

Retrieval Augmented Generation

Aayush Bajaj

July 10, 2026

Contents

1	why retrieve at all	1
2	dense retrieval	2
3	approximate nearest-neighbour indexes	2
4	chunking	3
5	reranking	3
6	the generation step	3
7	failure modes	4
8	evaluation	4
9	toy dense retrieval in numpy	4
10	see also	5

retrieval-augmented generation bolts a search engine onto a language model: fetch relevant documents at query time and paste them into the prompt, so the model answers from evidence rather than from its frozen weights (lewis et al. 2020, *retrieval-augmented generation for knowledge-intensive nlp*). it is the pragmatic alternative to baking every fact into an **llm**'s weights.

the cheapest way to give an llm new knowledge without retraining it

1 why retrieve at all

three problems that no amount of scaling fixes cleanly, and retrieval fixes cheaply:

- **knowledge cutoff.** a model's parameters freeze at training time; retrieval injects information that did not exist then — today's prices, this week's tickets, your private wiki.
- **grounding.** parametric knowledge is lossy and confabulates confidently. conditioning generation on retrieved text sharply reduces hallucination on factual queries, because the answer is copied from context rather than reconstructed from weights.
- **provenance.** retrieved chunks carry source ids, so the system can **cite** — the user can check the claim against the document. a bare llm cannot tell you where its answer came from.

updating a rag system is editing a document store; updating a model’s parametric knowledge is fine-tuning (**tune**) or full retraining. that asymmetry is the whole value proposition.

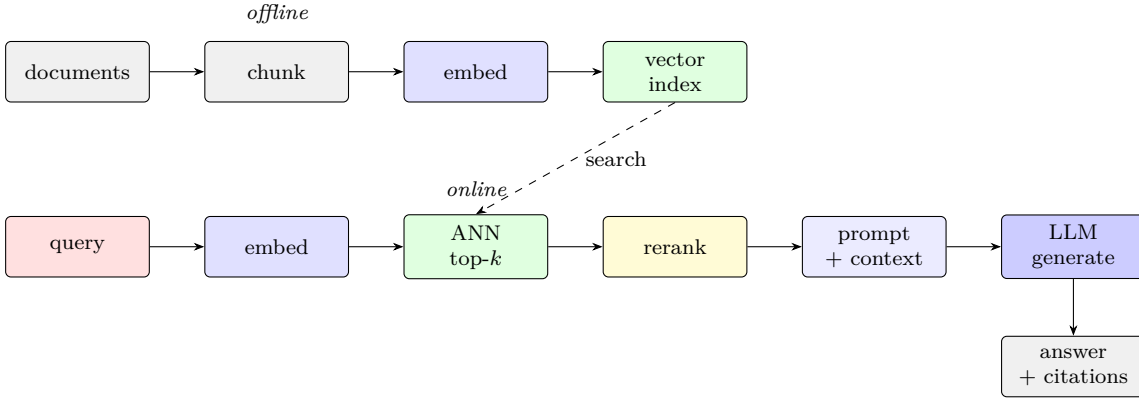


Figure 1: the rag pipeline. offline: documents are chunked and embedded into a vector index. online: the query is embedded, nearest chunks retrieved, optionally reranked, packed into the prompt, and answered with citations.

2 dense retrieval

classical search (bm25) matches words. **dense retrieval** matches meaning: a **bi-encoder** maps query and document independently to vectors in a shared space, and relevance is their similarity (karpukhin et al. 2020, *dense passage retrieval*).

- **bi-encoder.** two **transformer** encoders (often tied), $q \mapsto E_Q(q)$ and $d \mapsto E_D(d)$. crucially the document vectors are computed *once, offline*, so query time is a single query encode plus a vector search — this independence is what makes dense retrieval fast enough to scale.
- **similarity.** cosine similarity, or inner product when vectors are normalised:

$$\text{sim}(q, d) = \frac{E_Q(q) \cdot E_D(d)}{\|E_Q(q)\| \|E_D(d)\|}. \quad (1)$$

- **contrastive training.** the encoders are trained to pull a query toward its true passage and push it from irrelevant ones, via the infonce loss over a batch with one positive d^+ and negatives d_j^- :

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(q, d^+)/\tau)}{\exp(\text{sim}(q, d^+)/\tau) + \sum_j \exp(\text{sim}(q, d_j^-)/\tau)}. \quad (2)$$

the trick that makes it work is **in-batch negatives**: every other document in the batch serves as a negative for free, so a batch of B pairs gives $B - 1$ negatives per query at no extra encoding cost. hard negatives (bm25’s top wrong answers) sharpen it further.

3 approximate nearest-neighbour indexes

exact nearest-neighbour over millions of vectors means a full scan — $O(nd)$ per query, too slow at scale. **ann** indexes trade a little recall for large speedups:

- **hnsw** (hierarchical navigable small world): a multi-layer proximity graph; search greedily walks toward the query, descending layers. roughly $O(\log n)$ query time, excellent recall, high memory (the graph) and slow to build. the default for quality.
- **ivf** (inverted file): k-means-cluster the vectors, search only the few clusters nearest the query. tunable via how many clusters you probe — fewer is faster and lower recall. often paired with **product quantisation** to compress vectors ~10–50x, at some accuracy cost.

honest complexity note: “approximate” is load-bearing — these structures do not guarantee the true top- k , they return it with high probability, and the recall/latency/memory trade is a knob you tune per workload, not a solved problem.

