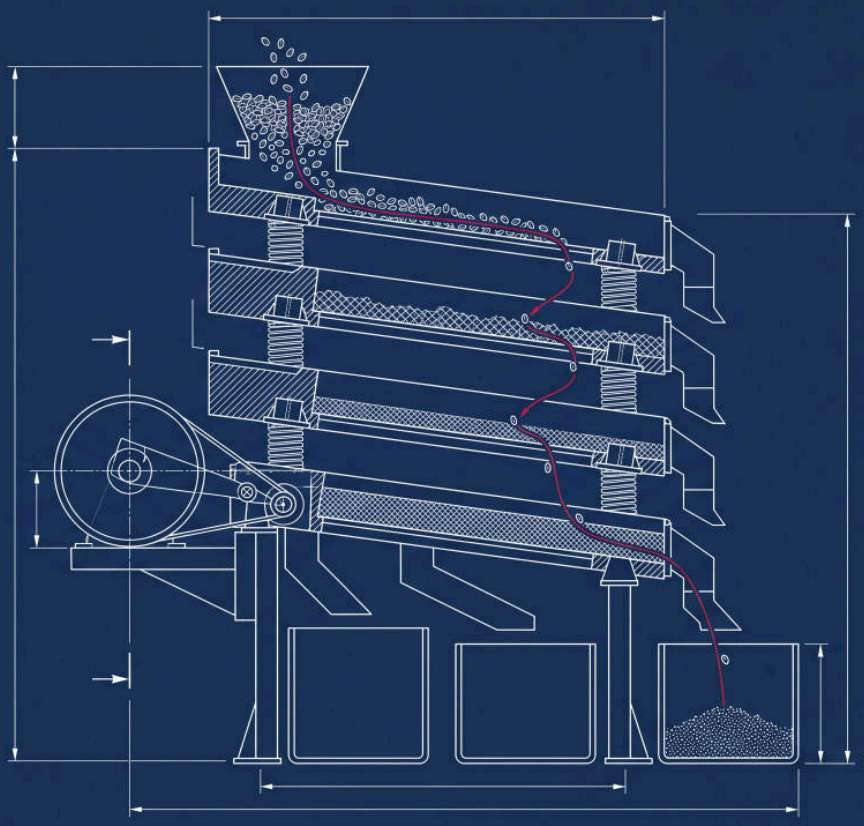

CHAPTER SEVEN

Retrieval Quality

SECTIONS 7.1 TO 7.11

The highest-leverage chapter in the book, and it opens by refusing to optimise anything. Most teams reach for a better model when the failure is three stages upstream, in a component nobody measures.

CHAPTER SEVEN

Contents

PART II · RETRIEVAL

This is the highest-leverage chapter in the book, and it opens by refusing to optimise anything. Most teams reach for a better model when the failure is three stages upstream, in a component nobody is measuring.

7.1 Why Naive RAG Fails: A Failure Taxonomy

Chapter 6 built a system that can cite and can refuse. It will still be wrong sometimes, and the useful question is not "how wrong" but **where**.

A RAG answer passes through five gates. It fails at exactly one of them, and the fix at each is different and *non-transferable*:

Stage	What went wrong	Where the fix lives
Not in corpus	The document was never ingested, or the scope excluded it	Ingestion pipeline *(6.2, 6.6)*
Not chunked usably	The answer exists but was cut in half, or lost its heading	Splitter *(6.5, 7.3)*
Not retrieved	The chunk was never in the candidate set at any k	Index recall, lexical leg, query rewrite *(7.5, 7.9)*
Not ranked high	Retrieved, then buried below the cut	Fusion and reranking *(7.6, 7.7)*
Not used by model	It was in the prompt and the model did not use it	Prompt, ordering, or capability *(5.1, 6.8)*

Reranking cannot help a chunk that was never indexed. A better embedding model cannot help a clause the splitter cut in half. A bigger model cannot help evidence that never reached the prompt. **Every one of those is a common and expensive mistake, and each is made by a team that skipped the measurement in section 7.2.**

7.2 Diagnosing Failures by Stage

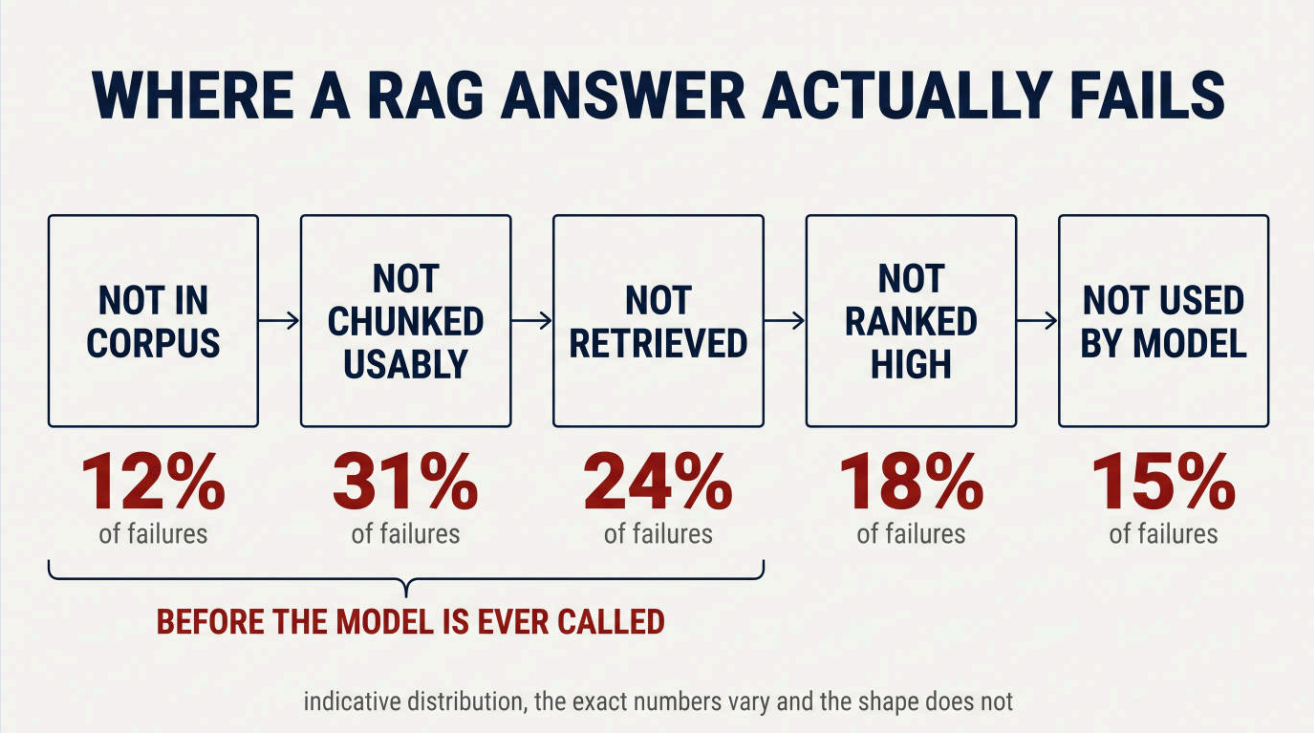

Figure 7.1 Indicative distribution. The exact numbers vary by corpus; the shape does not.

