

RAG Tutorial

Retrieval-Augmented Generation (RAG) is an architecture that:

1. **Retrieves** relevant documents from a knowledge base.
2. **Generates** an answer using those documents as context.

This is especially useful when the model needs up-to-date or factual information beyond its training data.

RAG operates by fetching relevant documents or data snippets based on a query and then using this retrieved information to generate a coherent and contextually appropriate response.

Retrieval-Augmented Generation (RAG) is a framework for building applications that use an external knowledge base to improve the responses of a language model. It works by first retrieving relevant documents or data from a large corpus and then using that context to generate a response.

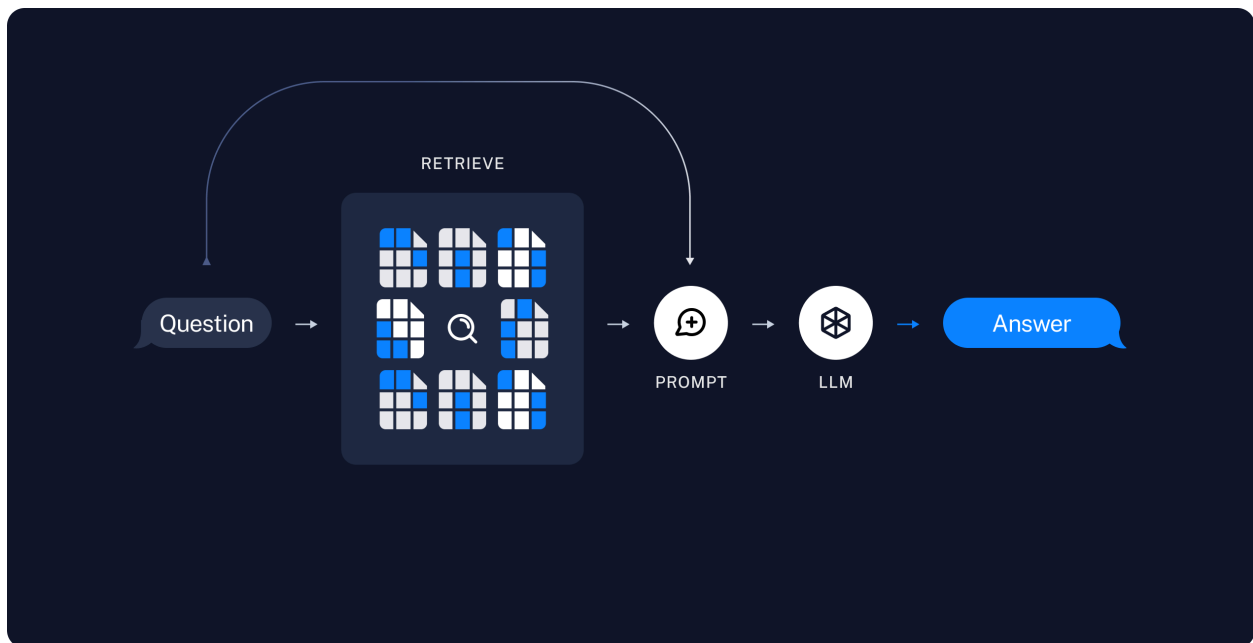
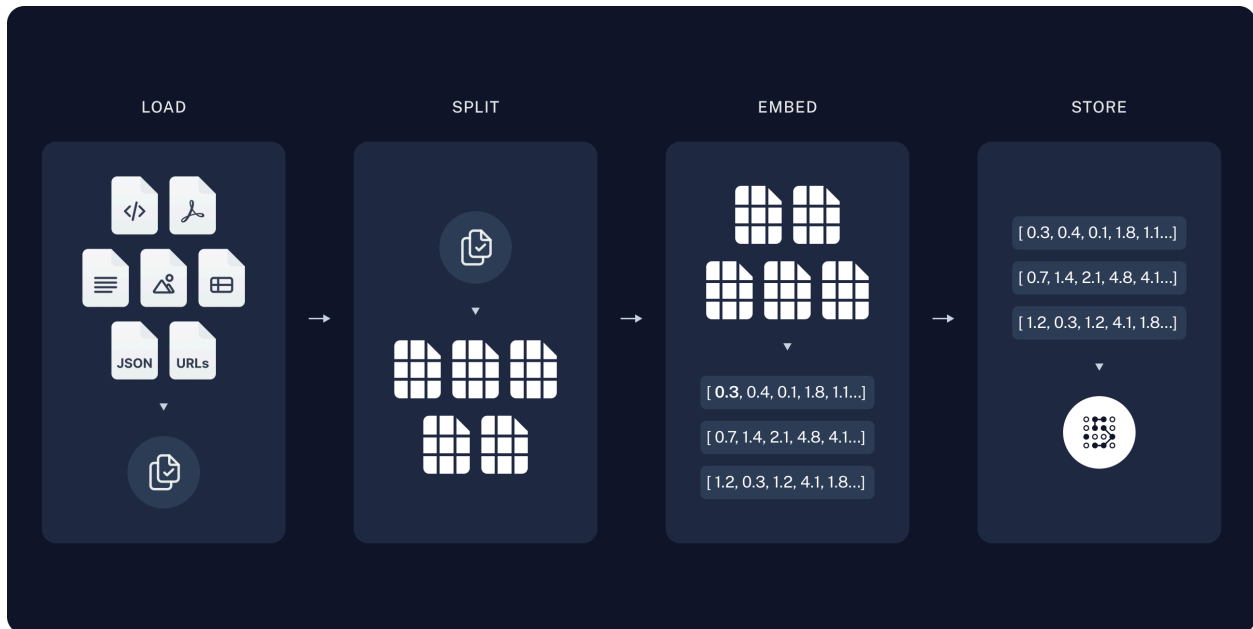
How RAG Works (Simplified)

