RAG & Retrievers

Complete Deep-Dive Knowledge Base

Everything you need to know about Retrieval-Augmented Generation:

concepts, internals, retriever types, why it works, and when to use each approach

Comprehensive Edition — 2024

Chapter 1: What is Retrieval-Augmented Generation (RAG)?

1.1 The Core Idea — and Why It Exists

Large Language Models (LLMs) are trained on enormous corpora of text. During training they compress knowledge into billions of numerical parameters — but that knowledge is frozen at a point in time. The model cannot look things up, cannot know what happened after its training cutoff, and cannot access private documents it was never trained on.

Retrieval-Augmented Generation (RAG) solves these problems by giving the model a dynamic memory: at inference time, before the model answers a question, a retrieval system finds relevant documents and hands them directly to the model inside its prompt. The model then generates its response based on both its learned knowledge and the freshly-retrieved evidence.

> **One-sentence definition**
> RAG = Retrieval + Augmentation + Generation: retrieve relevant documents, augment the model prompt with them, then generate a grounded answer.

1.2 Why Not Just Fine-Tune the Model?

A common question: why not simply train the model on your private data? Fine-tuning injects knowledge into the model's weights. RAG keeps knowledge external. The table below contrasts the two approaches:

Dimension	Fine-Tuning	RAG
Knowledge update	Requires retraining (hours to weeks)	Update the knowledge base (seconds to minutes)
Knowledge location	Baked into model weights	Stored externally; retrieved at inference
Cost	High GPU cost for training	Low: storage + retrieval compute
Transparency	Black box — cannot cite sources	Can return exact source documents
Accuracy risk	Can catastrophically forget old knowledge	Retrieval quality determines answer quality
Data privacy	Training data may leak through model outputs	Data stays in your controlled database
Freshness	Stale once training ends	Live: update the DB and answers change instantly

1.3 The Hallucination Problem — and Why RAG Helps

LLMs are trained to produce fluent, plausible text — not necessarily true text. When a model is asked about something it does not know well, it often generates confident-sounding but incorrect information. This is called hallucination.

Hallucinations arise because:

- The model learns statistical correlations between tokens, not verified facts.
- The model has no way to say 'I don't know' natively — it always produces something.
- Rare or recent facts are underrepresented in training data, so predictions about them are unreliable.

RAG mitigates hallucination by changing the task. Instead of 'generate an answer from memory,' the model is doing 'read these documents and extract an answer.' Grounding the generation in retrieved text dramatically reduces confabulation — especially for factual, domain-specific questions.

Why hallucinations cannot be eliminated by RAG alone

> If the retrieval step fails (returns irrelevant documents), the model may still hallucinate based on those bad documents. RAG shifts hallucination risk from 'model memory failure' to 'retrieval quality failure' — which is why retriever design is so critical.

1.4 RAG Architecture: All Six Components Explained

A complete RAG system has six components that work in a pipeline. Understanding each one is essential to debugging and optimizing the overall system.

1.4.1 Knowledge Base

The knowledge base is the raw source of truth — a collection of documents, PDFs, web pages, database tables, or any text that should inform the model's answers. The quality, coverage, and freshness of the knowledge base directly caps the quality of every answer the system can produce.

Why it matters: Garbage in, garbage out. A retriever can only return what exists in the knowledge base. If the answer to a question is not in any document, no amount of retrieval sophistication will help.

1.4.2 Chunking

Documents are too large to embed whole. Chunking splits documents into smaller segments — typically 256 to 1024 tokens — that can be embedded and stored individually. This is often the most underestimated step in RAG.

Why chunking size matters:

- Chunks too small: A single chunk may lack enough context to be useful. The retrieved snippet might be half a sentence.
- Chunks too large: Embedding a 2000-token chunk compresses too much information into a single vector, making the embedding less precise for retrieval.
- Chunk boundaries: A sentence split across a boundary loses meaning at both ends.

Common strategies include fixed-size with overlap (overlap of 10-20% preserves context at boundaries), sentence-aware splitting, paragraph-aware splitting, and semantic chunking (splitting at topic shifts detected by a model).

1.4.3 Embedding Model

An embedding model converts a piece of text into a dense numerical vector — a list of floating-point numbers, typically 768 to 3072 dimensions. The key property: semantically similar texts map to nearby vectors in this high-dimensional space.

Why this matters: The entire premise of dense retrieval is that **similar meaning produces similar vectors**. If the embedding model is weak or domain-mismatched, semantically related documents will not cluster together, and retrieval fails silently.

Embedding models are trained using contrastive learning: they are shown pairs of (query, relevant_document) and trained to minimize the distance between their vectors, while maximizing distance from irrelevant documents.

1.4.4 Vector Database

A vector database stores document embeddings and provides efficient similarity search. Given a query vector, it returns the most similar document vectors. The core algorithmic challenge is Approximate Nearest Neighbor (ANN) search: exact nearest-neighbor in high dimensions is prohibitively slow, so ANN algorithms trade tiny accuracy losses for dramatic speed gains.

Key ANN algorithms:

- HNSW (Hierarchical Navigable Small World): Builds a multi-layer proximity graph. Dominant in most modern vector DBs. Excellent recall/speed tradeoff.
- IVF (Inverted File Index): Clusters vectors into regions, searches only nearby clusters. Fast but slightly lower recall than HNSW.
- ScaNN (Google): Combines quantization and partition-based search. State-of-the-art performance at massive scale.

Popular vector databases include Pinecone (managed cloud), Weaviate (open-source with hybrid search), ChromaDB (lightweight, local), FAISS (Facebook's library), and Qdrant (Rust-based, payload filtering).

1.4.5 Retriever

The retriever is the query-time engine that converts a user question into a query embedding, searches the vector database, and returns the top-k most relevant chunks. The retriever's job is precision and recall: return the chunks that are both relevant and ranked correctly.

Chapter 3 covers every retriever type in depth. The retriever is where most of the engineering decisions in RAG live.

1.4.6 Language Model (Generator)

The LLM is the final step. It receives a prompt that includes: (a) the original user question, (b) the retrieved document chunks, and (c) any system instructions. It generates an answer grounded in that context.

The quality of the generation step depends on: context window size (can all retrieved chunks fit?), instruction following ability (does the model follow the 'answer only from context' instruction?), and reasoning capability (can it synthesize across multiple retrieved chunks?).