Before changing anything, take fifty failed cases and label each by stage. It takes an afternoon, requires no new infrastructure, and the result is almost always surprising.

```
# src/docassist/retrieval/diagnose.py
async def diagnose(case: EvalCase, pipeline: Pipeline, *,
                   wide_k: int = 50, shown_k: int = 4) -> Stage:
    """Assign a failure to EXACTLY ONE stage, checked in pipeline order.

    Order matters: a chunk that is not in the corpus is also not retrieved,
    and reporting it as a retrieval failure sends you to tune an embedding
    model when the document was never ingested.
    """
    gold = case.gold_chunk_ids

    if not any(pipeline.corpus_contains(cid) for cid in gold):
        return Stage.NOT_IN_CORPUS

    if not any(pipeline.chunk_is_intact(cid) for cid in gold):
        return Stage.NOT_CHUNKED_USABLY

    # Retrieve WIDE here, deliberately. The question is whether the candidate
    # was REACHABLE at all, not whether your production k found it.
    wide = set(await pipeline.retrieve_ids(case.question, k=wide_k))
    if not (gold & wide):
        return Stage.NOT_RETRIEVED

    reranked = set(await pipeline.rerank_ids(case.question, sorted(wide), shown_k))
    if not (gold & reranked):
        return Stage.NOT_RANKED_HIGH

    return Stage.NOT_USED_BY_MODEL
```

Two details make the difference between a diagnosis and a guess.

Checks run in pipeline order and return on the first hit. A chunk missing from the corpus also fails every later check, so testing retrieval first would label an ingestion bug as a retrieval bug and send a week of work to the wrong component.

The retrieval check uses a wide k, not your production k. The question at that gate is whether the candidate was *reachable*. If it appears at k=50 but not at k=20, that is a ranking problem, not a retrieval one, and the distinction determines whether you tune the embedding model or add a reranker.

The output is a work item, not a vibe

```
report = DiagnosisReport({
    Stage.NOT_IN_CORPUS: 6,      Stage.NOT_CHUNKED_USABLY: 15,
    Stage.NOT_RETRIEVED: 12,     Stage.NOT_RANKED_HIGH: 9,
    Stage.NOT_USED_BY_MODEL: 8,
})

report.before_the_model        # 0.84
report.priority()
#  15  not_chunked_usably  -> splitter (6.5, 7.3)
#  12  not_retrieved       -> index recall, hybrid leg, query rewrite (7.5, 7.9)
#   9  not_ranked_high     -> fusion and reranking (7.6, 7.7)
#   8  not_used_by_model   -> prompt, ordering, or model (5.1, 6.8)
#   6  not_in_corpus       -> ingestion pipeline (6.2, 6.6)
```

Eighty four percent of failures happened before the model was called. That is the number that reframes the meeting. It is also why the team's instinct, which is to try a larger model, addresses the smallest bucket at the highest price. The largest bucket points at the splitter, which is a two-day change to a component nobody was looking at.

Each stage carries the module its fix lives in, so the report is directly actionable:

```
def test_priority_points_at_a_component_not_a_vibe():
    report = DiagnosisReport({Stage.NOT_CHUNKED_USABLY: 15, Stage.NOT_RETRIEVED: 3})
    top_stage, count, where = report.priority()[0]
    assert top_stage is Stage.NOT_CHUNKED_USABLY
    assert "splitter" in where
```

> DO THIS BEFORE READING THE REST OF THE CHAPTER
>
> Sections 7.3 to 7.9 each describe a technique that helps at one stage and does nothing at the others. Adopting all of them because a paper said so is how a retrieval pipeline acquires six components and one improvement. The diagnosis tells you which two to build.

7.3 Chunking Strategies Compared

If `not_chunked_usably` is your largest bucket, which it frequently is, this is where the work goes. Section 6.5 built the structural splitter; this section is how to *choose* between strategies with a measurement rather than an argument.

Strategy	Fails when	Diagnostic signature
Fixed size	Clauses, tables and definitions cross the boundary	Gold chunk exists but is split across two ids
Recursive separators	Structure matters: tables, nested headings, cross-references	Chunk is intact but missing its heading context
Semantic (embedding shift)	The corpus has usable structure that layout already gave you for free	Boundaries move between two ingests of an unchanged document
Structure aware	Documents with no reliable structure: scans, transcripts, chat logs	Heading path is empty for most chunks

That last row matters and is often missed. **Structural chunking degrades to recursive chunking when the extractor found no structure**, which is exactly what happens on scanned documents. If your corpus is half scans, structural splitting helps half your corpus and the other half needs the vision path from section 6.2.

The semantic row is the strategy that splits where the embedding similarity between adjacent sentences drops, so boundaries follow topic rather than layout. It buys sensible units on unstructured prose, at the cost of an embedding pass at ingest, and its boundaries are not stable: re-embedding with a new model moves them, which churns content hashes and works against the idempotent upsert of section 6.6. The book defaults to structural because headings in professional documents are deterministic and already paid for; semantic splitting is the candidate to measure when the diagnosis points at chunking and the corpus has no structure to split on.

How to choose, in one afternoon

```
# Hold everything else fixed. Change ONLY the splitter, re-ingest, and run
# the retrieval eval from 7.10 plus an end-to-end answer metric.
for strategy in (split_fixed, split_recursive, split_structural):
    reindex(strategy)
    m = evaluate(retrieve, cases, k=20)
    print(strategy.__name__, m.recall_at_k, m.mrr, answer_accuracy())
```

Re-ingesting the whole corpus for each variant is the expensive part, which is the practical argument for the versioned index builds of section 1.2: you can hold two builds side by side and compare them on live traffic rather than rebuilding in place and hoping.

7.4 Small-to-Big and Contextual Prefixing

Two techniques that resolve the same underlying tension, and both are cheap.

The tension. The best unit for *retrieval* is small and specific: it has high relevance density and a focused embedding. The best unit for *generation* is larger and self-contained: it carries the surrounding context the model needs to answer. These are different requirements, and a single chunk size is a compromise between them.

