RAG and Agentic Search

Retrieval-Augmented Generation from scratch

Lecture 12 · Running example: `report.md` · Code: `examples/001` – `005`
Adapted from Anthropic Academy, *Building with the Claude API* — "RAG and Agentic Search"

Learning Objectives

By the end of this lecture you should be able to:

1. Explain what RAG solves that "put the whole document in the prompt" cannot — and **when it's worth it**
2. Implement **size-, sentence-, and structure-based chunking** and choose among them
3. Explain **text embeddings**, why dimensions aren't interpretable, and **asymmetric retrieval**
4. Trace the **six-step RAG flow** and compute **cosine similarity / distance** by hand
5. Build a brute-force **vector index** — and say when it stops scaling (**IVF, HNSW**, the **post-filter trap**)
6. Compute a **BM25** score from its parts and say when lexical beats semantic search
7. Fuse rankings with **Reciprocal Rank Fusion**, worked by hand
8. Contrast the fixed pipeline with **agentic search** (retrieval as a tool)

An 800-Page Document and a Short Question

You're building an assistant over an 800-page financial filing:

> *"What risk factors does this company disclose?"*

The answer is in there somewhere — but Claude has never seen this document. An LLM predicts *plausible* tokens; it isn't looking anything up.

> **You** have to put the relevant text in front of the model. How?

Option 1: Put Everything in the Prompt

```
Answer the user's question
about the financial document.

<user_question>
{user_question}
</user_question>

<financial_document>
{financial_document}   <- 800 pages
</financial_document>
```

Simplest thing that works — for small documents.
Then it breaks:

- **Hard limit** — the context window is finite
- **Quality** — reasoning degrades when the answer is one paragraph in hundreds of pages
- **Cost** — billed per input token: *every* question pays for all 800 pages again
- **Latency** — long prompts are slow, and the user is waiting

Option 2: Retrieve First, Then Prompt

Ahead of time, cut the document into **chunks**. When a question arrives, search for the few relevant chunks and put **only those** in the prompt.

Ask about risks → search finds "Risk Factors" → Claude reads a page, not a book.

> **Retrieval-Augmented Generation (RAG)** — two phases.
> **Preprocessing** (offline, once per document set): chunk the documents and build a searchable index.
> **Query time** (once per question): retrieve the most relevant chunks; put only those in the prompt.

Seen it before: in *Relational Databases* you built an **index once** and queried it instead of scanning the table. In *Elephant-Goldfish* the context window was the scarce resource — RAG **selects** what goes in.

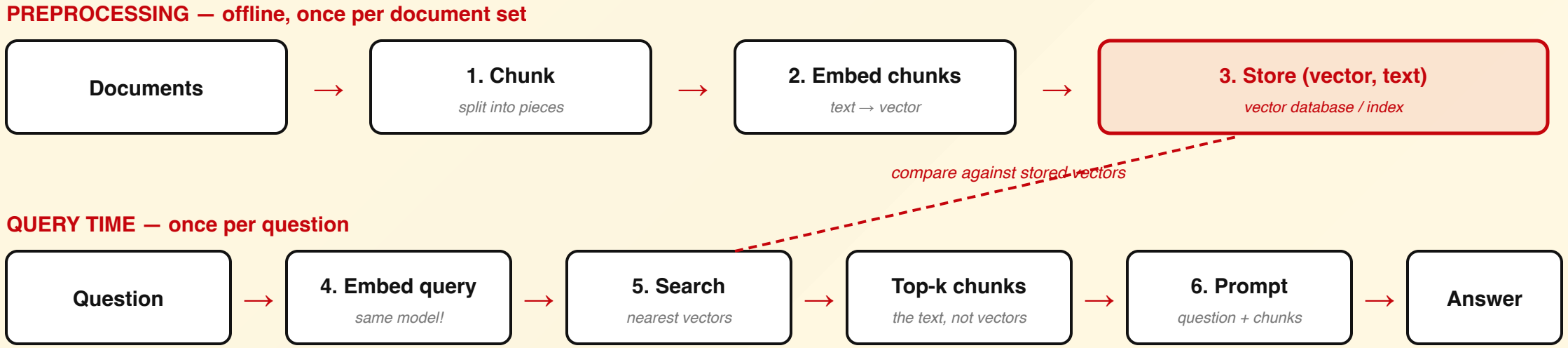
The Librarian and the Expert

Role	Who	Strength	Weakness
Librarian	the retrieval system	fast, has seen every page	doesn't deeply understand the question
Expert	Claude	understands the question deeply	only reads what it's handed

> The expert is never better than the stack the librarian hands over — which is why **most of today is about the librarian.**

In plain English: instead of handing Claude a haystack and asking for the needle, a cheaper, faster search finds the needle first.

The RAG Pipeline



Separately tunable stages — the system is only as good as its weakest one.

Misconceptions, and When to Use RAG

- *"RAG is a search box bolted onto an LLM."* — It's a **pipeline** of tunable stages. A great embedding model can't rescue bad chunks.
- *"More retrieved context is always better."* — Irrelevant chunks cost tokens, distract, and can cause a **confidently wrong** answer.
- *"RAG teaches the model the documents."* — No weights change. That's a **feature**: edit a doc and the next question sees the edit.

Project	Decision	Why
Q&A over an 18 KB `report.md`	no RAG	fits easily; stuffing the prompt is simpler
Q&A over one 800-page filing	**RAG**	too big, too slow, too costly per question
Search across 5,000 support tickets	**RAG**	many documents; must search across all

We build RAG over report.md anyway: small enough to inspect every chunk by eye. The techniques transfer unchanged.

