

- **No overlap.** Answers that straddle chunk boundaries get lost. Always overlap a little.
- **One-size-fits-all chunking.** Code, tables, and prose want different strategies; don't force one.
- **Chunking away structure.** Splitting mid-table or mid-code-block destroys meaning; respect structural boundaries.
- **Forgetting metadata.** Without `doc_id`/position you can't cite sources or filter — cripples RAG trust.
- **Re-embedding everything on every change.** Cache; embed only what changed.

---

## Key terms

chunking, chunk size, chunk overlap, fixed-size chunking, sentence chunking, semantic chunking, ingestion pipeline, batching, caching, re-indexing, parent-document retrieval, upsert. *(See GLOSSARY.)*

---

## Check your understanding

1. Give three reasons you must chunk long documents before embedding them.
2. What problem does chunk *overlap* solve? What goes wrong without it?
3. Describe the five stages of an ingestion pipeline in order.
4. A RAG answer is wrong. What's the first thing to check, and why?
5. Why is there no universal best chunk size, and how do you find a good one for your data?

---

## References


- Liu et al., *Lost in the Middle* (2023) — why context size and placement matter.
- LangChain / LlamaIndex documentation on text splitters and ingestion (concepts, not required).
- `tiktoken` for token-accurate chunking.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search* (inspiration).

# Chapter 12 — Advanced Topics

> **Level:** Advanced - **Status:** Complete **Prerequisites:** Chapters 5–11. **You'll build:** familiarity with the techniques that push retrieval beyond the core pipeline — and one runnable late-interaction demo.

---

## At a glance

Once the core stack (retrieve -> fuse -> re-rank -> generate) is solid, these techniques squeeze out more quality or unlock new capabilities. Treat this as a **menu of deeper dives**: each section is a concept plus a pointer to primary sources, and the notebook includes concrete demos — most notably a from-scratch **ColBERT-style late-interaction** score you can run and inspect.

---

## 1. Query understanding: the query is a lossy clue

The user's query is a thin, imperfect stand-in for their information need (Chapter 1). Several techniques reconstruct a better one:

- **Query expansion** — add synonyms/related terms so keyword and dense retrieval match more. *"heart attack" -> "heart attack, myocardial infarction, MI"*.
- **Query rewriting** — rephrase a messy or conversational query into a clean search query (crucial in chat: *"and what about its capital?" -> "what is the capital of Australia?"*).
- **HyDE (Hypothetical Document Embeddings)** — a clever trick: ask an LLM to *write a hypothetical answer* to the query, then embed **that** and retrieve with it. A fake answer is often closer in embedding space to the real answer passages than the short question is.

```
HyDE:  question ──LLM──> hypothetical answer ──embed──> retrieve real passages
```

---

## 2. Multi-vector retrieval (ColBERT) & late interaction

A bi-encoder (Chapter 5) squashes a whole passage into **one** vector — lossy. A cross-encoder (Chapter 7) is accurate but can't precompute. **ColBERT** is the middle ground: represent each text as **one vector per token**, and score a query–document pair with **MaxSim** — for each query token, take its best-matching document token, then sum:

$$\text{score}(q, d) = \sum_{i \in q} \max_{j \in d} \cos(\mathbf{q}_i, \mathbf{d}_j)$$

This "late interaction" keeps token-level detail (more accurate than a single vector) while still allowing document token vectors to be **precomputed** (scalable, unlike a cross-encoder). The notebook implements MaxSim in a few lines of numpy so you can see exactly how it rewards fine-grained matches.

---

## 3. Multilingual & multimodal search

- **Multilingual** — models trained on many languages place *"dog"* and *"perro"* near each other, so a query in one language retrieves documents in another. One embedding space, many languages.
- **Multimodal** — models like CLIP embed **images and text into the same space**, so a text query can retrieve images (and vice versa). Same nearest-neighbor machinery, new modality. Extends to audio and video.

---

## 4. Fine-tuning embedding models

Off-the-shelf models are general. If your domain is unusual (legal, medical, code, your product's jargon), **fine-tuning** on your own (query, relevant-passage) pairs — usually with contrastive learning — can lift retrieval quality substantially. The cost is building training data and a training loop; the payoff is a space shaped to *your* notion of relevance.

---

## 5. Agentic RAG & multi-step retrieval

Simple RAG retrieves once and answers. Hard questions need more:

- **Multi-hop** — decompose a question, retrieve for each part, combine (*"Which is older, the company that makes X or the one that makes Y?"* needs two lookups).
- **Iterative / agentic** — an LLM *agent* decides what to search, reads results, and searches again until it can answer — a loop of retrieve -> reason -> retrieve.
- **Tool use** — the agent chooses among tools (search, calculator, SQL, web) per step.

More powerful, but harder to control, evaluate, and keep cheap — use when single-shot RAG genuinely falls short.

---

## Hands-on

Notebook: `notebooks/12_advanced_topics.ipynb`

You will:

1. Do simple **query expansion** and see recall change.
2. Trace the **HyDE** flow (with a mock generator so it runs anywhere).
3. Implement **ColBERT-style MaxSim late interaction** from scratch in numpy and compare it to single-vector cosine on a crafted example.
4. See a sketch of **multimodal** (shared-space) retrieval and where to plug in CLIP.