1.5 The Full RAG Pipeline: Step by Step

Step 1: Indexing (offline)	Documents are chunked, embedded, and stored in the vector database. This happens once (or periodically) before any queries arrive.
Step 2: Query receipt	A user submits a natural language question.
Step 3: Query embedding	The question is passed through the same embedding model used during indexing, producing a query vector.
Step 4: Vector search	The query vector is sent to the vector database, which runs ANN search and returns the top-k most similar chunk vectors (and their text).
Step 5: Re-ranking (optional)	A cross-encoder re-ranker evaluates each chunk against the query and re-orders them by relevance. Expensive but highly accurate.
Step 6: Context assembly	The top-n chunks are formatted into a readable context block, including source metadata if available.
Step 7: Prompt construction	The system prompt, user question, and retrieved context are assembled into a complete prompt.
Step 8: Generation	The LLM reads the prompt and generates an answer grounded in the retrieved documents.
Step 9: Source attribution (optional)	The system returns citations pointing to the source documents used.

Chapter 2: Core Concepts — Everything Explained

2.1 Embeddings: What They Are and How They Work

An embedding is a mathematical representation of meaning as a point in high-dimensional space. The idea originates from the distributional hypothesis in linguistics: words that appear in similar contexts have similar meanings.

Modern embedding models (based on transformers like BERT or RoBERTa) extend this idea to full sentences and documents. The model reads the entire text, uses self-attention to model relationships between all words simultaneously, and outputs a fixed-size vector that represents the semantic content of the entire passage.

2.1.1 How Embeddings Capture Meaning

The embedding space has remarkable geometric properties. Semantically related concepts cluster together. You can perform arithmetic: the vector for 'king' minus 'man' plus 'woman' approximates the vector for 'queen'. Queries and their relevant documents land near each other in the same space — this is what makes retrieval work.

Embedding models are trained using contrastive loss. The training data consists of (anchor, positive, negative) triplets — where anchor and positive are semantically related, and negative is unrelated. The model learns to minimize distance between anchor and positive vectors while maximizing distance to negative vectors.

2.1.2 Bi-Encoder vs. Cross-Encoder Architecture

There are two fundamental architectures for computing relevance between a query and a document:

Bi-Encoder	Query and document are encoded SEPARATELY by the same model, producing independent vectors. Similarity is computed as a dot product or cosine. Fast: document vectors are pre-computed offline. Used for retrieval (first-stage recall).
Cross-Encoder	Query and document are fed TOGETHER into the model as a single concatenated input. The model attends across both, producing a single relevance score. Slow: cannot pre-compute. Used for re-ranking (second-stage precision).

Why bi-encoders for retrieval: you have millions of documents. Cross-encoding every document with every query at query time is computationally infeasible. Bi-encoders allow all documents to be encoded offline, and only the query is encoded at runtime.

Why cross-encoders for re-ranking: you only need to score the top 20-100 documents the bi-encoder returned. At that scale, the slower cross-encoder is affordable and its joint attention makes it significantly more accurate.

2.2 Vector Similarity: Cosine, Dot Product, and Euclidean Distance

Once embeddings are computed, similarity is measured mathematically. Three metrics are commonly used:

Cosine Similarity	Measures the angle between two vectors, ignoring magnitude. Range: -1 to 1. 1 = identical direction (most similar). Preferred when vector magnitude varies (common with text embeddings).
Dot Product	Multiply corresponding dimensions and sum. Equivalent to cosine when vectors are unit-normalized. Faster to compute. Preferred when embeddings are L2-normalized (as in OpenAI's embeddings).
Euclidean Distance (L2)	Straight-line distance between two points. Sensitive to magnitude. Less commonly used for semantic similarity but works for some embedding types.

> **Practical note**
> Most embedding model providers recommend a specific metric — use theirs. OpenAI text-embedding-3 models use dot product on normalized vectors. Sentence Transformers models typically use cosine. Using the wrong metric can hurt retrieval quality significantly.

2.3 Context Window: The LLM's Working Memory

Every LLM has a context window — a hard limit on how many tokens it can process in a single inference call. Tokens are roughly word-pieces: 'embedding' is one token, 'unbelievable' might be two. 1000 tokens is approximately 750 words.

The context window is critical for RAG because retrieved chunks must fit inside it alongside the system prompt and user question. If you retrieve 10 chunks of 512 tokens each (5120 tokens), plus a 500-token system prompt and 100-token question, you need a model with at least a 5720-token context.

2.3.1 The 'Lost in the Middle' Problem

Research has shown that LLMs perform better when relevant information appears at the beginning or end of a long context — not the middle. When you pass many retrieved chunks, the model may effectively ignore evidence buried deep in the context.

Mitigation strategies: re-rank retrieved chunks by relevance before assembling the context, put the most relevant chunk first, limit the number of chunks passed to the LLM, or use a model specifically trained to handle long contexts uniformly.

2.4 BM25: How Keyword Search Really Works

BM25 (Best Matching 25) is the dominant sparse retrieval algorithm. Despite being decades old, it remains competitive with dense retrievers for many tasks — and it requires no GPU or training.

2.4.1 The BM25 Formula

BM25 scores document D for query Q by summing, for each query term t: `IDF(t) × [TF(t,D) × (k1+1)] / [TF(t,D) + k1 × (1 − b + b × |D|/avgdl)]`

Where:

- TF(t, D) = term frequency: how many times term t appears in document D
- IDF(t) = inverse document frequency: log((N - df + 0.5) / (df + 0.5) + 1), where N is total documents and df is documents containing t. Rare terms get higher IDF — they're more discriminating.
- |D| = length of document D in words
- avgdl = average document length across the corpus
- k1 = term frequency saturation parameter (typically 1.2-2.0). Controls diminishing returns on repeated terms.
- b = length normalization parameter (typically 0.75). At b=1, fully normalize for length; at b=0, ignore length.

Why BM25 beats TF-IDF: Raw TF-IDF grows unboundedly with term frequency — a document mentioning 'python' 100 times scores 10x better than one mentioning it 10 times, even if the extra repetitions are redundant. BM25's k1 parameter introduces saturation: after a point, more occurrences of a term add diminishing score. It also normalizes for document length, so a long document doesn't automatically score higher just for containing more terms.

2.5 Approximate Nearest Neighbor (ANN) Search

Exact nearest-neighbor search — finding the single closest vector in a database of 10 million embeddings — requires comparing the query to every vector. At 768 dimensions and 10M vectors, this is 7.68 billion floating-point operations per query. That is too slow for production.

ANN algorithms sacrifice tiny amounts of recall (perhaps 1-5%) for dramatic speed improvements (100x-1000x). The key ANN method used in most vector databases is HNSW:

2.5.1 HNSW (Hierarchical Navigable Small World)

HNSW builds a multi-layer graph structure. At the top layer, only a sparse set of nodes exist — each connected to a few nearby neighbors. Lower layers are denser. Search starts at the top, greedily navigating toward the query vector, then descends to denser layers for finer search.

This is analogous to a highway system: to travel from New York to Los Angeles, you start on the interstate (top layer), take smaller highways as you near your destination, and finally take local roads for the last mile.

Key parameters: M (number of connections per node — higher M means better recall but more memory), efConstruction (size of the candidate set during build — higher means better quality index but slower build), ef (size of the candidate set during search — tune for recall/speed tradeoff at query time).