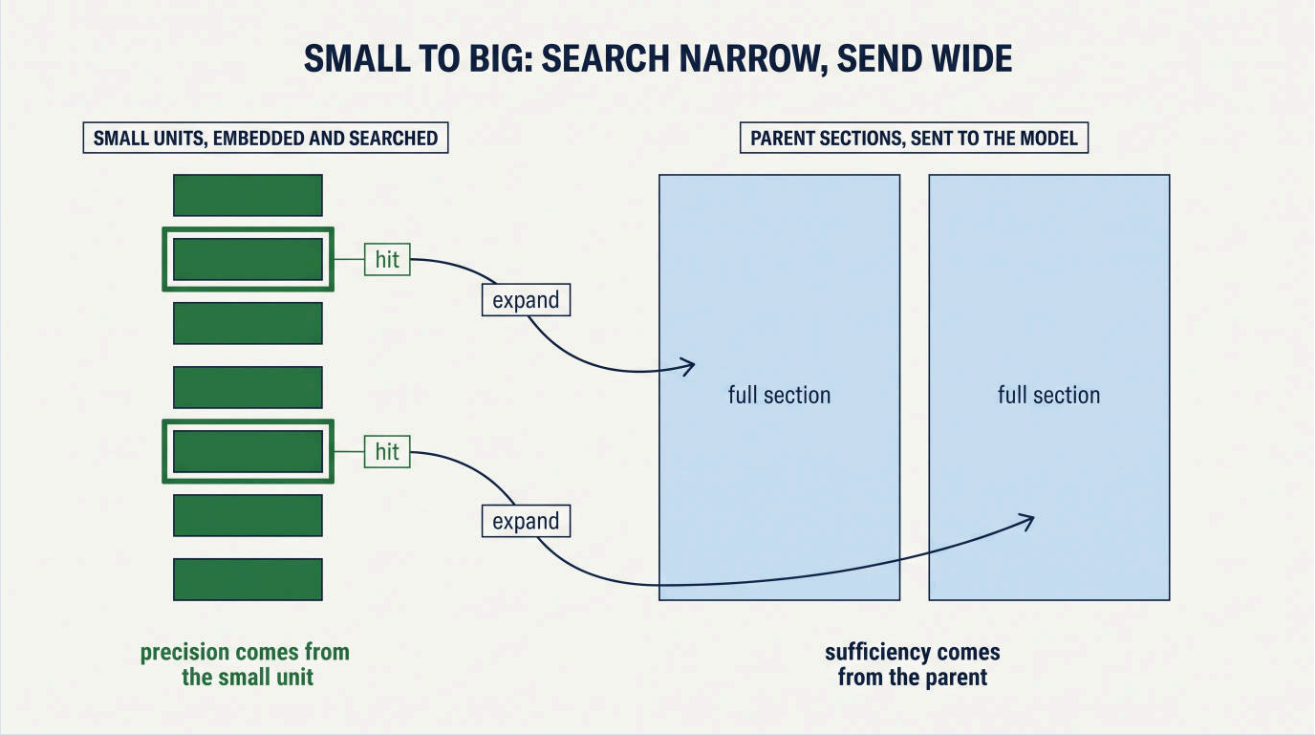

Figure 7.2 Precision from the small unit, sufficiency from the parent. One extra lookup.

Small to big

```
def expand_small_to_big(hits, parent_of: dict[str, str], keep: int) -> list[str]:
    """Search over small units, send the PARENT section to the model.

    Two hits inside one parent is a STRONG relevance signal, so the parent is
    boosted rather than returned twice: returning it twice would waste a slot
    in a budget that section 2.10 already made tight.
    """
    boosted: dict[str, float] = {}
    for chunk_id, score in hits:
        parent = parent_of.get(chunk_id, chunk_id)
        boosted[parent] = boosted.get(parent, 0.0) + score
    return sorted(boosted, key=lambda p: -boosted[p])[:keep]
```

```
def test_two_hits_in_one_section_boost_it_rather_than_duplicating_it():
    hits = [("c1", 0.4), ("c2", 0.35), ("c9", 0.5)]
    parents = {"c1": "S4", "c2": "S4", "c9": "S7"}
    out = expand_small_to_big(hits, parents, keep=2)
    assert out[0] == "S4"            # 0.75 combined beats 0.5
    assert out.count("S4") == 1      # boosted, not returned twice
```

Note what the boost encodes. A section containing two separately-retrieved passages is more likely to be the right section than one containing a single stronger passage, and summing the child scores expresses that without a tuning parameter. This is the same reasoning as reciprocal rank fusion in section 7.6: *agreement between independent signals is itself evidence.*

The `parent_id` field this depends on was populated in section 6.4, before it was needed, precisely because backfilling it means reprocessing the corpus.

Contextual prefixing

Already built in section 6.4 and worth restating here because it belongs to the same idea: prepend the section path to the text that gets *embedded*, while showing the user only the clause.

```
def embedding_text(self) -> str:
    return f"{self.section_path}\n\n{self.text}" if self.section_path else self.text
```

A chunk reading "shall not exceed 30 days" contains no noun a question would match on. The same chunk embedded as "Contract ABC > 4. Payment > 4.2 Late fees: shall not exceed 30 days" matches directly. This is usually the single highest-value change available to a system whose diagnosis points at `not_retrieved`, and it costs one string concatenation at ingestion.

That concatenation is the free tier of a technique the field now ships under the name **contextual retrieval**. When the document has usable structure, the section path *is* the context and you get it for a string join. The generated tier exists for the corpus that does not: prose where the chunk's referent lives three pages away. Consider a chunk that reads, in full, "revenue increased by twelve percent." Which company, which quarter, which revenue, gross or net: the chunk does not know, so its embedding does not know, so no question that names any of those things will land on it. The failure this repairs is not retrieving the *wrong* chunk. It is retrieving a chunk that forgot where it came from.

The mechanism is one model call per chunk at ingestion. The model sees the whole document and the chunk, and answers a single question: what would a reader need to know to understand this chunk without the rest of the document? It writes a two or three sentence header, "From Meridian plc's third-quarter 2025 interim report, in the segment discussion comparing hardware revenue with the prior quarter: revenue increased by twelve percent", and that header is prepended to the text that gets embedded, and to the text the lexical leg of section 7.5 indexes, while the displayed text stays the original clause, exactly as above. The arithmetic that makes this affordable is section 3.6's prompt caching: every chunk of one document shares the document as its prompt prefix, so the document is paid for once and each header costs pennies of marginal tokens, on the background tier where ingestion already runs per section 8.9.

The published measurements for the technique report retrieval failure rates falling by roughly a third from generated context alone, and by roughly two thirds when it is combined with the hybrid leg of section 7.5 and the reranker of section 7.7, which is worth reading carefully: the headline number belongs to the *stack*, not the trick, and the technique composes with this chapter rather than replacing any of it. Those are someone else's benchmarks. Your corpus decides, and the diagnosis that justifies the spend is specific: `not_retrieved` failures where the missing term is the chunk's implicit referent, the pronoun, "the company", "the agreement". A corpus whose section paths already carry those referents gets nothing from generated headers except an ingestion bill, which is why the free tier comes first and the measurement of section 7.10 decides the upgrade.

7.5 Hybrid Search: BM25 with Dense Vectors

Section 2.9 established that embeddings are topical. This section is the practical consequence: **dense retrieval fails predictably, and the failures are not edge cases in professional corpora, they are the majority of high-value queries.**

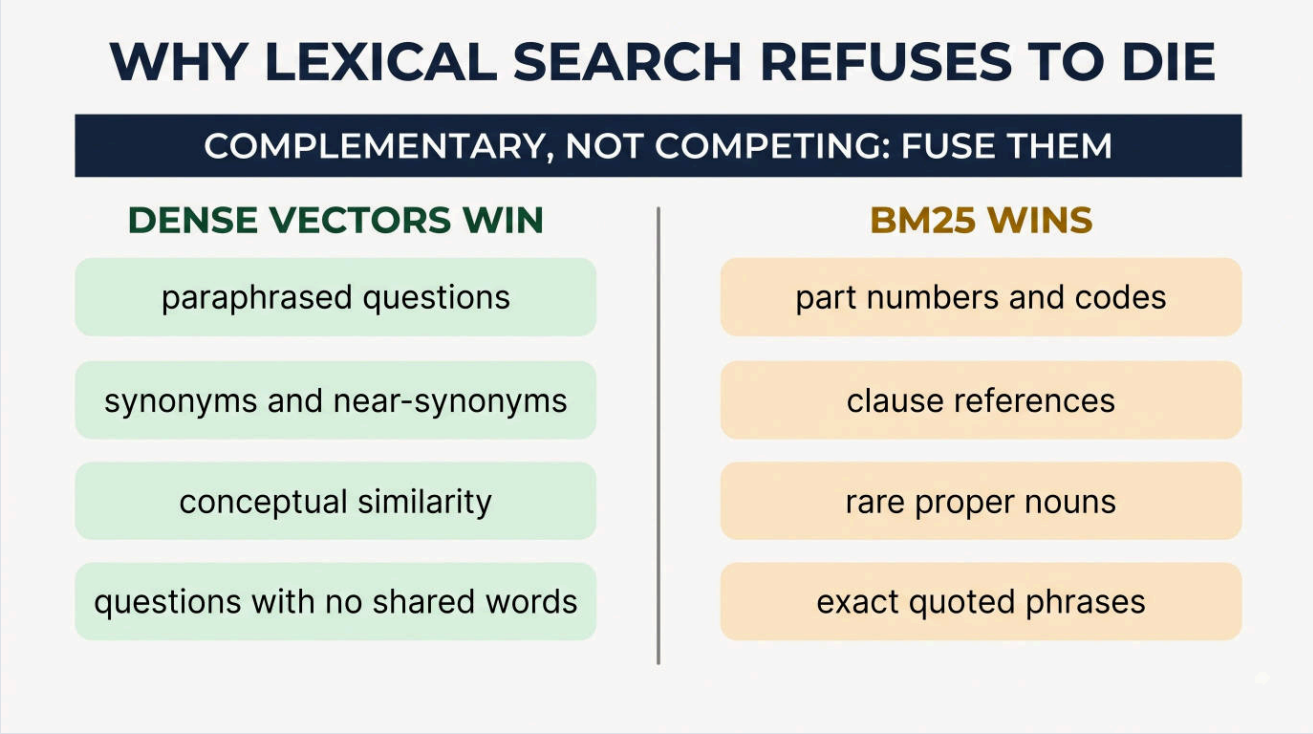

Figure 7.3 Complementary, not competing. Each is strong exactly where the other is weak.