The MaxSim and query-expansion demos run fully offline. HyDE and real embedding/CLIP steps are written to plug into a model on your machine.

---

## Going deeper

- **HyDE caveats.** It adds an LLM call (latency/cost) and can hurt when the model hallucinates a misleading hypothetical. Measure before adopting.
- **ColBERT storage.** One vector per token means much larger indexes; production uses compression and specialized indexes (PLAID).
- **Cross-encoder vs. ColBERT vs. bi-encoder.** A spectrum of accuracy/cost: bi-encoder (cheapest) -> ColBERT (mid) -> cross-encoder (most accurate, most expensive). Pick per stage and budget.
- **Evaluation still rules.** Every technique here should earn its place by improving Chapter 9 metrics on *your* data, not by reputation.

---

## Pitfalls & gotchas

- **Adding complexity without measuring.** Each advanced trick adds latency/cost/failure modes; justify it with metrics.
- **HyDE/agent loops without guards.** Unbounded LLM steps blow up cost and latency; cap them.
- **Assuming multilingual/multimodal models are as strong as specialized ones.** Often a trade-off; verify on your task.
- **Fine-tuning on tiny or biased data.** You can overfit and *worsen* general retrieval; you need enough diverse pairs.

---

## Key terms

query expansion, query rewriting, HyDE, ColBERT, late interaction, MaxSim, multi-vector retrieval, multilingual embeddings, multimodal search, CLIP, fine-tuning, contrastive learning, agentic RAG, multi-hop retrieval. *(See GLOSSARY.)*

---

## Check your understanding

1. Explain HyDE in one sentence and why embedding a *hypothetical answer* can beat embedding the question.
2. What does ColBERT keep that a bi-encoder throws away, and how does MaxSim use it?
3. Why can ColBERT precompute document vectors while a cross-encoder can't?
4. How does multimodal search reuse the exact machinery from Chapter 5?
5. When is agentic/multi-step RAG worth its extra cost over single-shot RAG?

---

## References


- Gao et al., *Precise Zero-Shot Dense Retrieval without Relevance Labels* (HyDE, 2022).
- Khattab & Zaharia, *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT* (2020).
- Radford et al., *Learning Transferable Visual Models from Natural Language Supervision* (CLIP, 2021).
- Reimers & Gurevych, multilingual sentence-transformers.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search* (inspiration).

# Chapter 13 — Capstone Project

> **Level:** Intermediate - **Status:** Complete **Prerequisites:** All previous chapters. **You'll build:** a complete semantic-search + RAG system end to end, over a corpus you care about, and assess it against a rubric.

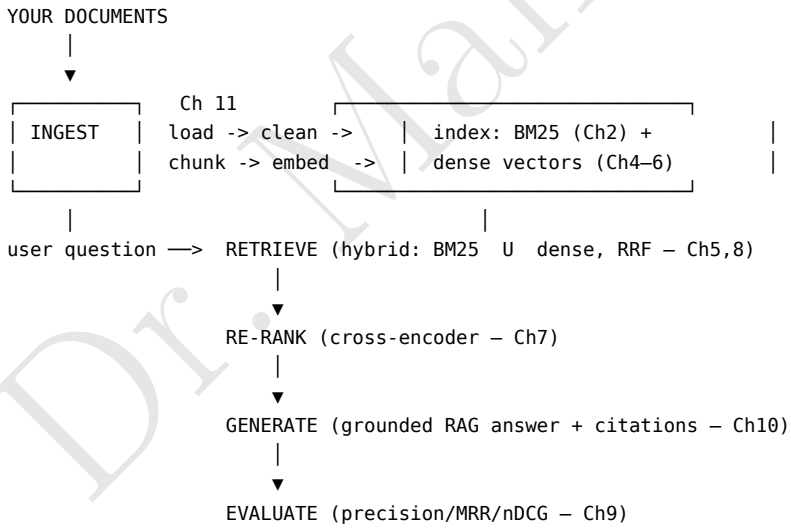
---

## At a glance

This is where everything converges. Across twelve chapters you built each piece in isolation — keyword search, embeddings, dense retrieval, vector indexes, re-ranking, fusion, evaluation, RAG, chunking. The capstone connects them into **one working answer engine** and asks you to run it on *your own* documents. The notebook provides a **reference implementation** (a `SearchStack` class) you can study, then adapt.

---

## The full stack you're assembling




Every arrow is a chapter you've already done. The capstone is the wiring — and the judgment about which pieces your use case actually needs.

---

## Step 1 — Choose a corpus and use case

Pick something you'll enjoy querying and can judge relevance for:

- Your own notes / a wiki / documentation.
- A set of articles, papers, or blog posts on a topic.
- A product catalog or FAQ.
- Meeting transcripts or a book.

Write down the **use case**: is it a search box (return passages) or a Q&A assistant (return answers)? That decides whether you stop at retrieval or go all the way to RAG, and which metric matters (nDCG for ranking; MRR/answer-quality for single-answer).