2.6 Reciprocal Rank Fusion (RRF)

When combining results from multiple retrievers (hybrid search), you need a way to merge ranked lists from different scoring scales. You cannot simply add scores: BM25 scores and cosine similarity scores have completely different ranges and meanings.

RRF solves this by using only rank positions, not raw scores: `RRF(d) = Σ 1 / (k + rank_i(d))` where k is typically 60 and rank_i(d) is document d's rank in retriever i's results.

Why k=60: This constant prevents the top-ranked document from being too dominant (1/(1+0)=1 vs 1/(60+1)≈0.016). The k=60 value was empirically found to work well across many fusion tasks in the original RRF paper.

RRF has several advantages: it is robust (rank-based, not score-based), it does not require normalization or parameter tuning per query, and it degrades gracefully when one retriever returns no results for a given document.

Chapter 3: Every Retriever Type — In Depth

3.1 Dense Retrievers (Semantic / Vector Search)

What it is

Dense retrieval uses learned neural embeddings to map both queries and documents into a shared vector space. Retrieval is performed by finding document vectors closest to the query vector using ANN search.

Why it exists

Keyword search cannot handle synonyms, paraphrasing, or conceptual similarity. If a document says 'cardiac arrest' and the query asks about 'heart attack,' a keyword search finds nothing. Dense retrieval captures semantics: both phrases land near each other in vector space, so the document is retrieved correctly.

How it works internally

1. Both query and all documents are encoded by a bi-encoder model (typically a fine-tuned BERT or Sentence Transformer variant).
2. Document vectors are pre-computed and stored in the vector database at indexing time.
3. At query time, the query is encoded (one forward pass through the embedding model).
4. ANN search (HNSW, IVF, etc.) finds the k document vectors with highest cosine/dot-product similarity.
5. The corresponding document texts are returned as retrieval results.

Strengths and Weaknesses

Semantic understanding	Handles synonyms, paraphrasing, cross-lingual queries, and conceptual similarity.
Training required	The embedding model must be trained or fine-tuned on in-domain data for best results.
Rare entities	Specific model numbers, names, codes — the model may not have learned their embeddings well.
Black box	Hard to debug why a specific document was or was not retrieved.
Latency	10-100ms depending on database size and ANN parameters.

When to use it

Use dense retrieval as your default starting point for most RAG applications. It excels at question answering, conversational search, and any domain where users phrase things variably. Add hybrid or sparse retrieval when you observe failures on exact-term queries.

3.2 Sparse Retrievers (BM25 / Keyword Search)

What it is

Sparse retrieval represents documents and queries as high-dimensional sparse vectors where most values are zero. Each dimension corresponds to a term in the vocabulary. Retrieval finds documents with high term-overlap with the query.

Why it exists

Dense retrieval is powerful but computationally expensive, requires a trained model, and struggles with exact matches. For decades before neural search, BM25 was (and in many production systems still is) the best retrieval algorithm available. It remains competitive or superior in specific scenarios.

Why BM25 works so well for exact matches

When a user searches for 'transformer architecture layer normalization placement,' the key terms are highly specific. If a document contains those exact words, it is almost certainly relevant. Dense retrieval might embed this query near documents about 'deep learning normalization techniques' — semantically related but not an exact match. BM25 finds the document with those exact terms first.

TF-IDF vs. BM25 — Why BM25 Wins

TF-IDF scores grow linearly with term frequency: mentioning a word 100 times makes a document score 100x higher for that word. In practice, this rewards document length and repetitive text. BM25 introduces two fixes:

- Saturation: Term frequency is capped via the k1 parameter. Going from 1 mention to 2 mentions helps a lot; going from 100 to 101 mentions helps almost nothing.
- Length normalization: A short document mentioning a term is penalized less than a long document mentioning it the same number of times, controlled by the b parameter.

When to use it

Code search, legal search, medical terminology, technical documentation with model numbers and identifiers, any domain where exact term matching is critical. Always include BM25 as one leg of a hybrid retrieval system.

3.3 Hybrid Retrievers

What it is

Hybrid retrieval runs both sparse (BM25) and dense (vector) retrieval simultaneously, then combines their ranked result lists using a fusion algorithm (typically RRF).

Why it exists — the fundamental tradeoff

Dense retrieval excels at semantic similarity but misses exact matches. Sparse retrieval excels at exact matches but misses paraphrasing. Real-world queries span both needs. A user searching for 'error code E404 in the payment gateway' needs exact matching on 'E404' and semantic matching on 'payment gateway issues.'

Hybrid retrieval consistently outperforms either method alone across diverse query types and datasets. It is the recommended architecture for production RAG systems.

Fusion Algorithms Compared

Algorithm	How it works	When to use
RRF (Reciprocal Rank Fusion)	Combines rank positions only: score = Σ 1/(k + rank). No score normalization needed.	Default choice. Robust and parameter-free.
Weighted Linear Combination	score = α × sparse_score + (1-α) × dense_score. Requires score normalization.	When you have labeled data to tune α per query type.
Learned Fusion	A trained model learns optimal combination weights, possibly per query.	High-value production systems with training data.

3.4 Multi-Vector Retrievers (ColBERT)

What it is

Instead of encoding an entire document into a single embedding vector, multi-vector retrievers encode each token (or sentence) into its own vector. A document with 100 tokens produces 100 vectors.

Why single-vector retrieval falls short

A single vector must compress an entire document's meaning. For a 500-token chunk, this is a significant lossy compression. If the query matches only one specific sentence in the middle of the chunk, the single vector may not capture that signal — it's diluted by all the other content.

ColBERT: How MaxSim works

ColBERT (Contextualized Late Interaction over BERT) encodes the query into token vectors q1...qm and the document into token vectors d1...dn. The relevance score is: `Score(Q, D) = Σ_i max_j(qi · dj)` — for each query token, find the most similar document token, then sum those maximum similarities.

This MaxSim operation is powerful because it does not require query and document to match globally — it finds the best local alignment for each query token. A query asking about 'inference speed optimization' will score highly against a document that discusses these concepts in different paragraphs, because each query token finds its closest document token match independently.

The efficiency trick

ColBERT's 'late interaction' means document vectors are pre-computed at indexing time (just like a bi-encoder). The expensive part — computing MaxSim — only happens for the top candidates retrieved by a fast first-stage filter. This makes ColBERT more accurate than bi-encoders while remaining scalable.

3.5 Re-Rankers (Cross-Encoders)

What it is

A re-ranker is a second-stage precision filter. After a fast retriever (dense, sparse, or hybrid) retrieves the top-k candidates (typically k=20 to 100), the re-ranker scores each candidate individually using a slower, more accurate cross-encoder model and re-orders the results.