A user asking "what is the replacement interval for A-1042/B" is not asking a conceptual question. They know the part number and they want the clause that mentions it. Dense retrieval maps that identifier into a region of embedding space populated by other alphanumeric strings, which is not where the answer is. BM25 finds it in one pass because the term is rare and the document contains it.

BM25 in eighty lines

```
# src/docassist/retrieval/hybrid.py
_TOKEN = re.compile(r"[a-z0-9]+(?:[.\-/][a-z0-9]+)*")

def tokenize(text: str) -> list[str]:
    """Keep internal punctuation in identifiers.

    'A-1042/B' must survive as ONE token, or the exact-match case this whole
    leg exists to serve is destroyed by the tokenizer before scoring begins.
    """
    return _TOKEN.findall(text.lower())
```

That regex is the most important line in the module and the easiest to get wrong. A default word tokenizer splits on the hyphen and the slash, turning one distinctive term into three common ones, and the lexical leg then performs no better than the dense one on precisely the queries it was added to serve.

```
def _idf(self, term: str) -> float:
    n = len(self._docs)
    df = self._df.get(term, 0)
    # +0.5 smoothing keeps a term present in EVERY document from going
    # negative, which would rank documents containing it BELOW ones that
    # do not.
    return math.log(1 + (n - df + 0.5) / (df + 0.5))
```

Run it on a small corpus and the complementarity is immediate:

```
bm.index({
  "d1": "The subbase layer shall be compacted to 95 percent density.",
  "d2": "Replace part A-1042/B before the next service interval.",
  "d3": "Pavement structures require adequate drainage provisions."})

bm.search("A-1042/B")            # [('d2', 0.98)]   dense search misses this
bm.search("compaction density")  # [('d1', 0.88)]   and finds this fine
```

> BUILD THE LEXICAL LEG FIRST
>
> Of everything in this chapter, this is the technique with the best ratio of gain to effort: roughly a day of work, no new infrastructure if your database supports full-text search, and it addresses a failure class that no amount of embedding-model upgrading will touch. If your diagnosis shows `not_retrieved` concentrated on questions containing identifiers, codes or proper nouns, this is the whole fix.

Two newer retriever families sit between the legs just described, and both are named here by property rather than by product, because the products change. **Learned sparse expansion**, the SPLADE class, uses a model to expand each document and query into weighted terms, so "compaction" also lights up "density" and "subbase". It occupies the lexical leg's slot in the architecture: an inverted index, exact-match strength preserved, with some of the dense leg's semantics added. Where the plain BM25 leg is failing on vocabulary mismatch rather than on identifiers, it is the upgrade to measure first.

Late interaction, the ColBERT class, embeds every token and scores query tokens against document tokens at query time. It sits between the bi-encoder and the cross-encoder of section 7.7 in both quality and cost: more precise than one vector per chunk, cheaper per candidate than a joint read, and paid for in index size, since every token stores a vector. It competes for the reranker's slot more often than for the retriever's.

Neither changes the architecture. Each is a leg, the legs still disagree, and the fusion of section 7.6 merges their rankings exactly as it merges the two you have. That is the payoff of fusing by rank: adding or swapping a leg is a local change, measured with the section 7.10 eval set like any other.

7.6 Reciprocal Rank Fusion

You now have two ranked lists that disagree. The instinct is to normalise the scores and take a weighted average. **Do not.**

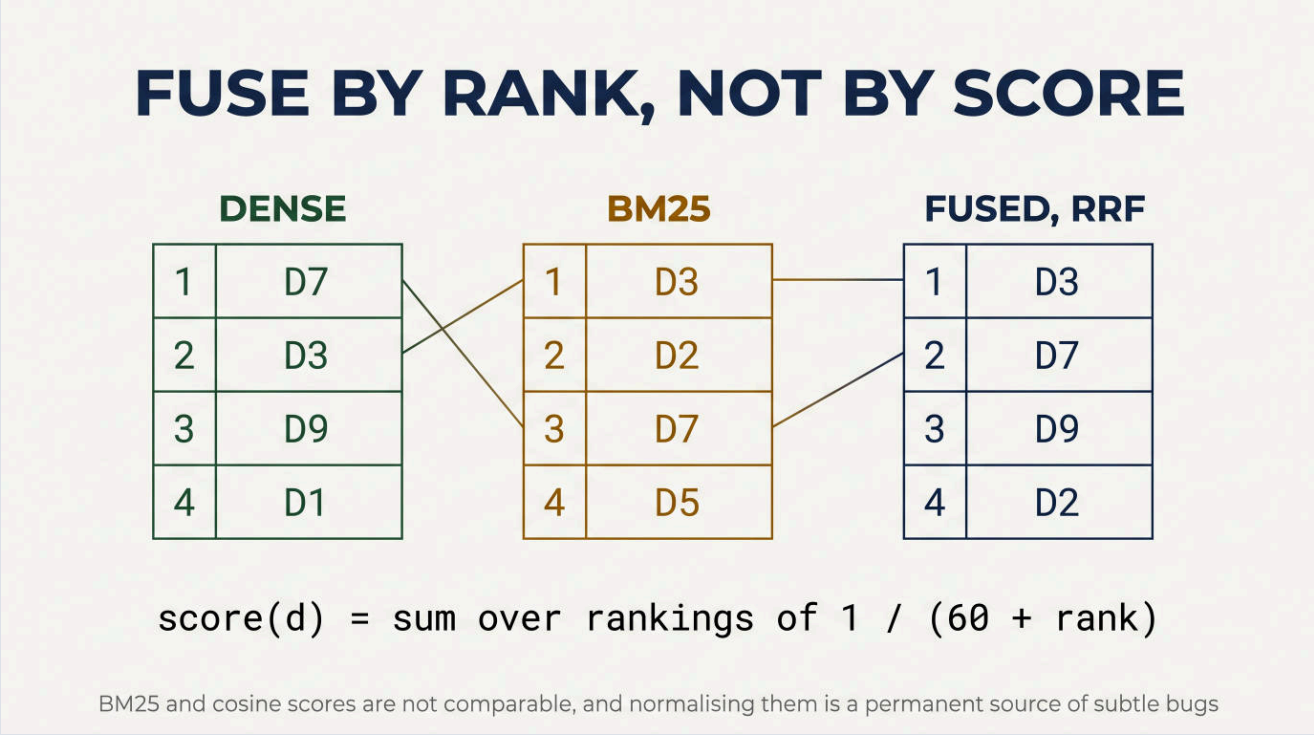

Figure 7.4 D3 ranks 2nd and 1st; D7 ranks 1st and 3rd. Agreement wins, and no score was compared.

```
def reciprocal_rank_fusion(rankings, k: int = 60) -> list[str]:
    """Fuse by RANK, not by score.

    BM25 scores are unbounded and corpus-dependent; cosine similarity is
    bounded and distribution-dependent. They are not comparable, and
    normalising them is a source of subtle, permanent bugs that surface as
    'retrieval got worse after we added more documents'.

    RRF needs no tuning, no calibration and no score-distribution assumption,
    which is why it is the right default rather than a compromise.
    """
    scores: dict[str, float] = {}
    for ranking in rankings:
        for rank, doc_id in enumerate(ranking, start=1):
            scores[doc_id] = scores.get(doc_id, 0.0) + 1.0 / (k + rank)
    return sorted(scores, key=lambda d: -scores[d])
```