---

## Step 2 — Build the stack

The notebook's `SearchStack` wires the pieces together with sensible defaults. In outline:

```
stack = SearchStack(
    encode_fn  = embedder.encode,      # Ch4 (encode_fn = any list[str] -> vectors callable)
    chunker    = sentence_chunks,      # Ch11
    reranker   = cross_encoder,        # Ch7  (pass None to skip re-ranking)
    generator  = my_llm,               # Ch10 (pass None for search-only)
    use_hybrid = True,                 # Ch2 + Ch5 + Ch8 (RRF)
).ingest(my_documents)                 # Ch11 pipeline

stack.search("a question")     # ranked passages
stack.answer("a question")     # grounded RAG answer + citations
stack.evaluate(my_eval_set)    # Ch9 metrics
```

Start minimal (dense only), confirm it works, then switch on hybrid, re-ranking, and RAG one at a time — **measuring after each** so you know which additions actually help on *your* data.

---

## Step 3 — Evaluate honestly

Build a small labeled set (10–30 queries with relevant doc/chunk ids — the more the better) as in Chapter 9, and report metrics for each configuration:

```
config                nDCG@10   MRR    notes
dense only             ...      ...
+ hybrid (RRF)         ...      ...
+ cross-encoder rerank ...      ...
```

Let the numbers — not vibes — decide your final configuration. Note where the system still fails; that's your roadmap.

---

## Self-assessment rubric

Score your project honestly. Each item is a level you can reach.

| Area | Baseline | Good | Excellent |
|---|---|---|---|
| **Ingestion** | Loads & chunks one doc type | Cleans + chunks with overlap + metadata | Handles multiple formats; caching/re-index |

| Area | Baseline | Good | Excellent |
|---|---|---|---|
| **Retrieval** | Dense retrieval works | Hybrid (BM25 + dense, RRF) | Hybrid + re-rank, tuned by measurement |
| **Generation** | Returns passages | Grounded RAG answers | Citations + refusal on unanswerable |
| **Evaluation** | A few queries eyeballed | Labeled set + precision/recall | nDCG/MRR across configs, dev/test split |
| **Engineering** | Runs in a notebook | Reusable functions/classes | Config-driven, documented, reproducible |
| **Understanding** | Can run it | Can explain each stage | Can justify every design choice & trade-off |

Aim for at least **Good** across the board before reaching for **Excellent** anywhere — a balanced system beats one spiky part.

---

## Hands-on

Notebook: `notebooks/13_capstone.ipynb`

The notebook provides the `SearchStack` reference implementation wired over the course corpus, plus a clearly marked **"bring your own corpus"** cell. You will:

1. Ingest documents (chunk -> embed -> index).
2. Retrieve with hybrid search and re-rank.
3. Generate grounded answers with citations.
4. Evaluate configurations and pick a winner.
5. Swap in your own corpus and repeat.

> The wiring/logic runs with mock components in-sandbox; embeddings, cross-encoder, and LLM run on your machine (or via an API).

---

## Extension ideas (make it a portfolio piece)

- **A UI** — wrap it in a small Streamlit/Gradio app with a search box and answer panel.
- **Persistence & scale** — move to a persistent vector DB (Chroma/FAISS on disk); handle updates.
- **Deployment** — expose a `/search` and `/answer` API; add caching and logging.
- **New modality** — add image search with CLIP (Chapter 12).
- **Better evaluation** — grow the labeled set; add answer-faithfulness checks.
- **Advanced retrieval** — try HyDE, ColBERT, or a fine-tuned embedding model (Chapter 12) and measure the lift.

---

## You've finished the course

You can now explain *and build* a modern search system from keyword matching to grounded RAG, measure it properly, and make deliberate engineering trade-offs. That's the full arc from

"matching words" to "matching meaning" — and shipping it.

Revisit the ROADMAP to review, the GLOSSARY to lock in terms, and any chapter's *Going Deeper* section to specialize.

---

## Check your understanding

1. For your use case, do you need RAG or is retrieval enough? Justify it.
2. Which single addition (hybrid, re-rank, RAG) gave the biggest measured lift, and why?
3. Where does your system still fail, and which chapter's technique would you try next?
4. If latency had to drop 5x, what would you cut first, and what quality cost would you accept?
5. Explain your final configuration to someone who's read only Chapter 1.

---

## References


- Every prior chapter of this course.
- Lin, Nogueira & Yates, *Pretrained Transformers for Text Ranking: BERT and Beyond* (2021) — a broad survey tying these pieces together.
- DeepLearning.AI x Cohere, *Large Language Models with Semantic Search* (inspiration).

# Answers to Check Your Understanding

*Model answers to the end-of-chapter questions. Try each question yourself before reading these.*

## Chapter 0 — Introduction

**Q1. In one sentence, why does keyword search miss relevant results?**

Because it matches literal *words*, not *meaning*: if the best document phrases the same idea with different words, keyword search sees no overlap and misses it.