Why retrieval alone is insufficient for high-stakes applications

Bi-encoders and BM25 are optimized for recall — they retrieve most of the relevant documents. But they are not optimized for precision — the top-1 result may not be the most relevant. For applications like legal research, medical Q&A, or compliance systems, you need the most relevant document at the top, not just somewhere in the top 50.

Why cross-encoders are more accurate

A cross-encoder concatenates the query and document and processes them jointly through a transformer. The self-attention mechanism can now attend between every token in the query and every token in the document simultaneously. This full interaction captures relevance signals that bi-encoders miss — nuanced context dependencies, negations, and subtle topic shifts.

The cost: you cannot pre-compute cross-encoder scores. Every query requires a separate forward pass for each candidate. At 50 candidates and 200ms per pass, that is 10 seconds — unacceptable as a first-stage retriever but perfectly acceptable as a second-stage filter over 50 candidates.

3.6 Contextual Retrieval

What it is

Contextual retrieval prepends LLM-generated context to each chunk before embedding it. The context summarizes where the chunk comes from within the larger document, giving the embedding more informative signal.

The problem it solves

Consider a chunk that reads: 'The results showed a 23% improvement over the baseline.' Without context, this chunk's embedding captures the idea of 'improvement over baseline' — but does not know this is from Section 4 of a machine learning paper about attention mechanisms. When a user asks

'what was the performance gain of the attention mechanism,' a bare embedding of this chunk may not rank highly.

With contextual retrieval, the chunk is preceded by LLM-generated text like: 'This chunk is from a paper on transformer attention mechanisms, Section 4 (Results). It reports the performance comparison against baseline methods.' Now the embedding captures the relevant context, dramatically improving retrieval.

How to generate context efficiently

Generating context for every chunk is expensive if done naively. Anthropic's implementation uses prompt caching to re-use the full document context across all chunk context generation calls, reducing the cost by 80-90%.

3.7 Parent Document Retriever

What it is

The parent document retriever maintains two different granularities simultaneously: small chunks for embedding and retrieval, and larger parent chunks (or full documents) for passing to the LLM.

The precision-context tradeoff

Small chunks embed well — a 64-token chunk cleanly represents one idea, so its embedding is precise. But a 64-token chunk passed to the LLM often lacks the surrounding context needed to answer a question well.

Large chunks embed poorly — a 1024-token chunk contains many ideas, making its embedding a blurry average. But a 1024-token chunk gives the LLM rich context to work with.

The parent document retriever gets both benefits: embed small chunks for precise retrieval, but when a small chunk matches, retrieve its parent (the large enclosing section or document) for the LLM to use.

3.8 Self-Query Retriever

What it is

The self-query retriever uses an LLM to parse a natural language query into two parts: a semantic search component (text to embed and search) and metadata filters (structured attributes to filter the vector database by).

Why metadata filtering matters

Vector similarity captures semantic relevance but ignores structured attributes. If a user asks 'find papers about transformers published after 2022,' vector search can find relevant transformer papers — but it cannot filter by year. The self-query retriever extracts the filter year > 2022 and applies it as a database-level filter before or during vector search.

Example decomposition: Query: `'Papers about transformers from 2022 by researchers at DeepMind'` → semantic: `'transformer architecture'` + filters: `{year: {$gte: 2022}, institution: 'DeepMind'}`

This dramatically narrows the search space and eliminates false positives that are semantically similar but structurally wrong.

3.9 Multi-Query Retriever

What it is

An LLM generates 3-5 alternative phrasings of the original query. Each phrasing is used for independent retrieval. All results are combined and deduplicated.

Why single-query retrieval misses relevant documents

Embedding models are sensitive to phrasing. 'What causes inflation?' and 'What are the drivers of rising prices?' are semantically similar to a human but may produce different query embeddings and retrieve different documents. A relevant document phrased in one way may not rank highly for a query phrased another way.

Multi-query retrieval exploits query diversity: by generating multiple phrasings, you increase the probability that each relevant document is retrieved by at least one of the query variants.

3.10 Graph RAG / Knowledge Graph Retriever

What it is

Graph RAG combines vector-based semantic retrieval with graph-based relationship traversal. Documents are processed to extract entities and their relationships, forming a knowledge graph. Retrieval can follow relationship edges to find related information that vector search would miss.

Why flat vector search misses relationship information

Consider the query: 'Which companies invested in firms that were acquired by Google?' Vector search might find documents about Google acquisitions, and separately find documents about investment rounds — but it cannot reason about the relationship chain: investor → investee → acquirer. A knowledge graph traversal can.

Multi-hop reasoning

Many complex questions require multi-hop reasoning: finding answer A, then using A to find answer B. In a knowledge graph, this is a path traversal: start at the query entity node, traverse k hops along relevant edges, collect all nodes encountered. These nodes become the retrieval results.

Microsoft's GraphRAG implementation first partitions the document corpus into communities of related entities using graph algorithms, then generates community summaries. These summaries enable 'macro' queries that span many documents — queries like 'what are the major themes in this document collection' that defeat traditional retrieval entirely.

3.11 Time-Weighted Retriever

What it is

A time-weighted retriever modifies retrieval scores by a recency factor. Documents are scored as: final_score = semantic_similarity × time_decay_factor, where documents published more recently receive higher time_decay values.

The decay function

A common decay function: `decay = (1 - decay_rate) ^ hours_elapsed`. With decay_rate=0.01 and a document 1 hour old, decay≈0.99. With the same document 72 hours old, decay≈0.48. The decay_rate is tuned per domain.

When recency matters

For customer support about software (version updates change answers), news retrieval (recent articles are more relevant), regulatory compliance (the newest regulation supersedes older ones), and conversation memory (recent messages in a chat are more relevant context), recency is a meaningful relevance signal that pure semantic similarity ignores.

3.12 Long Context Retriever (No Retrieval)

What it is

With LLMs now supporting context windows of 100K to 2M tokens, an alternative to retrieval is to simply include the entire knowledge base in the prompt. The LLM 'retrieves' information through its attention mechanism, not through a separate retrieval step.

When it makes sense

For small knowledge bases (under ~50 documents), frequently changing content where re-indexing is expensive, or very high-accuracy requirements where you cannot risk retrieval failures, long context can be simpler and more accurate than RAG.

Why it is not always the answer

Cost scales linearly with context size. At $15/M tokens, passing 500K tokens on every query costs $7.50 per query — economically infeasible at scale. Latency increases with context length. The 'lost in the middle' problem means the LLM may miss evidence deep in a very long context.

Chapter 4: Advanced RAG Patterns

4.1 Query Transformation

Query transformation modifies the user's raw query before retrieval to improve retrieval quality. The raw query is often imperfect: it may be ambiguous, too short, or assume context the retriever lacks.