Running Example: `report.md`

A fictional 18 KB "Annual Interdisciplinary Research Review" — every example in `examples/` uses it.

```
# Annual Interdisciplinary Research Review: Cross-Domain Insights
## Executive Summary
## Table of Contents
## Methodology
## Section 1: Medical Research - Understanding XDR-471 Syndrome
## Section 2: Software Engineering - Project Phoenix Stability Enhancements
## Section 3: Financial Analysis - Q3 Performance and Outlook
   ... Sections 4–9: materials, legal, product, history, project mgmt, pharma ...
## Section 10: Cybersecurity Analysis - Incident Response Report: INC-2023-Q4-011
## Future Directions
```

Two deliberate traps.
Words that cross topics — medical calls a disease *"a 'bug' we have not seen before"*; software studies *"infection vectors in our distributed systems."*
An exact ID in two places — `INC-2023-Q4-011` appears only in Section 2 and Section 10.

Chunking: The Highest-Leverage Decision

> A **chunk** is a contiguous span of a document, small enough to embed and retrieve on its own, ideally **one coherent idea**. A **chunking strategy** decides where one ends and the next begins.

Every later stage works on whatever pieces chunking produced. **Context destroyed at a chunk boundary can't be recovered downstream.**

The "bug" problem:

1. A careless chunker cuts *"XDR-471, a 'bug' we have not seen before"* away from its medical context
2. User asks: *"How many bugs did engineers fix this year?"*
3. The isolated chunk mentions a bug → retrieved → Claude answers a software question with medical research

Strategy 1: Size-Based Chunking

Slide a fixed-size window across the text; step back by `chunk_overlap` so text cut at a boundary reappears in the next chunk.

```python
def chunk_by_char(text: str, chunk_size: int = 150, chunk_overlap: int = 20) -> list[str]:
    if chunk_overlap >= chunk_size:          # otherwise the window never advances
        raise ValueError("chunk_overlap must be smaller than chunk_size")
    chunks, start_idx = [], 0
    while start_idx < len(text):
        end_idx = min(start_idx + chunk_size, len(text))
        chunks.append(text[start_idx:end_idx])
        # Step back by the overlap -- unless we just consumed the end of the text.
        start_idx = end_idx - chunk_overlap if end_idx < len(text) else len(text)
    return chunks
```

Trace: `"abcdefghijklmnopqrstuvwxyz"` (26 chars), `chunk_size=10`, `chunk_overlap=3`

```
iter  start  end  chunk          next start
  1     0    10   "abcdefghij"   10 - 3 = 7
  2     7    17   "hijklmnopq"   17 - 3 = 14
  3    14    24   "opqrstuvwx"   24 - 3 = 21
  4    21    26   "vwxyz"        end reached -> stop
```

Size-Based: Strengths and Limits

Each chunk begins with the last 3 characters of the previous one (`hij`, `opq`, `vwx`) — that's the overlap.

> **Misconception:** *"Overlap fixes mid-sentence cuts."* It doesn't. Any chunk can still start or end mid-word. Overlap only guarantees the cut-off text also shows up in a neighbor, so nothing vanishes completely.

- On `report.md` (150 / 20): **141 chunks**, most cut mid-sentence, headings split from their content
- **But** it assumes nothing about structure — Markdown, plain text, logs, even code

> **The most common production default.** Never great; never fails outright.

Strategy 2: Sentence-Based Chunking

Split into sentences, group N consecutive sentences per chunk, with sentence-level overlap.

```python
import re

def chunk_by_sentence(text, max_sentences_per_chunk=5, overlap_sentences=1):
    sentences = re.split(r"(?<=[.!?])\s+", text.strip())
    chunks, start_idx = [], 0
    while start_idx < len(sentences):
        end_idx = min(start_idx + max_sentences_per_chunk, len(sentences))
        chunks.append(" ".join(sentences[start_idx:end_idx]))
        start_idx += max_sentences_per_chunk - overlap_sentences
    return chunks
```

`(?<=[.!?])` is a **lookbehind**: it checks for the punctuation without consuming it, so the period stays with its sentence. (A heuristic: "Dr. Smith" and "v2.1" fool it, but a slightly wrong cut is cheap.)

```
sentences:  s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10
chunk 1:   [s0 s1 s2 s3 s4]
chunk 2:               [s4 s5 s6 s7 s8]          <- s4 repeated (overlap)
chunk 3:                           [s8 s9 s10]   <- s8 repeated
```

Strategies 3 & 4: Structure and Semantics

Structure-based: split at the document's own markers — for Markdown, headers.

```python
def chunk_by_section(document_text: str) -> list[str]:
    return re.split(r"\n## ", document_text)
```

- `report.md` has 14 `##` headers → **15 chunks** (title line + one per section)
- Boundaries line up with topics: the medical "bug" **never shares a chunk** with software text
- `re.split` consumes `##`, but the header line stays in the chunk — which helps retrieval
- **Risk:** a scanned PDF or `.txt` with no `##` silently returns **one giant chunk**; sizes vary wildly (70–2,185 chars)

Semantic-based: embed each sentence; start a new chunk where similarity to the next sentence drops. Most coherent, but a model call per sentence (Exercise A1).