**Q2. What does the "Retrieve" step do, and how does "semantic" change it?**

**Retrieve** quickly pulls a small set of likely-relevant documents from the whole corpus. Making it *semantic* means comparing the *meaning* of the query and documents (via embeddings) instead of shared words, so paraphrases still match.

**Q3. Which pipeline stages are optional, and when would you skip them?**

**Re-rank** and **Generate** are optional. Skip re-ranking when first-stage results are already good enough or latency is tight; skip generation when you just want to return passages (a search box) rather than a written answer.

## Chapter 1 — Foundations of Information Retrieval

**Q1. Explain the difference between an *information need* and a *query* in your own words.**

An **information need** is what the user actually wants to know; the **query** is the short, imperfect text they type to express it. The query is a lossy stand-in for the need.

**Q2. Why does a search system build an index *before* any query arrives?**

So it can answer in milliseconds. Building the **index** ahead of time means the system doesn't have to re-read every document at query time — it just looks up the precomputed structure.

**Q3. Name the two classic stages of a search system and what each optimizes for.**

**Retrieval** (stage 1) casts a wide, cheap net over the whole corpus, optimizing for speed and recall; **re-ranking** (stage 2) reorders the small candidate set precisely, optimizing for accuracy.

**Q4. Give your own example of two texts that mean the same thing but share no keywords.**

Any true paraphrase works, e.g. *"side of the head hurts"* and *"pain localized to one temple"* — same meaning, essentially no shared content words.

**Q5. Why is counting shared words a poor measure of relevance when the words are common?**

Common words like *the* and *is* appear in almost every document, so shared counts of them signal nothing about relevance — they inflate scores without indicating meaning. (IDF fixes this in Chapter 2.)

## Chapter 2 — Keyword / Lexical Search

**Q1. Why is it called an *inverted* index, and what problem does it solve?**

It maps term -> the documents containing it (the inverse of document -> words). This lets a query be answered by a direct lookup instead of scanning every document, which is what makes search fast.

**Q2. In the worked example, why does the word "is" contribute nothing to the score?**

Its IDF is 0: *is* appears in every document, so $\ln(N/df) = \ln(N/N) = 0$. A term in every document carries no power to distinguish documents, so it adds nothing to the score.

**Q3. What two problems does BM25 fix compared to plain TF-IDF, and which parameter controls each?**

**Term saturation** (extra occurrences give diminishing returns) — controlled by **k1**; and **length normalization** (long documents shouldn't win just for being long) — controlled by **b**.

**Q4. Predict: for the query "blue", will D1 (sky) or D4 (whale) score higher, and what would**

**D1 (sky) scores higher.** *blue* appears twice in D1 but once in D4, and D1 is shorter, so BM25's TF and length-normalization both favor D1. It would flip if D4 repeated *blue* more often or were much shorter than D1.

**Q5. Give a query where keyword search would fail on our corpus, and explain why.**

*"strong pain in the side of the head"* — the answer (the migraine passage) says *"sharp sensation localized to one temple"* and shares almost no words, so keyword search buries it.

## Chapter 3 — From Text to Vectors

**Q1. Why are all distinct one-hot vectors equally far apart, and why is that a problem?**

Every one-hot vector has a single 1 in a unique slot, so any two distinct words differ in exactly two coordinates and sit the same distance apart. That means the representation encodes identity but zero similarity of meaning.

**Q2. What information does bag-of-words throw away, and what does it keep?**

It throws away **word order** (and grammar); it keeps **which words appear and how often**. So "dog bites man" and "man bites dog" look identical.

**Q3. In one sentence, what does cosine similarity measure that the dot product doesn't isolate?**

Cosine isolates **direction** (the angle), ignoring vector **length/magnitude**; the raw dot product mixes both, so it grows with length even when the direction (meaning) is unchanged.

**Q4. Two vectors point the same way but one is twice as long. What is their cosine similarity?**

**1.0** — same direction means a cosine of 1 regardless of length. (The dot product would differ, but cosine divides out magnitude.)

**Q5. When would Euclidean distance and cosine similarity give the *same* ranking of neighbors?**

When the vectors are **L2-normalized** (scaled to unit length): then nearest-by-cosine and nearest-by-Euclidean produce the identical ranking.

## Chapter 4 — Embeddings Deep Dive

**Q1. In one sentence, what is an embedding model trained to do?**

To map text to a dense vector so that texts humans consider similar get vectors that are close (high cosine), and dissimilar texts get vectors that are far apart.

**Q2. Why does the word "bank" get one vector in word2vec but different vectors in BERT?**

word2vec assigns **one fixed vector per word** regardless of context, so both senses of *bank* collapse to the same vector; BERT produces **contextual** vectors that depend on the surrounding sentence, so the two *banks* differ.

**Q3. Why can a bi-encoder's document vectors be precomputed, and why does that matter for search?**

A bi-encoder embeds each document **independently of the query**, so document vectors never change per query and can be computed once and reused. That's essential because at search time you only embed the short query and do fast vector math over the precomputed index.

**Q4. Explain *why* a question's nearest embedding tends to be its own answer.**

The model was trained (contrastively) to place related texts close together, and a question and its answer are strongly related, so the answer's embedding is typically the nearest neighbor — even without shared words.