Query Expansion

Add related terms or synonyms to the query. 'MI' might be expanded to 'MI myocardial infarction heart attack cardiac event' before retrieval. This increases the probability of matching relevant documents that use different terminology.

Hypothetical Document Embedding (HyDE)

Instead of embedding the query directly, use the LLM to generate a hypothetical document that would answer the query, then embed that hypothetical document. The idea: the hypothetical answer lives in the same embedding space as real documents, producing a better retrieval vector than the original short query.

Example: Query: 'What is the half-life of aspirin?' → LLM generates: 'Aspirin has a plasma half-life of approximately 15-20 minutes, though its effects last much longer...' This document-length text produces an embedding that more precisely locates relevant pharmacology documents.

Step-Back Prompting

For narrow or overly specific queries, the LLM first generates a higher-level, more general version of the question. Retrieval is done on both the original and the stepped-back query. This helps when the direct answer is not in the knowledge base but can be inferred from higher-level concepts.

4.2 Chunking Strategies — The Most Impactful Engineering Decision

How you split documents before embedding is arguably the most impactful — and most overlooked — decision in RAG. Chunk size and strategy directly determine what information is retrievable.

Fixed-size chunking	Split every N tokens. Simple and fast. Risk: splits mid-sentence or mid-concept. Use overlap (10-20% of chunk size) to mitigate boundary losses.
Sentence-aware chunking	Split at sentence boundaries only. Preserves grammatical units. Better coherence per chunk. More variable chunk sizes.
Semantic chunking	Use an embedding model to detect topic shifts. Split when cosine similarity between adjacent sentences drops below a threshold. Creates thematically coherent chunks but is computationally expensive.
Recursive chunking	Try to split on paragraphs first, then sentences, then words, until target size is met. Preserves structure hierarchy while controlling size.
Document-structure-aware	Use document headers (markdown ## headings, HTML <h2> tags, PDF section headers) as split points. Chunks align with actual document sections.

> **Overlap is not optional**
> When splitting on fixed sizes, always add 10-20% overlap between adjacent chunks. A key sentence spanning a chunk boundary would otherwise be fragmented across two partial chunks, making neither one retrievable for queries about that sentence.

4.3 Agentic RAG

Agentic RAG extends basic RAG by giving the system agency: the ability to decide what to retrieve next, whether retrieved information is sufficient, and how to decompose complex questions across multiple retrieval steps.

ReAct (Reason + Act)

The LLM alternates between reasoning steps and retrieval actions. It reasons about what it needs to know, retrieves that information, reasons about what it still needs, retrieves again, and so on until it has enough information to answer.

Example for 'Compare the revenue of Apple and Microsoft in their most recent fiscal years': Step 1: retrieve Apple Q4 2024 revenue. Step 2: retrieve Microsoft fiscal 2024 revenue. Step 3: synthesize comparison. Each step is a retrieval action, and the LLM plans each one based on what it learned from the previous.

FLARE (Forward-Looking Active Retrieval)

During generation, if the LLM begins to generate a sentence with low confidence (high entropy in token probabilities), it pauses and retrieves relevant information before continuing. This way, retrieval is triggered only when needed — for uncertain claims, not routine statements.

4.4 Evaluation Metrics — Measuring What Matters

A RAG system has two distinct quality dimensions to measure: retrieval quality and generation quality. Both must be evaluated independently.

Recall@K	Of all relevant documents in the knowledge base, what fraction appear in the top-K retrieved results? Measures retrieval completeness. Critical if missing any relevant document is costly.
Precision@K	Of the K retrieved documents, what fraction are actually relevant? Measures retrieval efficiency. Important when the LLM context window limits how many chunks you can pass.
MRR (Mean Reciprocal Rank)	Average of 1/rank_of_first_relevant_doc across queries. If the first relevant doc is rank 1, MRR=1. If rank 2, MRR=0.5. Measures how soon relevant results appear.
NDCG (Normalized Discounted Cumulative Gain)	Measures ranking quality with graded relevance. A highly relevant doc ranked lower is penalized more than a slightly relevant doc ranked lower.
Faithfulness	Does the generated answer contain only information that is present in the retrieved context? Measures hallucination — answers making claims not supported by retrieved docs.
Answer Relevance	Does the generated answer actually address the user's question? A faithful but off-topic answer is not useful.
Context Recall	Was all information needed to answer the question present in the retrieved context? If not, the answer must have been hallucinated or fabricated from model memory.

4.5 Production Considerations

Caching

For high-traffic systems, cache the embedding of frequently-asked queries and their top-k results. Even a 1-hour cache can dramatically reduce vector database load and API costs for popular questions.

Asynchronous Indexing

Document indexing (chunking, embedding, vector DB insertion) should happen asynchronously and not block query serving. Use a queue (Kafka, SQS) to decouple document ingestion from retrieval.

Index Freshness

Define an SLA for how quickly newly added documents should be available for retrieval. Incremental indexing (adding new vectors without reindexing everything) is supported by all major vector DBs.

Guardrails and Hallucination Detection

For high-stakes applications, add a faithfulness check after generation: verify that each claim in the generated answer is supported by at least one retrieved chunk. This can be done with a fine-tuned NLI (Natural Language Inference) model or another LLM call.

Chapter 5: How to Choose the Right Retriever

5.1 Decision Framework

The right retriever depends on four factors: your data characteristics, query characteristics, accuracy and latency requirements, and system complexity budget.

Your Situation	Primary Need	Recommended Retriever
Starting a new project	Simplicity + baseline	Dense Retriever (all-MiniLM or text-embedding-3-small)
Need high accuracy in production	Precision + recall	Hybrid (BM25 + Dense) + Cross-Encoder Re-ranker

Your Situation	Primary Need	Recommended Retriever
Code or technical docs with identifiers	Exact term matching	Sparse BM25 (or Hybrid)
Legal, compliance, contracts	Precision + citations	Hybrid + Re-ranker + Source attribution
Data with rich metadata (date, author, type)	Structured filtering	Self-Query Retriever
Complex entity relationships	Multi-hop reasoning	Graph RAG (GraphRAG or similar)
Time-sensitive content (news, updates)	Recency weighting	Time-Weighted Retriever
Very small knowledge base (<50 docs)	Maximum simplicity	Long Context (no retrieval)
Ambiguous or broad queries	High recall	Multi-Query Retriever
Long documents (books, reports)	Granular retrieval + broad context	Parent Document Retriever

5.2 The Iterative Improvement Loop

No retriever choice is final. RAG systems require continuous monitoring and iteration. The recommended process:

6. Start simple: deploy dense retrieval with a good embedding model.
7. Instrument: log every query, every retrieved chunk, and whether the generated answer was marked helpful or not.
8. Analyze failure modes: categorize bad answers — is the failure in retrieval (wrong chunks returned) or generation (right chunks, wrong synthesis)?
9. Add complexity where needed: if retrieval is failing, add re-ranking or switch to hybrid. If generation is failing, improve prompting or add a faithfulness check.
10. Measure: always A/B test changes with offline evaluation metrics before deploying.