The bug that score normalisation produces is worth understanding, because it is slow and confusing. BM25 scores scale with corpus statistics: as documents are added, the IDF of every term shifts, and so does the range of achievable scores. A weighting tuned when the corpus held ten thousand documents is wrong at a hundred thousand, and the symptom is a gradual quality decline with no code change to blame. RRF is immune because it never looks at a score.

```
def test_rrf_promotes_documents_both_rankings_agree_on():
    dense   = ["D7", "D3", "D9", "D1"]
    lexical = ["D3", "D2", "D7", "D5"]
    fused = reciprocal_rank_fusion([dense, lexical])
    assert fused[0] == "D3"          # rank 2 and rank 1
    assert fused[1] == "D7"          # rank 1 and rank 3
    assert fused.index("D9") > 1     # single-list hits fall below agreements
```

The constant 60 damps the top of each list so that a first place is not overwhelmingly more valuable than a third. It is the one knob, it is rarely worth touching, and its insensitivity is a feature: a fusion method with no parameters cannot be mistuned by someone who does not know your corpus.

7.7 Cross-Encoder Reranking

Section 2.9 stated the gap: cosine similarity measures whether two passages are *about* the same thing, not whether one *answers* the other. This section is the component that closes it.

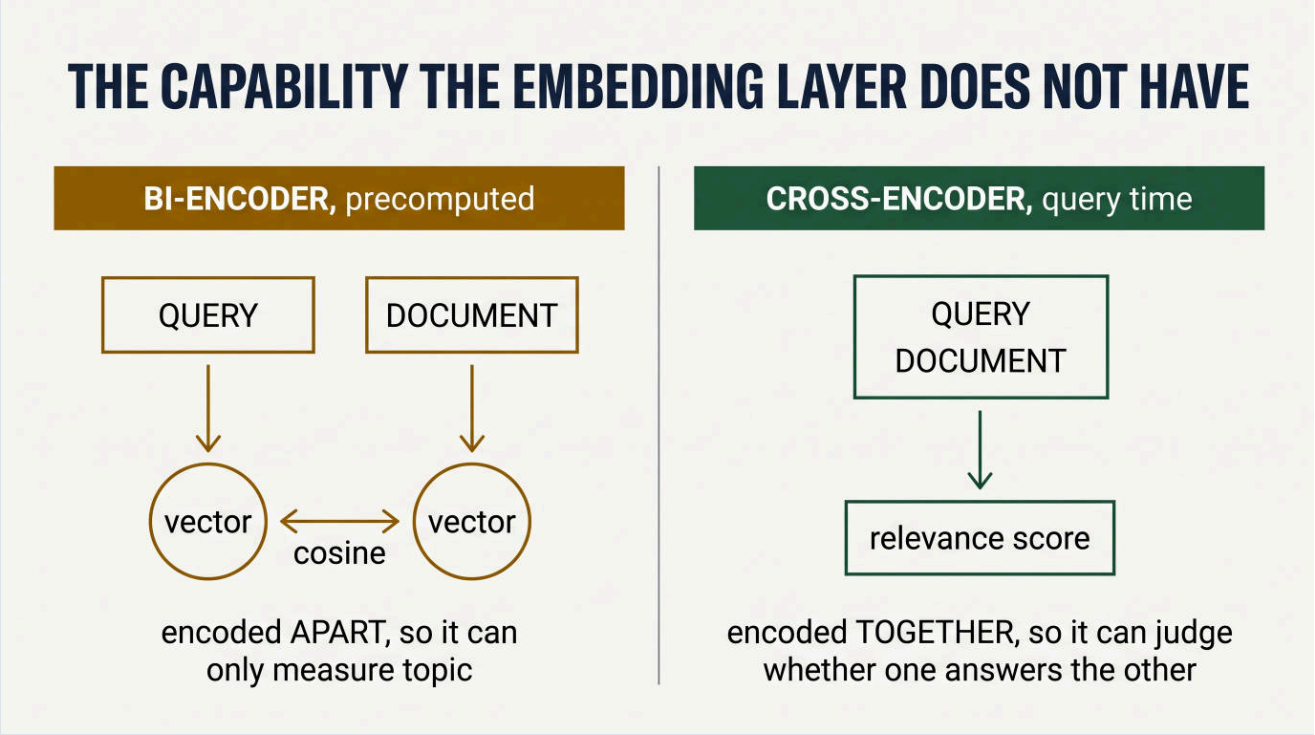

Figure 7.5 A bi-encoder never sees the query and document together. A cross-encoder cannot see anything else.

The architectural asymmetry follows directly. A bi-encoder embeds documents *independently of any query*, so the work is done at ingestion and search is a nearest-neighbour lookup. A cross-encoder reads query and document jointly, so nothing can be precomputed and the cost is paid per candidate at query time.

That constraint is what dictates the pipeline shape. You cannot cross-encode a million documents, so you retrieve broadly with the cheap method and rerank narrowly with the expensive one. Retrieve-broad-then-narrow is not a style choice; it is the only arrangement the cost structure permits.

Typical shape of the gain. Retrieving thirty candidates and reranking to four commonly improves answer accuracy by ten to twenty points over taking the top four directly, at a cost of roughly fifty to two hundred milliseconds. There are few interventions in this field with that ratio, which is why reranking belongs in the architecture rather than in a backlog of optimisations.

Treat that range as a hypothesis, not a promise: run your own eval set with the reranker on and off, and adopt it on the measured difference.

In code, the whole component is a sort and a cut, which is why it is so often the best return in the chapter:

```
candidates = await retrieve(question, scope, k=30)       # broad and cheap (6.7)
scores = await reranker.score(question, [c.text for c in candidates])
top = [c for c, _ in sorted(zip(candidates, scores),
                            key=lambda p: -p[1])[:4]]     # narrow and expensive
```

By 2026 there are three ways to buy the scorer, and the choice is operational rather than architectural. A **self-hosted cross-encoder** gives the lowest per-query cost at volume and adds a model deployment to run. A **hosted rerank API** is one HTTP call scoring the candidate list, priced per request, with tens of milliseconds of network added: the right default when you are not already serving models. **LLM listwise reranking**, asking a general model to order the candidates, is the most capable on hard queries and the most expensive per query by an order of magnitude, with generation latency on top; it is a technique to measure into a narrow slot, not a default. All three fit the same rerank-then-truncate seam, so the decision is reversible.

Two operational notes. The reranker's output scores are *comparable within one query* and not across queries, so a relevance floor (section 6.9) calibrated on rerank scores must be recalibrated whenever the reranker changes. And reranking is the natural place for the cache in section 7.11, because the same query over the same candidate set recurs more often than you would expect.

7.8 Metadata Filtering and Query Routing

The cheapest retrieval improvement available is not retrieving the wrong things in the first place.