**Q5. You switch from a 384-dim model to a 768-dim one. What must you redo, and why?**

You must **re-embed the entire corpus** (and the queries) with the new model. Vectors from different models live in different spaces and aren't comparable, so the old 384-dim index is unusable.

## Chapter 5 — Dense Retrieval & Semantic Search

**Q1. State the three steps of dense retrieval in your own words.**

(1) Embed every document once into a matrix (indexing); (2) embed the query; (3) score documents by similarity to the query vector and return the top-k nearest.

**Q2. Why can a bi-encoder precompute document vectors, and why is that essential for scale?**

Because a bi-encoder embeds documents without seeing the query, their vectors are query-independent and can be computed once and reused for every search. Without that, you couldn't search a large corpus fast.

**Q3. Give one query where BM25 beats dense retrieval, and explain why.**

Exact terms it doesn't understand semantically — e.g. a product code like `SKU-4471-B`, a rare proper name, or the exact string `Lucene`; keyword search matches these exactly while a dense model may drift.

**Q4. Why do we L2-normalize the embeddings before taking dot products?**

So that the dot product equals cosine similarity. Normalizing removes magnitude effects (which often just reflect length), leaving direction (meaning) — and it's faster than computing full cosine each time.

**Q5. Our search is "brute force." What does that mean, and when does it stop being good enough?**

"Brute force" means scoring the query against **every** document (exact, O(N)). It stops being good enough when N grows large (hundreds of thousands+), where the per-query cost becomes too slow — motivating ANN (Chapter 6).

## Chapter 6 — Vector Databases & Approximate Nearest Neighbor (ANN)

**Q1. Why does brute-force search stop being viable as the corpus grows?**

Its cost is proportional to N (corpus size) x dimensions, so every query scans the whole corpus. As N grows to millions, that linear cost blows past the milliseconds users expect.

**Q2. What exactly does ANN approximate, and how do we measure the cost of that approximation?**

It approximates the **true set of nearest neighbors** — you may miss a few. The cost of that approximation is measured by **recall@k**: the fraction of the exact top-k that the approximate index actually returns.

**Q3. Give a one-sentence intuition for how HNSW avoids scanning every vector.**

It builds a navigable graph linking each vector to near neighbors (with express-lane upper layers), so a query greedily hops toward its neighborhood, visiting only a small subset instead of all vectors.

**Q4. What does a vector *database* add on top of a raw ANN *index*?**

**Persistence** (save/reload the index), **metadata + filtering**, **CRUD** (add/update/delete), and serving/scaling — everything around the raw ANN index needed to run it as a real service.

**Q5. You have 5,000 vectors. Should you reach for HNSW? Why or why not?**

No — with only 5,000 vectors, exact search is already fast (sub-millisecond) and simpler. ANN pays off at much larger scales; under ~100k vectors, exact search is usually fine.

## Chapter 7 — Re-ranking

**Q1. In one sentence, what can a cross-encoder do that a bi-encoder can't — and what does it cost?**

A cross-encoder reads the query and a document **together**, giving a far more accurate relevance score; the cost is it must run the model for every query-document pair, so it can't be precomputed.

**Q2. Why can't we just use the cross-encoder as the first-stage retriever?**

Because it can't precompute: scoring N documents means N model runs **per query**, which is hopelessly slow over a full corpus. The bi-encoder narrows the field first so the cross-encoder only scores a few candidates.

**Q3. What limits how much a re-ranker can improve results, no matter how good it is?**

**First-stage recall.** A re-ranker only reorders the candidates it's given; if the relevant document never made the first-stage top-k, no re-ranker can surface it.

**Q4. You retrieve top-10 and re-rank, but quality is still poor. Name two different fixes and when**

Either (a) the right document isn't in the top-10 -> improve **first-stage retrieval** (bigger k, hybrid, better embeddings); or (b) it is but ranked wrong -> use a **stronger/better-matched re-ranker**. Diagnose by checking whether the relevant doc is in the candidate set.

**Q5. Where in a two-stage pipeline would you add a "prefer recent documents" rule, and why there?**

In the **re-ranking** stage — it's the natural place to blend extra signals (recency, popularity, business rules) with the relevance score, on the small candidate set, without slowing first-stage retrieval.

## Chapter 8 — Hybrid Search

**Q1. Why can't you simply add a BM25 score and a cosine score together?**

They live on different scales (BM25 is unbounded positive; cosine is roughly [-1, 1]), so summing lets BM25's larger numbers dominate arbitrarily and swamp the semantic signal.

**Q2. Explain, in one sentence, what RRF uses instead of raw scores and why that helps.**

It uses each document's **rank** in each list (via 1/(k+rank)), not the raw scores — so it never compares incompatible score scales and is robust to outliers; documents ranked highly by multiple retrievers win.

**Q3. In the worked example, why do D2 and D4 outrank D10 and D7?**

Because both retrievers ranked D2 and D4 highly, so they accumulate reciprocal-rank contributions from both lists; D10 and D7 appear in only one list, contributing from a single source and scoring lower.