Comparing the Strategies on `report.md` DEMO

```
cd examples/001_chunking && python chunking.py      # stdlib only
```

```
--- 1. Size-based: chunk_by_char ---          chunk count: 141   avg 149.7   min/max 105 / 150
--- 2. Sentence-based: chunk_by_sentence ---  chunk count: 33    avg 701.3   min/max 237 / 1023
--- 3. Structure-based: chunk_by_section ---  chunk count: 15    avg 1216.6  min/max 70 / 2185
```

Strategy	Needs structure?	Splits sentences?	Cost	Best for
size-based	no	yes	very low	universal fallback; code; unstructured text
sentence-based	no	no	low	general prose without reliable formatting
structure-based	**yes**	no	low	reports and docs whose format you control
semantic-based	no	no	**high**	when coherence matters more than cost

No universally best strategy. From here on: `chunk_by_section` — `report.md` is well-structured, and 15 chunks are easy to inspect.

Text Embeddings

> A **text embedding** is a fixed-length vector of numbers (hundreds to thousands of dimensions) produced by an **embedding model**, such that **texts with similar meanings map to nearby vectors**.

The model turns *meaning* into *geometry*: "how close are these two lists of numbers?" approximates "how similar are these two texts?" This is **semantic search**.

- Tempting: dimension 17 = "about oceans," dimension 340 = "sounds happy"
- Reality: each dimension's meaning is learned in training and is **not human-interpretable**
- Embeddings approximate meaning statistically. An arbitrary ID like `INC-2023-Q4-011` has **little meaning to encode** — remember this one

Queries ≠ Documents (Asymmetric Retrieval)

A question (*"What risks does this company face?"*) looks nothing like its answer (*"Risk Factors: our primary exposure is..."*). Models let you say which **role** a text plays.

We run **BAAI/bge-m3** locally (free, offline, 1,024-dim) instead of the course's hosted VoyageAI:

```
QUERY_INSTRUCTION = "Represent this sentence for searching relevant passages: "

def generate_embedding(text, input_type="query"):
    """Same signature as the VoyageAI version -- nothing downstream changes."""
    is_list = isinstance(text, list)
    inputs = text if is_list else [text]
    if input_type == "query":                       # only queries get the prefix
        inputs = [QUERY_INSTRUCTION + t for t in inputs]
    output = _get_model().encode(inputs, batch_size=12, max_length=8192,
                                 return_dense=True)  # lazy-loaded BGEM3FlagModel
    embeddings = output["dense_vecs"].tolist()       # numpy -> plain lists
    return embeddings if is_list else embeddings[0]
```

VoyageAI: account + API key, billed per token, data leaves your machine. BGE-M3: free after a one-time ~2.3 GB download, fully local. Same signature, so nothing downstream changes (Layered Architecture: depend on the interface).

The Full Flow by Hand: A "Perfect" 2-D Model

Imagine a model that returns exactly **two** numbers — and unlike any real model, we know what they mean: **dim 1 = medical**, **dim 2 = software engineering**.

Step 1 — chunk. Two chunks from the report:

- **Medical:** "...our understanding of XDR-471, a 'bug' we have not seen before."
- **Software:** "...studying various infection vectors in our distributed systems."

Step 2 — embed, then normalize to length 1 (so text length doesn't matter):

```
medical  -> [0.97, 0.34]   |v| = sqrt(0.97² + 0.34²) = 1.0279  ->  [0.944, 0.331]
software -> [0.30, 0.97]   |v| = sqrt(0.30² + 0.97²) = 1.0153  ->  [0.295, 0.955]
```

Step 3 — store both vectors **together with their chunk text** in a **vector database**.

> Preprocessing ends here. Everything so far happens before any user shows up.

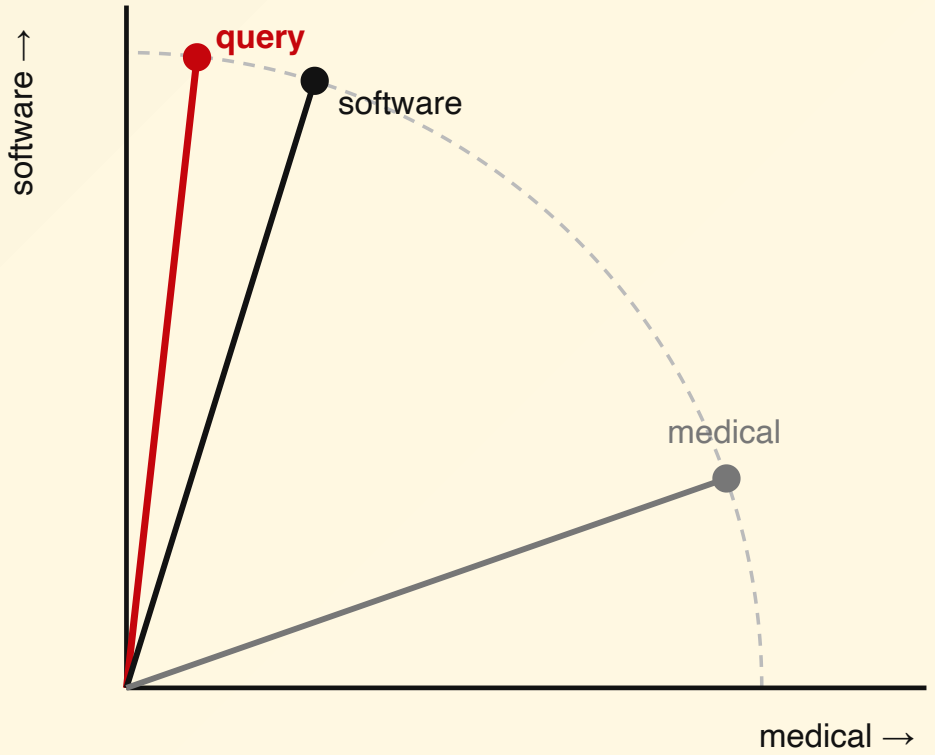
Step 4: Embed the Question

"I'm curious about the company. In particular, what did the software engineering dept do this year?"

```
query -> [0.10, 0.89]
|q| = 0.8956
    -> [0.112, 0.994]
```

All three normalized vectors lie on the **unit circle**.