Section 6.10 established the mandatory filters: tenant, and current version. Beyond those, metadata narrows the

candidate space before similarity is computed at all:

```
# A question about a specific contract should not search the whole corpus.
scope = QueryScope(tenant_id=ctx.tenant_id, doc_ids=("acme-msa-2026",))

# A question with a temporal qualifier can filter by effective date.
# A question about a document TYPE can filter by it.
# Each filter removes candidates that similarity would otherwise have to
# out-rank, which is a strictly easier problem for the reranker.
```

Query routing is the same idea applied one level up: decide which retrieval strategy to use before running any of them. A question containing a quoted phrase or a part number should weight the lexical leg. A question naming a single document should filter to it. A question with no entities at all is purely conceptual and the dense leg carries it.

Use the same discipline section 3.6 applied to model routing: **a static table over known patterns beats a classifier** until measurement shows the table is insufficient. It is free, deterministic and reviewable, and it cannot drift.

7.9 Query Rewriting and Multi-Query Expansion

Users write "what about the penalties" as a third turn. Embedding that string retrieves nothing useful, because the information need lives in the conversation rather than in the query.

Transformation	What it fixes	When to apply it
Contextualisation	Rewrites a follow-up into a standalone question using the conversation	Always, in any multi-turn product. Cheap and high value.
Decomposition	Splits a multi-part question into sub-queries, retrieves for each, merges	Gate on a cheap classifier. Necessary for comparative and multi-hop questions, wasteful on simple ones.
Hypothetical document	Generates a plausible answer and embeds *that*, on the theory that an answer sits closer to the passage containing it than the question does	Measure before adopting. Genuinely effective on some corpora and adds a full generation to every query.

All three add latency and cost to the retrieval path, and all three are worth exactly what they measure. Contextualisation is the one that is close to mandatory: without it, the second turn of every conversation retrieves as though the first had not happened.

Knowledge-graph retrieval, the GraphRAG family, is the heavier alternative for the multi-hop case: it moves the join from query time to ingest time and pays for an entity-extraction pipeline to do it. That trade earns its keep when the same joins recur across many queries; for docassist's ad hoc questions over customer documents, decomposition plus fusion serves the need without the standing cost, which is why this book considers it and does not build it.

> REWRITING CAN LOSE THE IDENTIFIER
>
> A rewrite that turns "what about A-1042/B" into "what is the replacement interval for the part discussed above" has just destroyed the exact term that section 7.5's lexical leg needed. When you rewrite, **send both the original and the rewrite to the retriever** and fuse the results. RRF handles the merge without needing to decide which one was better.

7.10 Measuring Retrieval: Recall, MRR, Faithfulness

None of the techniques in this chapter can be adopted responsibly without measurement, because each has corpora where it hurts. **The eval set is the deliverable; the pipeline is the easy part.**

```
@dataclass(frozen=True)
class RetrievalCase:
    question: str
    relevant_ids: frozenset[str]     # labelled by a HUMAN, from real traffic


def evaluate(retrieve, cases, k: int) -> RetrievalMetrics:
    for case in cases:
        hits = list(retrieve(case.question, k))
        found = set(hits) & case.relevant_ids
        recalls.append(len(found) / max(1, len(case.relevant_ids)))
        precisions.append(len(found) / max(1, len(hits)))
        rank = next((i for i, h in enumerate(hits, 1) if h in case.relevant_ids), None)
        rrs.append(1.0 / rank if rank else 0.0)
```

Read the three numbers together, never separately

```
def read_together(self, prev: "RetrievalMetrics | None" = None) -> str:
    """Recall holding or rising while MRR falls means you are retrieving the right thing
    and BURYING it: a ranking problem, not a retrieval one.

    Precision rising while end-to-end accuracy falls means k is now too small
    and you have cut the answer out.
    """
    if self.recall_at_k >= prev.recall_at_k and self.mrr < prev.mrr:
        return "retrieving it and burying it: fix ranking (7.6, 7.7)"
    if self.recall_at_k < prev.recall_at_k:
        return "candidate set lost the answer: fix retrieval (7.5, 7.9)"
    return "improved on both"
```

Each pair of movements names a different component. That is the whole value of tracking three numbers instead of one: a single metric tells you something changed, and the combination tells you *what*.

The trap that makes this chapter necessary

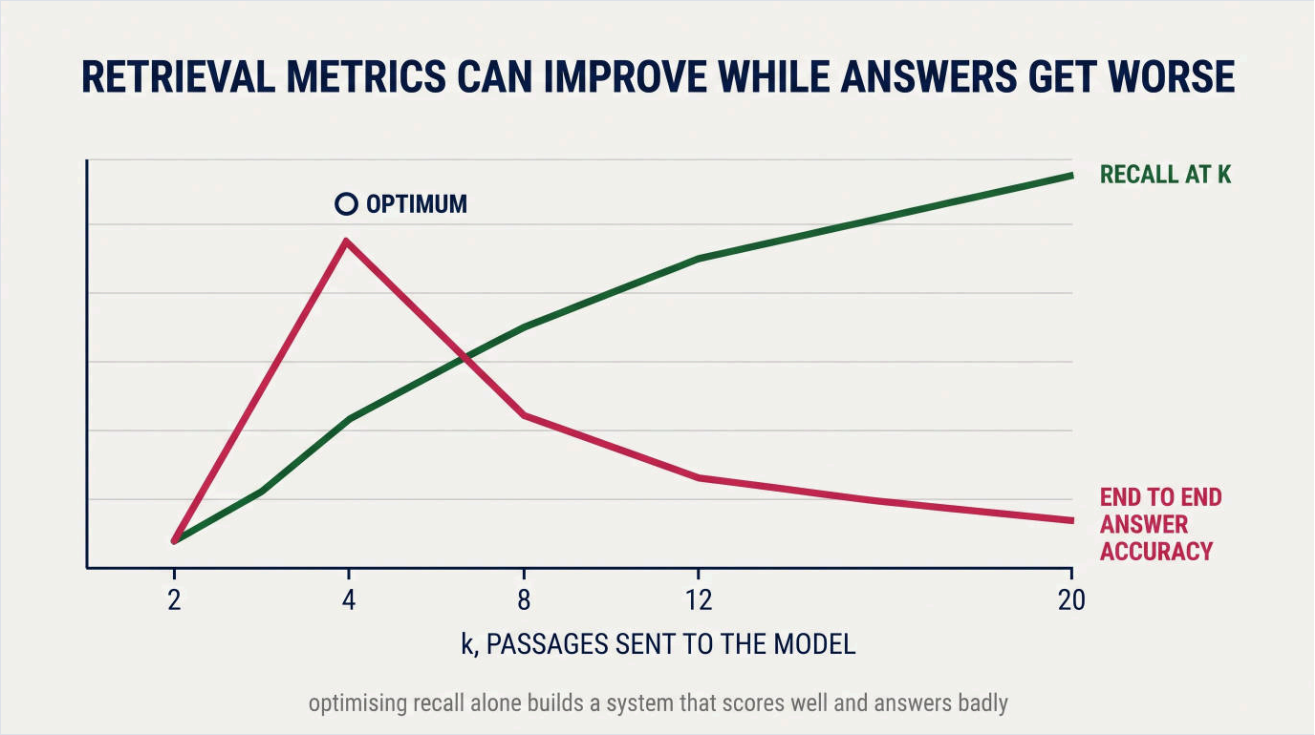

Figure 7.6 Recall climbs monotonically with k, almost by definition. End-to-end accuracy peaks and falls.

```
rows = sweep_k(retrieve, answer_accuracy, cases, ks=(2, 4, 8, 12, 20))

# {'k':  2, 'recall_at_k': 0.5, 'mrr': 0.500, 'answer_accuracy': 0.55}
# {'k':  4, 'recall_at_k': 1.0, 'mrr': 0.500, 'answer_accuracy': 0.88}
# {'k':  8, 'recall_at_k': 1.0, 'mrr': 0.500, 'answer_accuracy': 0.81}
# {'k': 12, 'recall_at_k': 1.0, 'mrr': 0.500, 'answer_accuracy': 0.74}
# {'k': 20, 'recall_at_k': 1.0, 'mrr': 0.500, 'answer_accuracy': 0.66}

best_k_by_answers(rows)     # 4
```

At k=2 half the gold evidence never reaches the candidate set, and accuracy is correspondingly poor: partial evidence yields partial answers and abstentions. Now look at what a recall-driven team would conclude. Recall is 1.0 from k=4 onward, so raising k further costs nothing by that metric and feels safer. Meanwhile answer accuracy falls from 0.88 to 0.66, a twenty-two point loss, through the dilution and positional effects of section 2.10.