**Q4. Give a query where hybrid clearly beats both BM25-only and dense-only, and say why.**

A query mixing an exact term with fuzzy intent, e.g. *"Lucene for semantic search"* or *"SKU-4471-B waterproof jacket"*: keyword nails the exact token, dense captures the intent, and RRF keeps documents both liked on top.

**Q5. Where does re-ranking (Chapter 7) fit relative to fusion in the full stack?**

**After** fusion: hybrid (BM25 U dense -> RRF) produces the candidate set, then the cross-encoder re-ranks those candidates for the final order (BM25 U dense -> RRF -> re-rank -> RAG).

## Chapter 9 — Evaluating Search

**Q1. Give the one-line question each of precision@k, recall@k, MRR, and nDCG answers.**

**Precision@k**: of the top-k I showed, how many were relevant? **Recall@k**: of all relevant docs, how many did I find in the top-k? **MRR**: how high was the first relevant result? **nDCG**: are the most relevant results near the top?

**Q2. Why does nDCG need to be *normalized*, and what by?**

Because raw DCG grows with the number of relevant documents and isn't comparable across queries; dividing by the **ideal DCG** (the best possible ordering) rescales it to [0, 1] so queries are comparable.

**Q3. When is MRR the right metric to optimize, and when is it misleading?**

Right when only the single best answer matters (e.g. one passage for RAG, an "I feel lucky" result). Misleading when there are many relevant documents and you care about finding all of them or their overall order — MRR only looks at the first hit.

**Q4. Why are six queries not enough to declare one retriever better than another?**

The results are too noisy: with so few queries, one lucky or unlucky query swings the average, and differences between systems aren't statistically significant. Real evaluations use hundreds+ of queries.

**Q5. What's the danger of tuning k1, b, or alpha on the same queries you report metrics on?**

You **overfit** to those queries — you'll pick settings that look good on the test set but don't generalize. Tune on a separate dev/validation split and report on a held-out test set.

## Chapter 10 — RAG: Retrieval-Augmented Generation

**Q1. Name the two problems with asking an LLM directly that RAG addresses, and how it addresses each.**

**Hallucination** (fluent but false answers) and **stale/missing knowledge**. RAG retrieves relevant, current passages and instructs the model to answer *from them*, so answers are grounded in real evidence and can include your private/up-to-date documents.

**Q2. List the three steps of RAG and what each contributes.**

**Retrieve** relevant passages for the question; **augment** the prompt with those passages (plus instructions); **generate** an answer grounded in that context. Retrieval supplies the facts; the prompt enforces grounding; generation writes the answer.

**Q3. What two clauses in the prompt reduce hallucination, and how?**

The **grounding instruction** ("use ONLY the context") stops the model from drawing on unverified memory, and the **refusal clause** ("say I don't know if it's not there") gives it permission to decline instead of inventing an answer.

**Q4. Why are most RAG failures actually *retrieval* failures?**

Because the LLM can only answer from what it's given: if the retriever didn't surface the passage containing the answer, the model has nothing correct to work from — so fixing retrieval (chapters 6-8) usually matters more than tweaking the prompt.

**Q5. When would you choose RAG over fine-tuning to give a model new knowledge?**

When knowledge changes often, must be **citable/verifiable**, or is private/large — RAG injects it at query time and is easy to update. Fine-tuning bakes knowledge into weights (fast at inference, hard to update) and suits fixed style/behavior more than fast-changing facts.

## Chapter 11 — Chunking & Production Pipelines

**Q1. Give three reasons you must chunk long documents before embedding them.**

(1) Embedding models have **input limits** and silently truncate long text; (2) one vector for a long document **blurs many topics**, hurting retrieval precision; (3) for RAG you want to hand the LLM the **relevant paragraph**, not a whole document (cheaper, less distracting).

**Q2. What problem does chunk *overlap* solve? What goes wrong without it?**

Overlap prevents a thought that **straddles a chunk boundary** from being cut in half so neither chunk contains it. Without overlap, answers spanning a seam are lost because no single chunk holds the complete context.

**Q3. Describe the five stages of an ingestion pipeline in order.**

**Load -> Clean -> Chunk -> Embed -> Index.** (Read raw text, strip boilerplate/normalize, split into chunks with metadata, encode each chunk, store vectors + text + metadata in a vector DB.)

**Q4. A RAG answer is wrong. What's the first thing to check, and why?**

**What was actually retrieved.** Most RAG failures are retrieval failures; if the right chunk wasn't retrieved, fix retrieval/chunking, not the prompt. Only if the right chunk *was* retrieved do you debug the generation step.

**Q5. Why is there no universal best chunk size, and how do you find a good one for your data?**

Because it depends on your documents and queries: too small loses context, too large blurs topics and wastes tokens. Find a good value by **measuring** (Chapter 9 metrics) across a few sizes/overlaps on your own data.

## Chapter 12 — Advanced Topics

**Q1. Explain HyDE in one sentence and why embedding a *hypothetical answer* can beat embedding the**

HyDE asks an LLM to write a **hypothetical answer** to the query and retrieves using that answer's embedding. It helps because a full answer-like passage sits closer in embedding space to the real answer passages than a short, sparse question does.