Query and software chunk point in **nearly the same direction**; medical points elsewhere. "Nearest" needs a precise definition.



Step 5: Cosine Similarity and Distance

```
cosine_similarity(a, b) = (a · b) / (|a| |b|)          range [-1, 1]   higher = closer
cosine_distance(a, b)   = 1 - cosine_similarity(a, b)  range [0, 2]    lower  = closer
```

Measures the **angle**, ignores length: **1** = same direction, **0** = unrelated (perpendicular, *not* opposite), **−1** = opposite. Most vector databases report **distance**.

Worked example — unit vectors, so similarity = dot product:

```
query · software = 0.112×0.295 + 0.994×0.955 = 0.033 + 0.949 = 0.982
query · medical  = 0.112×0.944 + 0.994×0.331 = 0.106 + 0.329 = 0.434

                 similarity   distance
software chunk     0.982        0.018     <- nearest: retrieved
medical chunk      0.434        0.566
```

The course video quotes 0.398 for medical — the dot product of the unnormalized vectors. True cosine is 0.434; the ranking is the same either way.

Step 6: Build the Prompt

Only after retrieval do we write the prompt:

```
Answer the user's question about the financial document.

<user_question>
How many bugs did engineers fix this year?
</user_question>

<report>
## Section 2: Software Engineering
This division dedicated significant effort to studying various
infection vectors in our distributed systems
</report>
```

> Claude never sees the medical section. That's the whole point of RAG — and its risk: **the answer can be no better than the chunks retrieval chose.**

The Six Steps at a Glance

Step	Phase	What happens	In the example
1 chunk	preprocessing	split documents	medical, software chunks
2 embed	preprocessing	chunk → vector (normalized)	`[0.944,0.331]`, `[0.295,0.955]`
3 store	preprocessing	save **(vector, text)** pairs	vector database
4 embed query	query time	question → vector, **same model**	`[0.112,0.994]`
5 search	query time	rank stored vectors by similarity	software 0.982 > medical 0.434
6 prompt	query time	question + top chunks → Claude	Section 2 only

Misconceptions: *similarity 0 = opposite* (no — unrelated) · *real models separate topics this cleanly* (no — our 0.98 vs 0.43 gap is a 2-D artifact) · *the query can use a different model* (no — different coordinates, distances become noise).

Implementing It: a Minimal `VectorIndex`

```python
class VectorIndex:
    """A tiny in-memory vector store with brute-force nearest-neighbor search."""
    def __init__(self, embedding_fn):
        self.vectors, self.documents = [], []          # parallel lists
        self._embedding_fn = embedding_fn

    def add_documents(self, documents):                # ONE batched embedding call
        vectors = self._embedding_fn([d["content"] for d in documents],
                                     input_type="document")
        for vector, document in zip(vectors, documents):
            self.vectors.append(vector)
            self.documents.append(document)            # keep the TEXT next to the vector

    def search(self, query, k=1):                      # -> k (document, distance) pairs
        query_vector = self._embedding_fn(query, input_type="query")
        distances = [(self._cosine_distance(query_vector, v), self.documents[i])
                     for i, v in enumerate(self.vectors)]
        distances.sort(key=lambda item: item[0])       # smallest distance first
        return [(doc, dist) for dist, doc in distances[:k]]

    @staticmethod
    def _cosine_distance(v1, v2):
        mag1, mag2 = math.sqrt(sum(x*x for x in v1)), math.sqrt(sum(x*x for x in v2))
        sim = sum(p*q for p, q in zip(v1, v2)) / (mag1 * mag2)
        return 1.0 - max(-1.0, min(1.0, sim))          # clamp float round-off
```

Condensed from examples/003_vectordb/vectordb.py (adds validation, dimension checks, Euclidean distance).

`VectorIndex`: Design Decisions

- **Store the text next to the vector.** Search returns numbers, but Step 6 needs the chunk text. Forgetting this is **the classic first RAG bug**.
- **Batch the embedding call.** One call for all chunks — avoids hosted-API rate limits; a local model runs a batch far faster than a loop.
- **Query vs. document.** `add_documents` uses `input_type="document"`, `search` uses `"query"` — asymmetric retrieval, applied automatically.
- **The clamp.** Round-off can produce similarity 1.0000000002; clamping keeps distance from going negative.
- **Provider-agnostic.** It only sees lists of floats: swap BGE-M3 for VoyageAI by passing a different `embedding_fn`. No other line changes.

The Five Steps on `report.md` DEMO

```
text = Path("report.md").read_text()

chunks = chunk_by_section(text)                              # 1. chunk

store = VectorIndex(embedding_fn=generate_embedding)         # 2-3. embed + store
store.add_documents([{"content": chunk} for chunk in chunks])

question = "What did the software engineering department do last year?"
for doc, distance in store.search(question, k=2):             # 4-5. embed query + search
    print(f"cosine distance={distance:.4f}  {doc['content'].splitlines()[0][:70]}")
```

```
cd examples/003_vectordb && pip install -r requirements.txt && python vectordb.py
```

> The right answer wins — by **0.01**, not the toy model's 0.02 vs 0.57. Real embeddings are a **crowded continuum**. BGE-M3 gives different numbers; Section 2 should still rank first.