> RETRIEVAL METRICS ARE INSTRUMENTS, NOT OBJECTIVES
>
> Optimising recall@k directly is one of the most reliable ways to build a system that scores well and answers badly. **Always report an end-to-end answer metric alongside the retrieval ones**, and treat recall and MRR as diagnostics that tell you which component to look at rather than as targets to maximise.
>
> Note also that k at the *retriever* and k at the *prompt* are different numbers with different owners, per section 6.7. Retrieve thirty; show four. Confusing them is how this trap gets sprung.

The third instrument in this section's title is the one the sweep cannot supply. **Faithfulness is the fraction of answer**

claims entailed by the retrieved evidence. The machinery already exists: the `GroundingChecker` of section 6.9 runs claim-level entailment at runtime as a gate, deciding whether one answer ships. Run the same check over the eval set and average it, and the gate becomes a metric:

```
async def faithfulness(cases, pipeline, checker: GroundingChecker) -> float:
    """The 6.9 gate, run as a metric over the eval set. No new machinery."""
    fractions = []
    for case in cases:
        answer, evidence = await pipeline.answer(case.question)
        if not answer.answered:
            continue                # abstentions are scored by answer rate
        r = await checker.verify(answer.claims, evidence)
        fractions.append(len(r.supported) /
                         max(1, len(r.supported) + len(r.unsupported)))
    return mean(fractions)
```

Keep the three numbers distinct, because each answers a different question. Recall asks whether the evidence arrived in the candidate set. Answer accuracy asks whether the answer was correct. Faithfulness asks whether the answer *came from* the evidence, and it is the only one of the three that measures the property the word "grounded" claims.

They can disagree, and the disagreement worth watching for is accurate but unfaithful: the model answered correctly from its own weights while the citation it printed does not support the claim. That passes every accuracy check and fails the book's grounding requirement anyway. The correctness is borrowed and unverifiable, a reader who follows the citation finds it does not say what the answer said, and the same behaviour on a question the weights get wrong produces a confident fabrication. High accuracy with low faithfulness is a warning, not a success.

Read the three together with the sweep above. Recall names the retrieval stage, accuracy names the outcome, and faithfulness says whether the outcome was earned. Recall low while accuracy stays high means the system is answering from weights, so tighten the relevance floor and fix retrieval. Recall flat while faithfulness falls as k rises means the model is drifting off the evidence in a diluted prompt, which is the trap above wearing a different metric. A change is safe to ship when accuracy holds *and* faithfulness has not fallen.

Building the case set

Fifty to two hundred labelled cases drawn from real questions is enough to make decisions. The labelling is tedious and it *is* the work: every technique in this chapter is a hypothesis, and without the eval set you are adopting hypotheses because they appeared in a paper about a different corpus.

Two sources beat synthetic generation. **Real questions from your own traffic**, which you have if you built the trace seam in Chapter 1. And **the dead letter queue** from section 8.10 once it exists, because it contains exactly the inputs your pipeline could not handle.

7.11 Caching Embeddings and Responses

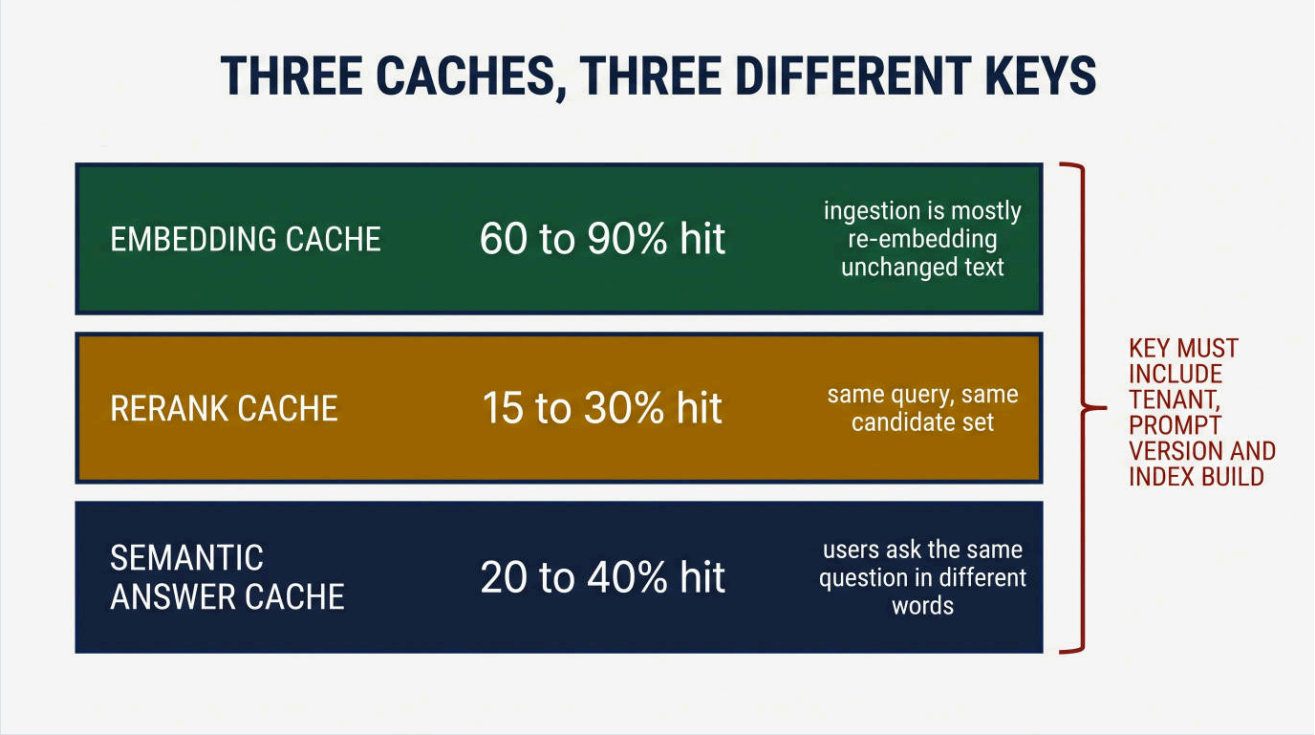

Figure 7.7 Three caches at three layers. The keys differ, and getting a key wrong is a correctness bug rather than a performance one.

Chapter 3 built the semantic answer cache. Two more belong in the retrieval path, and the reason to treat them separately is that *each has a different key*.