**Q2. What does ColBERT keep that a bi-encoder throws away, and how does MaxSim use it?**

It keeps **token-level (per-token) vectors** instead of squashing the text into one vector. **MaxSim** scores a pair by, for each query token, taking its best-matching document token's similarity and summing — preserving fine-grained matches.

**Q3. Why can ColBERT precompute document vectors while a cross-encoder can't?**

Because ColBERT embeds each document's tokens **independently of the query**, so those token vectors can be precomputed and stored. A cross-encoder must read query and document jointly, so nothing can be precomputed.

**Q4. How does multimodal search reuse the exact machinery from Chapter 5?**

A multimodal model (e.g. CLIP) embeds images and text into **one shared space**, so a text query vector can be compared to image vectors with the **same nearest-neighbor search** used for text-to-text retrieval in Chapter 5.

**Q5. When is agentic/multi-step RAG worth its extra cost over single-shot RAG?**

When a single retrieve-then-answer pass isn't enough — e.g. **multi-hop** questions needing several lookups, or queries requiring reasoning/tools between retrievals. It costs more latency and complexity, so reserve it for questions single-shot RAG genuinely can't handle.

## Chapter 13 — Capstone Project

**Q1. For your use case, do you need RAG or is retrieval enough? Justify it.**

*(Reflective — model answer.)* Choose **retrieval only** if users just need to find and read source passages (a search box); choose **RAG** if they need a synthesized, direct answer. Justify by the user's task and whether a written, citable answer adds value over a ranked list.

**Q2. Which single addition (hybrid, re-rank, RAG) gave the biggest measured lift, and why?**

*(Reflective.)* Report it from your own measurements. Typically **hybrid** helps most when queries mix exact terms and intent; **re-ranking** helps most when first-stage recall is good but ordering is off; **RAG** helps when users want answers, not passages. The right answer is whichever moved your Chapter 9 metrics most on *your* data.

**Q3. Where does your system still fail, and which chapter's technique would you try next?**

*(Reflective.)* Identify the failing query type, then map it to a technique: missed synonyms -> better embeddings/hybrid (Ch4/8); wrong ordering -> re-ranking (Ch7); answer not in a chunk -> chunking (Ch11); hallucinated answers -> prompt grounding (Ch10); missed exact terms -> hybrid (Ch8).

**Q4. If latency had to drop 5x, what would you cut first, and what quality cost would you accept?**

*(Reflective.)* Usually drop the most expensive stage first — the **cross-encoder re-ranker** (or shrink its candidate set), then consider ANN over exact search. Accept the resulting small drop in nDCG/MRR; measure it so the latency/quality trade-off is explicit.

**Q5. Explain your final configuration to someone who's read only Chapter 1.**

*(Reflective.)* In plain terms: "We turn every document into numbers that capture meaning, find the ones closest to the question, [optionally re-check the top few carefully], and [optionally] let an AI write an answer using only those documents." Tie each piece back to the words-vs-meaning idea from Chapter 1.

# Glossary

A plain-language dictionary of every key term used across the course. Each entry gives a one-line definition, a slightly fuller explanation, and the chapter where it's introduced. Terms are grouped by theme; within each group they're ordered roughly from foundational to advanced.

> Tip: when a chapter introduces a term, it links back here. As we write chapters, we'll keep this file the single source of truth for definitions.

---

## Core search vocabulary

**Information need** — what the user actually wants to know, which they express imperfectly as a query. *(Ch 1)*

**Query** — the text a user submits to search. *(Ch 1)*

**Document** — a single retrievable unit of text (an article, paragraph, product description, etc.). *(Ch 1)*

**Corpus** — the full collection of documents you search over. *(Ch 1)*

**Relevance** — how well a document satisfies the user's information need. The thing search tries to maximize. *(Ch 1)*

**Retrieval** — the act of selecting candidate documents for a query from the corpus. *(Ch 1)*

**Ranking** — ordering retrieved documents so the most relevant appear first. *(Ch 2)*

---

## Classical / lexical search

**Lexical search** — matching on the literal words (lexemes) in the query and documents. Also called keyword search. *(Ch 2)*

**Tokenization** — splitting text into smaller units (tokens), usually words or subwords. *(Ch 2)*

**Stop words** — extremely common words (the, a, of) often removed because they carry little meaning. *(Ch 2)*

**Stemming / Lemmatization** — reducing words to a base form ("running" -> "run") so variants match. *(Ch 2)*

**Inverted index** — a lookup table mapping each term to the list of documents containing it; what makes keyword search fast. *(Ch 2)*

**Term frequency (TF)** — how often a term appears in a document. *(Ch 2)*

**Inverse document frequency (IDF)** — how rare a term is across the corpus; rare terms are more informative. *(Ch 2)*

**TF-IDF** — a classic score combining TF and IDF to weight terms by importance. *(Ch 2)*

**BM25** — a refined, widely-used ranking function that improves on TF-IDF with length normalization and saturation. *(Ch 2)*