Closing the Loop: Prompting Claude

```
PROMPT_TEMPLATE = """\
Answer the user's question about the report, using only the excerpts below.
If the excerpts do not contain the answer, say so instead of guessing.

<user_question>
{question}
</user_question>

<report_excerpts>
{excerpts}
</report_excerpts>"""

def build_prompt(question, results):
    """Wrap each retrieved chunk in its own tag, in retrieval order."""
    excerpts = "\n".join(
        f"<excerpt>\n{doc['content'].strip()}\n</excerpt>" for doc, _ in results)
    return PROMPT_TEMPLATE.format(question=question, excerpts=excerpts)
```

- **XML tags** separate the question from the excerpts — the model can't confuse the two
- **The refusal instruction** gives a safe exit: a retrieval miss becomes a visible *"I don't know"*, not a confident invention
- **Debug tip:** print the prompt instead of sending it — read exactly what Claude would read

From a List to a Vector Database

Brute force costs one similarity per vector: **O(n·d)**.

Corpus	n × d	Per question
`report.md`	15 × 1,024 ≈ 15K multiply-adds	microseconds
every company report, chunked	20M × 1,024 ≈ 20 **billion**	seconds of CPU, every search

Approximate nearest neighbors (ANN): return *almost always* the true top-k, in a fraction of the time. Quality = **recall** (fraction of the true top-k returned). OK because the embedding is already an approximation.

- **IVF** — k-means into ~1,000 clusters; at query time brute-force only the nearest `n_probe` clusters (~1% of data). Misses: a neighbor just across an unprobed boundary.
- **HNSW** — layered neighbor graph, like express and local trains; greedy hops, drop a layer when stuck. ~log-time queries; costs RAM and slower inserts. **Default in most vector DBs.**

> ANN makes search **faster, never more accurate**. Bad results on a small corpus? Fix chunking or embeddings, not the index.

Where the Vectors Live — and the Post-Filter Trap

Option	Examples	Reach for it when
a list + a loop	`VectorIndex`	under ~100K vectors; prototypes
embedded library	FAISS, Chroma, sqlite-vec	one app, local files, no server
database extension	pgvector (PostgreSQL)	vectors next to your existing rows
dedicated service	Qdrant, Weaviate, Milvus, Pinecone	many clients, billions of vectors

*"Cybersecurity risks, **2023 report only**?"* → store **metadata** (year, section) next to content.

- **Pre-filter:** predicate first, then rank survivors. Always correct.
- **Post-filter:** top-k first, then drop failures. Fast — and **silently wrong**: if all 5 nearest are from 2022, the result is **empty**.

> Classic bug report: *"search works, but returns nothing when I filter."* Also: **switching embedding models = re-embed everything** (store the model name; plan it like a migration), and edited documents leave **stale vectors** until re-embedded.

What Still Goes Wrong

Ask the vector index: ***"What happened with INC-2023-Q4-011?"***
The ID appears in **Section 2** (Software) and **Section 10** (Cybersecurity).

Rank	Chunk (course's semantic-search run)	Contains the ID?
1	Section 10: Cybersecurity	✅
2	**Section 3: Financial Analysis**	❌ never mentions it
↓	Section 2: Software Engineering	✅ pushed down

To an embedding model `INC-2023-Q4-011` is an arbitrary string. It encodes the question's **topic** ("incidents, problems, impact") — and Section 3's quarterly risks match that topic about as well as Section 2.

> **Semantic search is weak on exact terms** — IDs, error codes, product numbers, function names. Users type those all the time.

BM25 Lexical Search

> **Lexical search** ranks documents by the words they **literally share** with the query. **BM25** ("Best Match 25") is the standard lexical ranking function.

The course's four steps:

1. **Tokenize** the query into terms
2. **Count** how often each term appears across documents — "the" is everywhere; `INC-2023-Q4-011` is in two places
3. **Weight terms by rarity (IDF)** — this is what lets an exact ID dominate
4. **Score and rank** documents by the high-weight terms they contain

> Semantic: *"what does this mean?"* · Lexical: *"which chunks contain these exact words, and how rare are they?"*

The BM25 Formula and Its Three Knobs

$$\text{score}(q,d) = \sum_{t \in q} \frac{\text{IDF}(t) \cdot f(t,d) \cdot (k1 + 1)}{f(t,d) + k1 \cdot (1 - b + b \cdot |d| / \text{avgdl})}$$

$$\text{IDF}(t) = \ln\left((N - n(t) + 0.5) / (n(t) + 0.5) + 1 \right)$$

$f(t,d)$ = count of t in d · $|d|$ = doc length · avgdl = average length · N = #docs · $n(t)$ = #docs containing t · defaults $k1 = 1.5$, $b = 0.75$

Knob	Idea	Effect
IDF	rarity is information	a term in every doc tells you nothing; a term in one tells you nearly everything
k1	diminishing returns	1st occurrence adds a lot, 10th adds little; saturates at `k1 + 1`
b	length fairness	long docs contain more of everything; `b = 0.75` discounts them, `b = 0` turns it off

Small k1 ≈ "contains it or not"; large k1 keeps rewarding repetition. The +0.5 smoothing keeps IDF finite and non-negative.

BM25 by Hand (1/2)

```
d0: "Engineers fixed the bug behind incident INC-2023-Q4-011."           10 tokens
d1: "The incident response team contained INC-2023-Q4-011 within hours." 11 tokens
d2: "Revenue grew this quarter and the team hired engineers."             9 tokens
                                                  N = 3,  avgdl = 10.0
```

Step 1: tokenize. Lowercase, split on non-word characters — the hyphens split the ID:

```
"What happened with INC-2023-Q4-011?"
   -> ["what", "happened", "with", "inc", "2023", "q4", "011"]
```

`what`, `happened`, `with` appear in no document → contribute nothing.