Cache	Key must include	Why it hits
Embedding	content hash, embedding model, dimensions	Ingestion is mostly re-embedding text that did not change. Section 6.6's idempotent upsert is this cache, expressed at the index layer.
Rerank	query, candidate id set, reranker version	The same question over the same candidates recurs, particularly on popular documents.
Semantic answer	tenant, prompt version, model, document version	Users ask the same question in different words. Section 3.6.

The rerank cache key deserves a note: it must include the *candidate set,* not just the query. The same question against a corpus that has since been re-ingested has a different candidate set and must not hit. Keying on the query alone produces stale rankings that survive an index rebuild, which is a bug that presents as "retrieval quality regressed after ingestion" and is very hard to find.

And the general rule from section 3.6 holds for all three: **the namespace must include every input that changes the answer.** Omit the tenant and you have a data leak. Omit the prompt version and your eval gate measures one thing while production serves another. Omit the index build id and a migration serves answers from the old index indefinitely.

One request, assembled

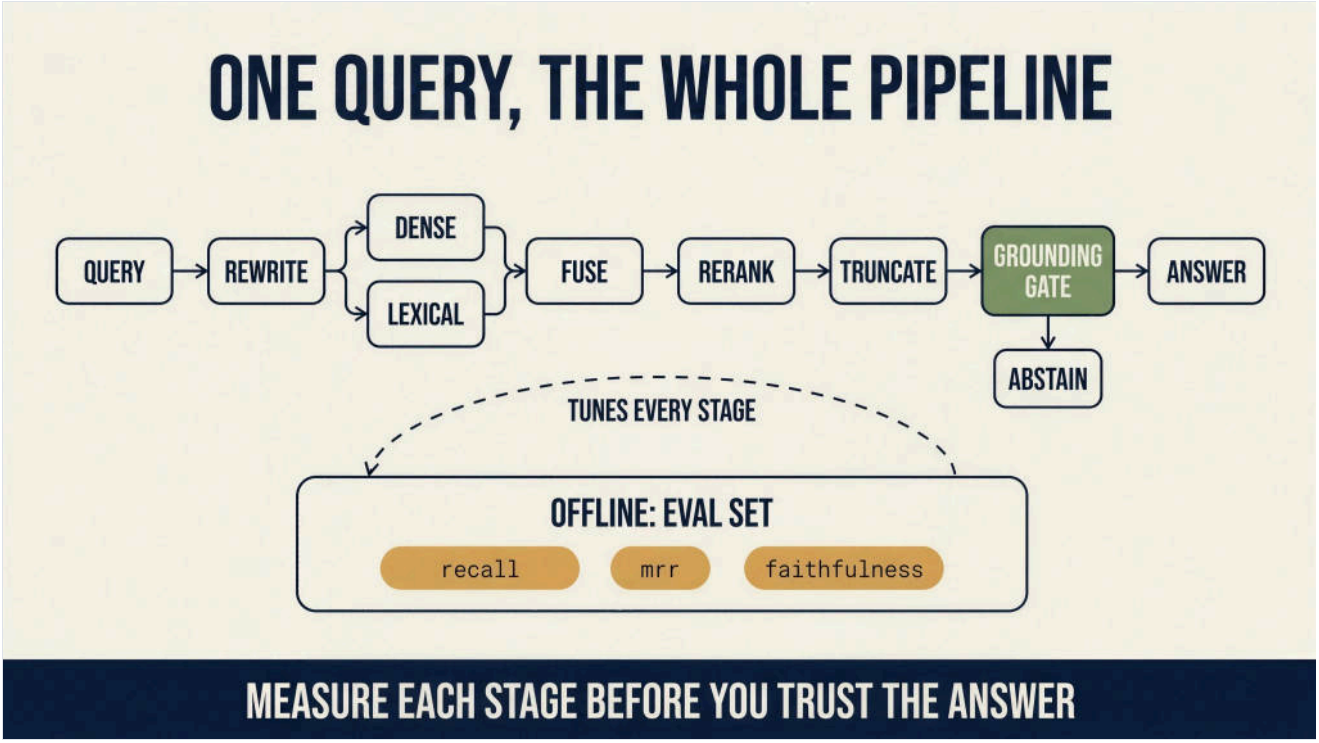

Figure 7.8 The chapter in one line: rewrite, retrieve twice, fuse, rerank, truncate, gate. The dashed loop is what makes the rest defensible.

This chapter is organised by technique, which is how you diagnose. A request does not arrive that way, so here is the same material in the order it fires, once, on one query.

The query is rewritten before it touches an index: contextualised against the conversation, decomposed if it is two questions wearing one question mark, with section 7.9's warning enforced, the exact identifiers from the original preserved through every rewrite. The rewritten query runs twice, dense and lexical in parallel, because section 7.5 showed the two lists fail on different queries, and section 7.6 fuses them by rank, not score. The fused list is deliberately broad; section 7.7's cross-encoder then spends its latency budget ranking that breadth, and the truncation that follows is where section 7.10's k-sweep trap is dodged: what reaches the prompt is the narrow, reranked head, not the recall-maximising tail. Chapter 6 assembles what survives, and section 6.9's grounding gate decides whether the answer earns its citations or becomes an abstention.

Everything to the left of the answer runs inside the latency budget. The judgement of whether to *trust* what comes out does not: it happened earlier, offline, on section 14.2's eval set, where recall says the evidence arrived, MRR says it arrived where the model will use it, and faithfulness says the answer that came back actually rests on it. Those three numbers, read together per section 7.10, are what tuned every stage in the line above. The runtime gate checks *this* answer; the eval numbers justify the pipeline that produced it. A team that has the gate without the numbers is trusting an architecture on reputation, and this chapter exists because reputation is exactly what the measurements keep contradicting.

What Chapter Seven Establishes

What exists now	What later chapters put through it
Stage diagnosis over failed cases	Continuous failure triage from production traces *(14.2)*
BM25 and reciprocal rank fusion	Retrieval exposed as an MCP tool *(12.8)*
Small-to-big expansion	Memory retrieval scoped by subject *(11.6)*
Retrieval metrics read together	The eval gate and canary *(14.6)*
The k-sweep and end-to-end pairing	Quality proxies during rollout *(16.3)*
Three caches with distinct keys	Cost dashboards and hit-ratio alerting *(16.2)*

Chapter checklist

Can you...	Section
Name the five failure stages and the component that fixes each	7.1
Say why the diagnosis must check stages in pipeline order	7.2
Explain why the retrieval check uses a wider k than production	7.2
State what fraction of your own failures happen before the model	7.2
Say when structural chunking degrades to recursive, and what to do	7.3
Explain why two hits in one section boost it rather than duplicate it	7.4
Give four query types where BM25 beats dense retrieval	7.5
Explain why score normalisation breaks as the corpus grows	7.6
Say why a cross-encoder cannot be precomputed, and what that dictates	7.7
Name the risk a query rewrite introduces for exact terms	7.9
Read recall, MRR and answer accuracy together to name a component	7.10
Say what each of the three caches must include in its key	7.11

What Chapter 8 builds

Chapters 6 and 7 built a retrieval pipeline that is accurate and, in places, slow: hybrid retrieval runs two searches, reranking adds a model call, and ingestion parses layout and embeds hundreds of chunks per document. All of that currently happens inside a request.

Chapter 8 moves it. Generation takes seconds to minutes and web request handling is designed for milliseconds, and almost every scaling failure in an AI product is a consequence of ignoring that mismatch for one release too long. The arithmetic is stark: sixteen workers at fourteen seconds per generation is a hard ceiling of roughly one request per second, regardless of how many cores the machine has. Then classes of service, so that a customer's ten-thousand-document import cannot starve interactive chat, and idempotent handlers, so that at-least-once delivery does not mean at-least-once side effects.