4 chunking

documents must be split before embedding, because one vector cannot faithfully summarise a whole pdf. the trade:

- **too large** — the chunk’s embedding averages many topics into mush, and retrieval precision drops.
- **too small** — chunks lose the surrounding context needed to be interpretable (“it increased 12%” — what did?).

typical practice: 200–500 token chunks with ~10–20% **overlap** so a fact straddling a boundary survives in at least one chunk. structure-aware splitting (on headings, paragraphs, code blocks) beats blind fixed-size cutting.

5 reranking

the bi-encoder is fast but crude — it compressed each document to one vector before ever seeing the query. a **cross-encoder** reranker fixes this for the shortlist: it feeds the query and a candidate *together* through a transformer, so every query token attends to every document token, and outputs a single relevance score.

$$\text{score}(q, d) = \text{CrossEncoder}([q; d]) \in \mathbb{R}. \quad (3)$$

it is far more accurate and far too slow to run over the whole corpus — quadratic attention over (q, d) for every document. so the standard two-stage recipe: bi-encoder retrieves top-100 cheaply, cross-encoder reranks those 100 to a top-5. best of both — the recall of dense retrieval, the precision of full cross-attention, only on a shortlist.

6 the generation step

the top chunks are formatted into the prompt with an instruction to answer **only from the provided context and to cite sources**. good grounding prompts:

- interleave chunk text with explicit source markers ([1], [2]) so the model can attribute claims.
- instruct the model to say “not in the context” rather than fall back on parametric guessing — retrieval’s whole point is undone if the model ignores the evidence.

7 failure modes

rag has three characteristic ways to be wrong, and they compound:

- **retrieval miss.** the right chunk never makes the top- k (bad embeddings, bad chunking, vocabulary mismatch). the generator then answers from parametric memory, confidently and unsourced — the worst failure because it looks grounded.
- **context stuffing.** cramming in as many chunks as fit dilutes signal with noise and can push the true answer past the model's effective attention.
- **lost in the middle.** models attend most strongly to the *start* and *end* of a long context and neglect the middle (liu et al. 2023, *lost in the middle*) — so a correct chunk buried in position 8 of 15 may be ignored despite being retrieved. reranking to put the best chunk first is a partial fix.

8 evaluation

rag is two systems, so evaluate both:

- **retrieval: recall@k** (did any relevant chunk make the top k ?), plus rank-aware measures like mrr and ndcg. this is the ceiling — the generator cannot cite what was never retrieved.
- **generation: faithfulness** (is every claim supported by the retrieved context?) and **answer relevance** (does it address the question?). increasingly scored by an llm-judge against the retrieved chunks, with the usual caveat that the judge is itself fallible.

9 toy dense retrieval in numpy

the retrieval core — embed, normalise, cosine top- k — with a hashing-trick embedding standing in for a trained bi-encoder. real systems swap **embed** for a transformer; the search logic is identical.

```
1 import numpy as np
2 import re
3 from zlib import crc32                                # deterministic hash (unlike builtin
  ↪ hash())
4
5 # --- toy corpus of "documents"
6 docs = [
7     "the cat sat on the warm mat by the fire",
8     "dogs are loyal animals and make great pets",
9     "the kitten chased a ball of red wool",
10    "python is a popular programming language",
11    "neural networks learn features from raw data",
12    "a warm fireplace keeps the cottage cosy in winter",
13 ]
14
15 def tokenize(s):
16     return re.findall(r"[a-z]+", s.lower())
17
18 # --- hashing-trick embedding: map each token to a fixed bucket, count = bag of
  ↪ words
19 DIM = 24
20 def embed(text):
21     v = np.zeros(DIM)
```

```

22     for tok in tokenize(text):
23         v[crc32(tok.encode()) % DIM] += 1.0 # token -> fixed bucket
24     n = np.linalg.norm(v)
25     return v / n if n else v # L2-normalise so dot product = cosine
26
27 D = np.stack([embed(d) for d in docs]) # (n_docs, DIM) document matrix
28
29 def retrieve(query, k=3):
30     q = embed(query)
31     sims = D @ q # cosine similarity (all vectors
32     ↪ unit-norm)
33     order = np.argsort(sims)[::-1][:k]
34     return [(docs[i], sims[i]) for i in order]
35
36 for query in ["a warm cosy fireplace", "loyal pet animals", "learn features
37 ↪ from data"]:
38     print(f"query: {query!r}")
39     for doc, sim in retrieve(query):
40         print(f" {sim:.3f} {doc}")
41     print()

```

```

query: 'a warm cosy fireplace'
0.667 a warm fireplace keeps the cottage cosy in winter
0.408 python is a popular programming language
0.333 neural networks learn features from raw data

```

```

query: 'loyal pet animals'
0.577 dogs are loyal animals and make great pets
0.385 a warm fireplace keeps the cottage cosy in winter
0.236 python is a popular programming language

```

```

query: 'learn features from data'
0.833 neural networks learn features from raw data
0.333 a warm fireplace keeps the cottage cosy in winter
0.316 the kitten chased a ball of red wool

```

each query pulls the right document to rank 1 by cosine alone. the noisy tail (a fireplace matching “loyal pet animals” at 0.385) is exactly the weakness a trained bi-encoder plus a cross-encoder reranker exists to remove — a bag-of-words embedding shares buckets between unrelated words, whereas a learned encoder places *meaning*, not spelling, near meaning.

10 see also

- [llms](#) — the generator this pipeline grounds
- [transformers](#) — the encoder behind bi- and cross-encoders
- [tokenisers](#) — how documents become token sequences before embedding
- [fine tuning](#) — the alternative for injecting knowledge into weights