Steps 2–3: document frequency → IDF

```
term            n(t)  IDF = ln((3 - n + 0.5)/(n + 0.5) + 1)
the              3    ln(0.5/3.5 + 1) = 0.134    <- everywhere: nearly worthless
inc,2023,q4,011  2    ln(1.5/2.5 + 1) = 0.470 each
bug              1    ln(2.5/1.5 + 1) = 0.981    <- rare: valuable
```

BM25 by Hand (2/2)

Step 4: score. Each of the 4 ID tokens appears once (`f = 1`) in `d0` and `d1`:

```
d0 (|d| = 10 = avgdl):  denom = 1 + 1.5·(1 − 0.75 + 0.75·10/10) = 2.500
   per term = 0.470 · 1 · 2.5 / 2.500  = 0.470     × 4 terms = 1.880

d1 (|d| = 11 > avgdl):  denom = 1 + 1.5·(1 − 0.75 + 0.75·11/10) = 2.6125
   per term = 0.470 · 1 · 2.5 / 2.6125 = 0.450     × 4 terms = 1.799

d2: contains none of the query terms                           = 0.000
```

- Both documents containing the ID score high; `d2` scores zero
- `d0` edges out `d1` only because it's **one token shorter** — the `b` parameter at work
- Exactly the numbers `BM25Index` in `examples/004_bm25/bm25.py` computes

The `BM25Index` Class (Core)

```python
class BM25Index:
    """From-scratch Okapi BM25 with the same API as VectorIndex. Stdlib only."""

    def add_document(self, document):
        tokens = self._tokenizer(document["content"])
        self.documents.append(document); self._corpus_tokens.append(tokens)
        self._doc_len.append(len(tokens))
        for token in set(tokens):                    # count each doc once per term
            self._doc_freqs[token] = self._doc_freqs.get(token, 0) + 1
        self._index_built = False                    # IDF and avgdl are now stale

    def search(self, query_text, k=1, score_normalization_factor=0.1):
        if not self._index_built:
            self._build_index()                      # lazy: one rebuild, not one per add
        query_tokens = self._tokenizer(query_text)
        raw = [(self._compute_bm25_score(query_tokens, i), doc)
               for i, doc in enumerate(self.documents)]
        raw = [(s, d) for s, d in raw if s > 1e-9]   # drop non-matches
        raw.sort(key=lambda item: item[0], reverse=True)   # higher raw = better
        # Flip to VectorIndex's "smaller is better" convention, keeping the order.
        return [(doc, math.exp(-score_normalization_factor * s)) for s, doc in raw[:k]]
```

Watch the score convention. Raw BM25: *higher = better*. Cosine distance: *lower = better*. `exp(-0.1 · raw)` flips direction, keeps order (raw 1.880 → 0.829). Mixing them up is **the most common bug in student hybrid retrievers.** The tokenizer is pluggable (Exercise I3).

BM25 on `report.md` DEMO

```
cd examples/004_bm25 && python bm25.py        # stdlib only: no model, no key
```

```
1. Chunked report.md into 15 sections.
2. Indexed 15 chunks. BM25Index(count=15, k1=1.5, b=0.75, index_built=False)

3. Top 3 BM25 results for: 'What happened with INC-2023-Q4-011?'

  score=0.2713  section='Section 2: Software Engineering - Project Phoenix Stability ...'
  score=0.3317  section='Section 10: Cybersecurity Analysis - Incident Response Report: ...'
  score=0.9391  section='Methodology'
```

Lower is better. **Both sections that literally contain the ID take the top two places** — which semantic search failed to do. Methodology is a distant third via `what`, `with`.

Same idea as Claude Code grepping for a function name: when you know the exact string, exact matching beats meaning.

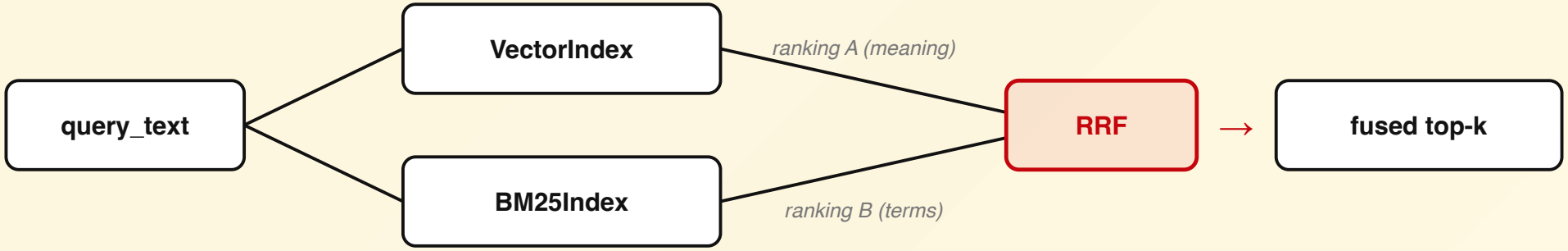
Neither Search Is Enough Alone

Query	Semantic search	BM25
What happened with INC-2023-Q4-011?	topic only; ranks Section 3 too high	✅ finds both sections with the ID
How did the developers improve uptime?	✅ finds Section 2 by meaning	❌ no shared words with "stability enhancements"
errors in the payment system	✅ good	✅ good if the words appear

- *"BM25 understands the query."* — No notion of meaning, synonyms, or paraphrase.
- *"BM25 is obsolete."* — Still the default ranking in Elasticsearch: exact rare-term matching is a real, different requirement.