5.3 Common Pitfalls and How to Avoid Them

Pitfall: Chunks too large	Each chunk covers many topics, making embeddings imprecise. Fix: reduce chunk size to 256-512 tokens for most use cases.
Pitfall: No overlap between chunks	Content at boundaries is split across two partial chunks, making neither retrievable for queries about that content. Fix: add 10-20% overlap.
Pitfall: Wrong embedding model	Using a general model for a domain-specific corpus. Fix: fine-tune or select a domain-adapted model (e.g., medical, legal, code-specific).

Pitfall: k too small	Retrieving only top-3 chunks. If any relevant doc is ranked 4th, it is missed. Fix: retrieve top-10 to top-20, re-rank, then pass the top-5 to the LLM.
Pitfall: No re-ranker in production	Bi-encoder ranking quality is good but not great. Fix: always add a cross-encoder re-ranker for user-facing applications.
Pitfall: Not filtering by metadata	Semantic search ignores structured attributes like date, category, or author. Fix: use metadata filtering or a self-query retriever.
Pitfall: No evaluation pipeline	Launching without measuring retrieval quality. Fix: build a golden question set with known relevant documents before launch.

Chapter 6: When NOT to Use RAG — Every Case

RAG is a powerful tool, but it is not the right solution for every problem. Applying RAG where it does not fit introduces unnecessary complexity, latency, cost, and new failure modes — without meaningful benefit. This chapter catalogues every situation where RAG is the wrong choice, explains precisely why, and provides the better alternative in each case.

The core principle

RAG is justified only when (a) the information needed to answer a question exists in documents, AND (b) those documents cannot fit in the context window, AND (c) the cost or latency of full-context is unacceptable. Violate any of these conditions and RAG may be unnecessary or harmful.

6.1 Knowledge Already in the LLM's Training Data

If the information required to answer a question is well-covered in the model's training data — widely documented facts, established science, general programming knowledge, historical events — RAG adds cost and latency without improving accuracy.

Why RAG can actually hurt here

Retrieval may return documents that are subtly different from the most accurate answer, or documents with errors. The LLM might defer to a flawed retrieved document rather than its more reliable trained knowledge. For well-established facts, model knowledge is often more accurate than any single retrieved document.

Examples of questions where RAG is unnecessary:

- 'What is the time complexity of quicksort?' — universally documented, model knows this reliably.
- 'Explain Newton's second law of motion.' — foundational physics, no retrieval needed.
- 'Write a Python function to sort a list.' — coding knowledge is baked into the model.
- 'What year did World War II end?' — historical facts, model is reliable.

Better alternative

Prompt the LLM directly. No retrieval needed. Use RAG only for domain-specific, private, or recent information the model was not trained on.

6.2 The Knowledge Base Fits Entirely in the Context Window

If your entire knowledge base — every document you want the model to be aware of — fits within the LLM's context window, you should simply include all of it in the prompt rather than building a retrieval system.

Modern LLMs support context windows of 128K to 2M tokens. At roughly 750 words per 1000 tokens, a 128K context window holds approximately 96,000 words — a 350-page book. If your knowledge base is smaller than this, retrieval infrastructure is unnecessary overhead.

Why RAG is wrong here

Every retrieval step has a chance of failure: the right document might not be retrieved, or might be ranked too low. When the entire knowledge base fits in context, you eliminate retrieval failure entirely. The model sees everything and uses its attention mechanism to find what is relevant — perfectly and without missing anything.

Signs you are in this situation:

- Your knowledge base is a single document or a handful of documents.
- Your total document corpus is under ~50,000-100,000 words.
- You are building a prototype or internal tool with a fixed, small document set.

Better alternative

Long context prompting: pass all documents directly in the prompt. No indexing, no vector database, no retrieval pipeline. Simpler, cheaper for small corpora, and eliminates retrieval failure modes entirely.

6.3 Real-Time or Streaming Data Requirements

RAG requires documents to be indexed before they can be retrieved. There is always a delay between when a document is created and when it is available for retrieval — the indexing lag. For applications that require truly real-time information (milliseconds-fresh), RAG is the wrong architecture.

The indexing lag problem

A new document must be: chunked → embedded (LLM API call) → inserted into the vector database. This pipeline typically takes 2-30 seconds per document. During a fast-moving event (a live market price change, a live sports score, a new error log line), the 'latest' information in your vector DB is already stale.

Scenarios where RAG fails on recency:

- Live stock price queries: 'What is Apple's current stock price?' — No indexed document is fresh enough.
- Real-time sensor or IoT data monitoring.
- Live sports scores or event outcomes.
- Current weather conditions.
- Live system status or incident monitoring.

Better alternatives

Direct API calls (weather API, financial data API, sports API) injected into the prompt at query time. Structured database queries for live operational data. Tool use / function calling: give the LLM a tool to call a live API directly rather than retrieving from a stale index.

6.4 Pure Reasoning, Math, or Logic Tasks

Tasks that require computation, formal reasoning, mathematical proof, code execution, or logical deduction are not retrieval problems. The answer does not exist in any document — it must be derived.

Why retrieval cannot help with reasoning

Retrieval finds documents similar to your query. If you ask 'What is 17 × 384 + the square root of 144?', there is no document that contains this specific calculation. Even if a retrieved document explains arithmetic, the LLM still needs to perform the computation. Retrieval adds no value and may distract the model with irrelevant 'similar' math examples.

Task types where RAG adds nothing:

- Mathematical calculations and proofs.
- Formal logic problems and puzzles.
- Code execution and debugging (unless looking up docs).

- Data analysis over provided structured data (use code interpreter instead).
- Step-by-step reasoning over a problem given entirely in the prompt.
- Translation between languages.
- Text summarization when the document is already in the prompt.

Better alternatives

Chain-of-thought prompting for reasoning. Code interpreter / tool use for computation and data analysis. Structured output prompting for logic tasks.

6.5 High-Latency Sensitivity with No Accuracy Benefit

RAG adds latency to every response: the query must be embedded, the vector database must be searched, results must be fetched and re-ranked, and a longer prompt (with retrieved context) takes longer to generate. In total, RAG typically adds 200ms to 2+ seconds of latency per query.

For latency-critical applications where the knowledge required is either in-model or real-time (via API), this added latency provides zero benefit and actively degrades user experience.

RAG latency breakdown

Embedding the query: ~20-50ms. Vector ANN search: ~10-100ms. Re-ranking (if used): ~200-500ms per candidate batch. Longer prompt → longer TTFT (time to first token): ~100-500ms extra. Total additional latency: 300ms to 1+ seconds on top of baseline LLM latency.