---

## Vectors & embeddings

**Vector** — an ordered list of numbers; here, a point in a high-dimensional space. *(Ch 3)*

**One-hot encoding** — representing a word as a vector that is all zeros except a single 1. Sparse and meaning-blind. *(Ch 3)*

**Bag-of-words** — representing a document by its word counts, ignoring order. *(Ch 3)*

**Dense vector** — a compact vector where every dimension carries information (vs. sparse). *(Ch 3)*

**Embedding** — a learned dense vector that captures the *meaning* of text, so similar meanings sit close together. *(Ch 3/4)*

**Embedding model** — a neural network that maps text to an embedding. *(Ch 4)*

**Dimensionality** — the number of components in a vector (e.g., 384, 768, 1536). *(Ch 4)*

**Cosine similarity** — a measure of closeness based on the angle between two vectors; 1 = same direction, 0 = unrelated. *(Ch 3)*

**Dot product** — multiply-and-sum of two vectors; relates to similarity (equals cosine when vectors are normalized). *(Ch 3)*

**Normalization** — scaling a vector to unit length so only its direction matters. *(Ch 3)*

**word2vec / GloVe** — early models that learn one fixed vector per word. *(Ch 4)*

**Contextual embeddings** — vectors that depend on surrounding words (e.g., BERT), so "bank" differs by context. *(Ch 4)*

---

## Semantic retrieval

**Semantic search** — finding documents by meaning rather than exact words. *(Ch 5)*

**Dense retrieval** — semantic search implemented by comparing dense embeddings. *(Ch 5)*

**Bi-encoder** — architecture that embeds query and documents separately, then compares; fast and scalable. *(Ch 5)*

**Nearest neighbor search** — finding the corpus vectors closest to the query vector. *(Ch 5)*

**Approximate Nearest Neighbor (ANN)** — algorithms that find *almost* the closest vectors far faster than exact search. *(Ch 6)*

**Vector database** — a system that stores embeddings and serves fast similarity search, with filtering and persistence. *(Ch 6)*

**FAISS** — a popular open-source library for efficient similarity search over vectors. *(Ch 6)*

**HNSW** — Hierarchical Navigable Small World; a graph-based ANN index, very common. *(Ch 6)*

**IVF (inverted file index)** — an ANN technique that partitions vectors into clusters to prune the search. *(Ch 6)*

**Product quantization (PQ)** — compressing vectors to save memory while keeping approximate distances. *(Ch 6)*

---

## Improving results

**Re-ranking** — a second pass that re-scores top candidates more precisely. *(Ch 7)*

**Cross-encoder** — a model that reads query and document *together* for an accurate relevance score; slow but precise. *(Ch 7)*

**Two-stage retrieval** — retrieve many candidates cheaply, then re-rank a few expensively. *(Ch 7)*

**Hybrid search** — combining lexical (BM25) and semantic (dense) retrieval. *(Ch 8)*

**Reciprocal Rank Fusion (RRF)** — a simple, robust way to merge multiple ranked lists. *(Ch 8)*

---

## Evaluation

**Ground truth / relevance judgments** — human-labeled "correct" results used to score a system. *(Ch 9)*

**Precision@k** — fraction of the top-k results that are relevant. *(Ch 9)*

**Recall@k** — fraction of all relevant documents found within the top-k. *(Ch 9)*

**Mean Reciprocal Rank (MRR)** — average of 1/(rank of first relevant result). *(Ch 9)*

**nDCG** — Normalized Discounted Cumulative Gain; rewards putting highly-relevant results near the top. *(Ch 9)*

**MAP** — Mean Average Precision across queries. *(Ch 9)*

---

## Generation & RAG

**Large Language Model (LLM)** — a neural network trained to predict and generate text. *(Ch 0/10)*

**Hallucination** — when an LLM produces fluent but false or unsupported content. *(Ch 10)*

**Retrieval-Augmented Generation (RAG)** — retrieving relevant context and having the LLM answer using it, for grounded answers. *(Ch 10)*

**Grounding** — tying an answer to retrieved source text so it's verifiable. *(Ch 10)*

**Context window** — the amount of text an LLM can consider at once. *(Ch 10)*

**Chunking** — splitting documents into smaller pieces suitable for embedding and retrieval. *(Ch 11)*

**Chunk overlap** — repeating some text between adjacent chunks so context isn't cut off. *(Ch 11)*

**Ingestion pipeline** — the load -> clean -> chunk -> embed -> index process. *(Ch 11)*

---

## Advanced

**Query expansion / rewriting** — improving a query by adding terms or rephrasing it. *(Ch 12)*

**HyDE (Hypothetical Document Embeddings)** — generate a hypothetical answer, embed *that* to retrieve. *(Ch 12)*

**ColBERT / late interaction** — multi-vector retrieval that compares token-level embeddings. *(Ch 12)*

**Multimodal search** — searching across modalities, e.g., text queries over images. *(Ch 12)*

**Fine-tuning** — further training an embedding model on your own labeled data. *(Ch 12)*

**Agentic RAG** — an LLM agent that plans multiple retrieval/reasoning steps. *(Ch 12)*