> **The obvious next step: run both.**

Hybrid Search: One Interface, Many Indexes



Hybrid search sends each query to several indexes and merges their rankings; the **retriever** does the forwarding and merging. `VectorIndex` and `BM25Index` share nothing internally — but they have the **same shape**:

```
class SearchIndex(Protocol):
    def add_document(self, document: Dict[str, Any]) -> None: ...
    def add_documents(self, documents: List[Dict[str, Any]]) -> None: ...
    def search(self, query: Any, k: int = 1) -> List[Tuple[Dict[str, Any], float]]: ...
```

Merging Rankings: Reciprocal Rank Fusion

A cosine distance of 0.3 and a BM25 score of 0.3 live on unrelated scales — you can't average them. **RRF ignores the scores and uses only rank order:**

```
RRF_score(d) = Σ over indexes i of   1 / (k_rrf + rank_i(d))      rank 1 = best; absent = 0
```

Worked example (searching `INC-2023-Q4-011`, `k_rrf = 1` so the arithmetic is easy):

```
VectorIndex:  Section 2 (1), Section 7 (2), Section 6 (3)
BM25Index:    Section 6 (1), Section 2 (2), Section 7 (3)
```

Chunk	Vector rank	BM25 rank	RRF score
Section 2	1	2	1/2 + 1/3 = **0.833**
Section 6	3	1	1/4 + 1/2 = 0.750
Section 7	2	3	1/3 + 1/4 = 0.583

> Section 2 wins **without being first in BM25** — it ranks well in *both*.

What `k_rrf` Does

The same ranks with the production default `k_rrf = 60`:

```
             k_rrf = 1     k_rrf = 60
Section 2      0.833         0.03252
Section 6      0.750         0.03227
Section 7      0.583         0.03200
```

Same order, much flatter.

- **Small `k_rrf`:** one first-place finish counts heavily (1/2 vs 1/4 for third)
- **Large `k_rrf`:** every position counts almost equally (1/61 vs 1/63), so **agreement between indexes** matters more than any one index's top pick

> That makes **60** a robust default.

The `Retriever` Class

```python
class Retriever:
    """Fuses any number of SearchIndex rankings with Reciprocal Rank Fusion."""

    def __init__(self, *indexes: SearchIndex):
        self._indexes = list(indexes)

    def add_documents(self, documents):
        for index in self._indexes:
            index.add_documents(documents)          # the SAME dicts go to every index

    def search(self, query_text, k=1, k_rrf=60):
        # Over-fetch (k * 5) from each index so fusion has candidates to work with.
        all_results = [index.search(query_text, k=k * 5) for index in self._indexes]

        doc_ranks = {}                              # id(doc) -> doc + rank in each index
        for idx, results in enumerate(all_results):
            for rank, (doc, _score) in enumerate(results):
                entry = doc_ranks.setdefault(
                    id(doc), {"doc_obj": doc, "ranks": [float("inf")] * len(self._indexes)})
                entry["ranks"][idx] = rank + 1      # ranks are 1-based

        def rrf(ranks):
            return sum(1.0 / (k_rrf + r) for r in ranks if r != float("inf"))

        fused = [(e["doc_obj"], rrf(e["ranks"])) for e in doc_ranks.values()]
        fused.sort(key=lambda item: item[1], reverse=True)   # RRF: HIGHER = better
        return fused[:k]
```

`Retriever` : Design Decisions

- **Identity, not content.** `id(doc)` matches the *same dict object* across indexes. Build each index from separately created dicts and one chunk silently becomes two.
- **Over-fetching.** With only `k` per index, a chunk ranked `k+1` by *both* (a strong consensus pick) could never surface. (When is 5× not enough? Exercise A2.)
- **Missing means zero.** Absent from one list → rank ∞ there → contributes nothing. Not penalized, just not helped.
- **Convention flip, again.** Fused score is *higher = better*, unlike the indexes it wraps.

> **Extensible = open/closed** (*Layered Architecture*). A third index — metadata filter, knowledge graph, a different chunking — means implementing three methods and passing it to `Retriever(...)`. The fusion code doesn't change.

Hybrid Search on `report.md` DEMO

```
chunks = chunk_by_section(Path("report.md").read_text())

retriever = Retriever(BM25Index(), VectorIndex(embedding_fn=generate_embedding))
retriever.add_documents([{"content": c} for c in chunks])    # same dicts -> both

for doc, score in retriever.search("What happened with INC-2023-Q4-011?", k=3):
    print(f"rrf score={score:.4f}  {doc['content'].splitlines()[0][:70]}")
```

```
cd examples/005_hybrid && python hybrid.py    # prints vector-only, then hybrid results
```

Course's hybrid ranking:

```
1. Section 10: Cybersecurity Analysis - Incident Response Report: INC-2023-Q4-011
2. Section 2:  Software Engineering - Project Phoenix Stability Enhancements
```

> Both sections with the ID are on top; **Section 3 (the semantic false positive) is gone.** Compare rankings, not numbers — BGE-M3's scores differ from VoyageAI's.

Hybrid Search: Misconceptions

- *"RRF averages the scores."* — It **never reads** the scores. Only positions matter — that's how it combines scores that can't be compared.
- *"Hybrid always beats either index alone."* — Usually, not always. A badly tuned index adds noise. **Measure on your own questions.**
- *"More indexes is always better."* — Each adds ingestion cost and query latency. Add one when it catches failures the others miss.

> **Why hybrid is the production default:** real users mix conceptual questions (*"how did engineering improve stability?"*) with exact lookups (*"INC-2023-Q4-011"*) and expect both to work. Two cheap searches still cost far less than one LLM call.