Situations where this matters:

- Voice assistants requiring sub-500ms response time.
- Real-time autocomplete or code completion.
- High-throughput, cost-sensitive pipelines where every millisecond costs money.
- Gaming or interactive simulations requiring low-latency AI responses.

Better alternatives

Fine-tuning for domain knowledge (no retrieval latency at inference). Caching layer for known common queries. Smaller, faster models without retrieval vs. larger models with retrieval.

6.6 The Answer Requires World Knowledge Beyond Any Document

Some questions require synthesizing broad world knowledge, common sense reasoning, or cultural context that is not present in any single document — or any combination of retrievable documents. Retrieval cannot help when the 'knowledge' is implicit and distributed.

Examples:

- 'Why is blue often associated with sadness in Western culture?' — requires cultural anthropology, linguistics, historical context scattered across hundreds of sources, none of which individually answers the question.
- 'What would a medieval peasant find most surprising about modern life?' — requires reasoning across historical knowledge, sociology, and technology that cannot be retrieved.
- Open-ended creative tasks requiring synthesis and imagination.

The retrieval trap

A RAG system might retrieve documents about color psychology, blues music, and the phrase 'feeling blue' — none of which individually answers the question. The retrieval step costs time and money, and the model ends up answering from its own knowledge anyway, potentially distracted by the retrieved fragments.

Better alternative

Direct prompting with chain-of-thought. The model's trained knowledge across the full internet corpus is exactly what is needed here — not a narrow document retrieval.

6.7 Highly Structured Data (Use SQL/APIs Instead)

When your data is perfectly structured — relational databases, spreadsheets, time-series data, transactional records — embedding it into a vector database is the wrong approach. Structured data is best queried with structured query languages.

Why RAG degrades structured data quality

Embedding a CSV row or a SQL record as text destroys the structural information. The embedding for 'revenue: 1245000, quarter: Q3, year: 2024, region: APAC' might be confused with 'revenue: 124500, quarter: Q1, year: 2023, region: EMEA' because they are semantically similar text. SQL queries on the original database are perfectly precise; vector search on embedded rows is approximate and lossy.

Data types that belong in structured systems, not vector databases:

- Financial records (transactions, revenues, balances).
- Inventory and product catalogs with exact attributes.
- Customer databases with demographic and behavioral data.
- Time-series data (metrics, logs, sensor readings).

- Any data where exact filtering, aggregation, or numerical comparison is needed.

Better alternatives

Text-to-SQL (NL2SQL): use an LLM to convert natural language to SQL, execute the query, and return structured results. Direct API integration: give the LLM tools to call your existing APIs and databases. Pandas/code interpreter for analytical tasks over provided data tables.

6.8 The Knowledge Base Changes Faster Than You Can Re-Index

If your content is updated so frequently that the indexed version is perpetually stale relative to the live version, RAG retrieves outdated information. The cost and complexity of continuous re-indexing may exceed the benefit.

Scenarios:

- Wikis or documentation that are edited multiple times per hour.
- Software configuration files that change with every deployment.
- Live chat transcripts or comment threads being read mid-conversation.
- Dynamic pricing or inventory levels changing every few seconds.

The staleness risk

A user asks about a policy that was updated 20 minutes ago. Your RAG system retrieves the old version (indexed 2 hours ago) and confidently presents outdated information. This is worse than having no system at all, because the user trusts the AI-generated answer.

Better alternatives

Direct database or CMS API calls at query time (always fresh). Webhooks + streaming indexing for near-real-time update (mitigates but doesn't eliminate lag). Long context with documents passed directly for small, fast-changing content.

6.9 Single-Turn, Closed-Domain Q&A with a Known Fixed Answer Set

If your application has a finite, small set of known questions with known answers — such as a customer FAQ with 50 questions — you do not need RAG. The overhead of embedding, vector search, and LLM generation is unnecessary when a simple lookup suffices.

Over-engineering risk

Building a RAG pipeline for an FAQ with 30 questions introduces: vector database infrastructure, embedding API costs, retrieval latency, potential retrieval errors, and LLM hallucination risk — all to answer questions whose answers are already precisely known. This is a classic case of the wrong tool for the job.

Better alternatives

Exact-match lookup or fuzzy string matching for small FAQ sets. Traditional keyword search with pre-written answers. Simple intent classification + template response for rule-based Q&A systems.

6.10 Creative Generation Without Factual Grounding

Tasks requiring pure creative output — writing fiction, generating poetry, brainstorming ideas, composing music, inventing product names — have no factual answer to retrieve. Retrieval is irrelevant and the retrieved context may actively constrain creativity.

A retrieved document about 'dragons in fantasy literature' does not help the model write a better dragon story — and may anchor the output to existing tropes rather than enabling novel ideas.

Better alternative

Direct prompting with style guides, examples (few-shot), and creative constraints in the prompt itself. If you want the model to write 'in the style of Hemingway,' provide examples of Hemingway's prose directly in the prompt — not retrieved from a vector database.

6.11 Extreme Cost Sensitivity with Marginal Quality Benefit

RAG has real costs beyond the LLM API: embedding model API calls (for indexing and query-time), vector database hosting, re-ranking API calls, increased prompt length (more tokens = more cost per query), and engineering / operational complexity.

If the accuracy improvement from RAG over direct prompting is small — say, improving answer quality from 85% to 87% — and the cost is 3-5x higher per query, the economics may not justify the complexity.

Cost calculation example

Direct prompting: $0.003 per query (1K tokens input + 200 tokens output at $0.01/K). RAG: embedding call $0.0001 + vector DB $0.001 + 5K token prompt (retrieved context) $0.05 + output $0.002 = ~$0.053 per query. That is 17x more expensive. If accuracy only improves marginally, direct prompting wins on economics.

Better alternative

Start with direct prompting and measure quality. Only add RAG if the quality gap is meaningful and the use case justifies the cost. For internal tools with low query volume, quality may matter more than cost — but for high-volume consumer applications, the economics are critical.

6.12 When Model Fine-Tuning is the Actual Right Answer

RAG is often positioned as an alternative to fine-tuning, but there are specific cases where fine-tuning is genuinely better:

Situation	Why RAG Fails Here	Better Alternative
Highly specific output format required	RAG grounds content but cannot change output style or format as reliably as fine-tuning.	Fine-tune on examples of the desired output format.
Domain-specific reasoning patterns	A medical model needs to think like a clinician, not just know medical facts. RAG gives facts but not reasoning style.	Fine-tune on domain expert reasoning examples.
Extremely low latency required	RAG always adds retrieval latency. Fine-tuned models answer from weights with no retrieval step.	Fine-tune for zero-latency domain knowledge.
Confidential data cannot leave your infrastructure	RAG requires embedding API calls (data leaves to embedding provider). Fine-tuning bakes knowledge into local model weights.	Fine-tune locally on confidential data.
Behavior/persona customization	You want the model to always respond as a specific character or with specific values. RAG cannot change model personality.	Fine-tune on persona-consistent examples.

6.13 Master Decision Table: RAG vs. Alternatives

Scenario	Use RAG?	What to Use Instead
Knowledge in model's training data	**No**	Direct prompting
All docs fit in context window (<128K tokens)	**No**	Long context prompting
Real-time / live data needed	**No**	API calls, tool use
Math / logic / reasoning task	**No**	Chain-of-thought, code interpreter
Latency <500ms, in-model knowledge	**No**	Direct prompting or fine-tuning
Broad world knowledge / common sense	**No**	Direct prompting
Perfectly structured relational data	**No**	Text-to-SQL, direct API
Data changes faster than indexing lag	**No**	Real-time API, streaming indexing
< 50 known Q&A pairs	**No**	Lookup table, intent classification
Pure creative / generative task	**No**	Direct prompting, few-shot examples
Extreme cost sensitivity, marginal quality gain	**No**	Direct prompting + prompt engineering
Output format / behavior customization	**No**	Fine-tuning
Large private knowledge base, semantic queries	**YES**	Dense or Hybrid RAG
Proprietary docs the model wasn't trained on	**YES**	RAG with contextual retrieval
Knowledge updated weekly or slower	**YES**	RAG with periodic re-indexing
Many users, many topics across a doc corpus	**YES**	RAG with hybrid retrieval + re-ranker

Chapter 7: Quick Reference & Glossary

6.1 Retriever Types at a Glance

Retriever	Speed	Accuracy	Best For
Dense (Vector)	⚡⚡⚡	★★★★	Semantic Q&A, general RAG
Sparse (BM25)	⚡⚡⚡⚡⚡	★★★	Exact terms, code, legal
Hybrid (BM25 + Dense)	⚡⚡⚡	★★★★★	Production systems (default)
+ Re-Ranker	⚡⚡	★★★★★	High-stakes precision
Multi-Vector (ColBERT)	⚡⚡	★★★★★	Long docs, fine-grained
Parent Document	⚡⚡⚡	★★★★	Hierarchical docs
Self-Query	⚡⚡⚡	★★★★	Structured metadata
Multi-Query	⚡⚡	★★★★	Ambiguous queries
Graph RAG	⚡⚡	★★★★★	Entity relationships, multi-hop
Time-Weighted	⚡⚡⚡	★★★★	News, recent updates
Long Context	⚡	★★★★★	Small knowledge bases

6.2 Glossary

ANN (Approximate Nearest Neighbor)	Algorithms that find vectors 'approximately' closest to a query vector. Trade tiny accuracy loss for massive speed gains. Used inside all vector databases.
BM25	Best Matching 25. The dominant sparse retrieval algorithm. Scores documents by term frequency (with saturation) and inverse document frequency, normalized for document length.
Bi-Encoder	Neural model that encodes query and document independently into vectors. Enables pre-computation of document vectors. Used for fast first-stage retrieval.
Chunking	Splitting documents into smaller segments before embedding. Chunk size (typically 256-1024 tokens) and overlap are critical tunable parameters.
ColBERT	Multi-vector retrieval model using MaxSim scoring. Each token gets its own embedding; relevance = sum of per-query-token maximum similarities to document tokens.
Context Window	Maximum number of tokens an LLM can process in a single call. Limits how many retrieved chunks can be passed to the model.
Contrastive Learning	Training approach where a model learns to bring similar items closer and push dissimilar items apart in embedding space.

Cross-Encoder	Neural model that processes query and document jointly. More accurate than bi-encoder but cannot pre-compute; used for re-ranking, not first-stage retrieval.
Embedding	Dense vector representation of text. Semantically similar texts have nearby embeddings. Dimensionality is typically 384-3072.
Faithfulness	Evaluation metric: does the generated answer only contain information present in the retrieved context? Low faithfulness = hallucination.
Fine-Tuning	Additional training of a pre-trained model on domain-specific data. Bakes knowledge into weights. Contrast with RAG, which keeps knowledge external.
FLARE	Forward-Looking Active Retrieval. Triggers retrieval mid-generation when the model predicts low-confidence tokens.
GraphRAG	RAG variant using knowledge graphs. Extracts entities/relationships into a graph for multi-hop retrieval beyond what vector search supports.
Hallucination	LLM generating plausible-sounding but factually incorrect information. RAG mitigates by grounding generation in retrieved documents.
HNSW	Hierarchical Navigable Small World. Multi-layer proximity graph algorithm for fast ANN search. Default algorithm in most modern vector databases.
HyDE	Hypothetical Document Embedding. Embeds an LLM-generated hypothetical answer rather than the query itself to improve retrieval.
IDF (Inverse Document Frequency)	Statistical measure of how rare a term is across the corpus. Rarer terms score higher in BM25/TF-IDF.
Knowledge Graph	Structured representation of entities and their relationships as a graph (nodes = entities, edges = relationships). Enables multi-hop reasoning.
Lost in the Middle	LLM tendency to underutilize information positioned in the middle of a long context. Affects RAG systems passing many retrieved chunks.
MaxSim	ColBERT's scoring operator: for each query token, find the max similarity to any document token, then sum across all query tokens.
NDCG	Normalized Discounted Cumulative Gain. Retrieval metric measuring ranking quality with graded relevance judgments.
RAG (Retrieval-Augmented Generation)	AI architecture combining retrieval from an external knowledge base with LLM generation. The retrieved context augments the model's prompt.

Re-Ranker	Second-stage cross-encoder model that re-orders first-stage retrieval results by more accurately scoring each candidate against the query.
ReAct	Reason + Act agent framework. LLM alternates reasoning about what to retrieve and retrieval actions until sufficient information is gathered.
Reciprocal Rank Fusion (RRF)	Score fusion algorithm using only rank positions: score = Σ 1/(k + rank). Parameter-free, robust to different score scales. Used in hybrid retrieval.
Self-Query Retriever	Uses an LLM to decompose natural language queries into a semantic component and structured metadata filters for vector database filtering.
Sentence Transformers	Family of BERT-based embedding models optimized for semantic similarity. Trained with contrastive objectives on sentence pairs.
TF (Term Frequency)	Count of how many times a term appears in a document. One component of BM25 and TF-IDF scoring.
Token	Subword unit used by LLMs. Roughly 3/4 of a word on average. Context windows are measured in tokens.
Top-k	Parameter controlling how many results the retriever returns. Typical values: k=5 for generation, k=20-100 for re-ranking input.
Vector Database	Database optimized for storing and querying high-dimensional embedding vectors. Provides ANN search. Examples: Pinecone, Weaviate, ChromaDB, FAISS, Qdrant.

End of RAG & Retrievers Knowledge Base