From Pipeline to Agentic Search

Everything so far is a **fixed pipeline**: it can't decide *not* to search ("hello" still retrieves), can't rephrase a bad query, can't search again when results are weak.

Agentic search hands the retriever to the model **as a tool**. It can turn *"that thing about the outage last quarter"* into `search("INC-2023-Q4-011")`, notice two divisions, and search again.

```
fixed pipeline:   question -> retrieve -> prompt -> Claude -> answer                (1 call)

agentic search:   question -> Claude --search("INC-2023-Q4-011")--> Retriever
                              ^                                        |
                              +-------------- results ----------------+
                              ... repeat until satisfied ...  -> answer         (N calls)
```

You've used one all semester: **Claude Code** — grep is lexical search, reading a README is coarse retrieval, and search → read → refine → search again is agentic retrieval over your repo.

> **Rule of thumb:** small corpus → put it in the prompt · large corpus, single-step questions → fixed pipeline · messy, multi-step → agentic search (trades latency and tokens for judgment).

Activity — `rag-from-scratch`

One runnable folder per step, all on the same `report.md`. Tests use fake embeddings: **offline, < 1 s**.

```
examples/
|- 001_chunking/    chunking.py      chunk_by_char / _sentence / _section    (stdlib)
|- 002_embeddings/  embeddings.py    generate_embedding (local BGE-M3)
|- 003_vectordb/    vectordb.py      VectorIndex + the five-step flow
|- 004_bm25/        bm25.py          BM25Index                               (stdlib)
+- 005_hybrid/      hybrid.py        SearchIndex, Retriever, RRF
```

```
cd 001_chunking && python chunking.py && cd ..     # no install needed
cd 004_bm25     && python bm25.py     && cd ..     # no install needed

pip install -r 005_hybrid/requirements.txt         # ~2.3 GB model on first run
cd 005_hybrid   && python hybrid.py   && cd ..

pip install pytest
for d in 00*/; do (cd "$d" && python -m pytest -q tests); done
```

Each folder is self-contained (own report.md, own copy of shared code) — a deliberate teaching trade-off. In your own project, factor it into a package (Exercise I5).

Practice Exercises

Basic

- **B1** Trace `chunk_by_char("The quick brown fox jumps", 10, 3)`
- **B2** Why `(?<=[.!?])\s+` and not `[.!?]\s+`?
- **B3** Cosine of `[0.6,0.8]` vs `[0.8,0.6]` and vs `[-0.6,-0.8]`
- **B4** Label code lines with RAG step + phase
- **B5** IDF as `n(t)` grows; score as `k1 → 0`

Intermediate

- **I1** Chunk by **words** with overlap; test no chunk starts mid-word
- **I2** Retrieve Section 9 first *without* "pharmaceutical" or "CTX-204b"
- **I3** Tokenizer that keeps `INC-2023-Q4-011` **whole**; redo the toy example
- **I4** `k_rrf` = 1, 10, 60 on the incident query — table + explanation
- **I5** Refactor the folders into one `rag/` package

Practice Exercises — Advanced

Advanced

- **A1 Semantic chunking.** Embed sentences; cut where consecutive similarity drops below a threshold. Compare with `chunk_by_section`.
- **A2 When over-fetching fails.** Build a corpus where `k * 5` hides the true best fused doc. Is any fixed multiplier enough?
- **A3 Agentic search.** Expose `Retriever.search` as a tool. Write five multi-search questions, e.g. *"Which divisions were affected by INC-2023-Q4-011, and what did each do about it?"* Compare with the fixed pipeline on quality, API calls, tokens.

Common Mistakes

- Structure-based chunking on documents **without that structure** → one giant chunk
- Storing vectors **without their chunk text**
- Queries and documents embedded with **different models**, or ignoring `input_type`
- Switching models **without re-embedding**
- **Post-filtering** top-k → "filtered search returns nothing"
- Confusing **similarity (higher = better)** with **distance (lower = better)**, or raw vs normalized BM25
- **Averaging** BM25 scores and cosine distances instead of fusing ranks
- Building indexes from **different dict objects** → broken `id(doc)` fusion
- Semantic search alone when users type **IDs and error codes**
- No **refusal instruction** → a miss becomes a confident invention

Connections: context window as scarce resource (*LLM*, *Elephant-Goldfish*) · index once, query many (*Relational Databases*) · `embedding_fn` / `SearchIndex` as interfaces (*Layered Architecture*) · **next:** tool use, MCP, agents; evaluating RAG; retrieved text is **untrusted input** → prompt injection.

Summary

1. **RAG trades simplicity for scale.** If the document fits and cost doesn't matter, just put it in the prompt.
2. **Chunking sets the ceiling.** Size-based with overlap is the robust default; structure-based when you control the format.
3. **Embeddings turn meaning into geometry.** Same model for queries and documents; often different `input_type`s.
4. **Six steps:** chunk, embed, store (*with the text!*), embed the query, search by cosine distance, prompt with only the hits.
5. **Brute force is fine below ~100K vectors.** ANN (IVF, HNSW) is faster, never more accurate. Filter **before** ranking.
6. **Semantic misses exact terms; BM25 misses meaning.** IDF = rarity, `k1` = saturation, `b` = length.
7. **Hybrid fuses ranks, not scores.** RRF rewards agreement; a shared interface makes it extensible.
8. **Retrieval can become a tool.** Agentic search — which is how Claude Code already works.

Questions?

Before next lecture: run all five `examples/` folders and their tests, then do **I3** (keep IDs whole) and **I4** (tune `k_rrf`).

Next: **Prompt Engineering** — read the Anthropic system-prompts and XML-tags guides before class.