1. **User Query** → "What is the capital of France?"
2. **Retriever** → Searches a database (e.g., Wikipedia, custom docs) and finds top documents.
3. **Generator** → Uses the query + retrieved docs to generate the answer: "The capital of France is Paris."

Components of a RAG System

1. **Retriever:**
 - Dense Retriever (e.g., FAISS, Elasticsearch, OpenSearch).
 - Converts query and documents into embeddings and finds similar ones.
2. **Generator:**
 - Usually a large language model (LLM) like GPT, BERT2BART, or T5.

- Generates answers using retrieved context.



Indexing

1. **Load:** First we need to load our data. This is done with Document Loaders.
2. **Split:** Text splitters break large **Documents** into smaller chunks. This is useful both for indexing data and passing it into a model, as large chunks are harder

to search over and won't fit in a model's finite context window.

3. **Store:** We need somewhere to store and index our splits, so that they can be searched over later. This is often done using a VectorStore and Embeddings model.

Retrieval and generation

1. **Retrieve:** Given a user input, relevant splits are retrieved from storage using a Retriever.
2. **Generate:** A ChatModel / LLM produces an answer using a prompt that includes both the question with the retrieved data

Key Terms in RAG

1. Retriever

The retriever is the component in RAG that is used to retrieve relevant documents or its chunks from the knowledge base based on the user's query.

Its main purpose is to minimize the work of LLM in order to search all documents or all its chunks by supplying relevant information at inference time.

⚙️ How the Retriever Works — Step-by-Step

Step 1: Query Embedding

The user's question is passed through an **embedding model** to get a high-dimensional **vector representation** (dense vector).

| "What is a quantum computer?" → [0.12, -0.56, 0.88, ...]

Step 2: Document Embeddings

Every document (or chunk) in your knowledge base is **preprocessed** and **embedded** during the indexing phase (offline). These embeddings are stored in a **vector index**.

| Document chunks are like:

- Chunk 1: "Quantum computers use qubits..." → vector A
- Chunk 2: "Classical computers use bits..." → vector B

Step 3: Similarity Search

The query vector is compared with all stored document vectors using a **similarity metric** like:

- **Cosine similarity**
- **Euclidean distance**
- **Dot product**

The system returns the **top-k most similar documents**.

How Similarity Search Works (To retrieve TOP K documents)

Types of Retrievers

1. Dense Retriever

- Uses **neural embeddings** (vector representations).
- Learns **semantic similarity**, not just exact keywords.
- **More accurate** for natural language questions.
- Needs **vector databases**.

 Embedding Models:

- Sentence-BERT (SBERT)
- OpenAI's `text-embedding-3-small` / `3-large`
- HuggingFace models (e.g., `all-MiniLM-L6-v2`)

 Tools:

- **FAISS, Milvus, Weaviate, Pinecone, Chroma**

 Pros:

- Handles synonyms, paraphrases well.

✅ "What is the capital of France?" matches "Paris is the capital city of France."

2. Sparse Retriever

- Based on **exact token match** using term frequency.
- Often uses **BM25** algorithm.

🔧 Tools:

- **Elasticsearch**
- **OpenSearch**
- **Whoosh**

✅ Pros:

- Simple, interpretable.
- Works well when documents and queries share exact words.

❌ Cons:

- Can miss semantically similar content with different words.

❌ "capital of France" might not match "main city of France"

3. Hybrid Retriever

- Combines both dense and sparse retrievers.
- Example: Weighted combination of BM25 score + embedding similarity score.
- Improves both **recall** (find more) and **precision** (find better).

🔧 Libraries like **Haystack** and **LangChain** support this.

2. Generator

The generator is the heart of RAG architecture. Its the component that takes the user query and retrieved documents, and then produces a fluent, coherent, and grounded natural language answer.

Step-by-Step: How the Generator Works in RAG

Input from Retriever

By this point in the pipeline, we have:

1.  **User query:** A natural language question.

| e.g., "What are the benefits of quantum computing?"

2.  **Retrieved documents** (Top-k):

- Chunk A: "Quantum computers use qubits that allow parallel computation..."
- Chunk B: "They can potentially solve problems in seconds that take years for classical computers..."

Step 1: Prompt Construction

Before the generator (the language model) can produce an answer, we need to **construct a prompt** that it can understand.

The prompt includes:

- The **retrieved context** (top-k documents)
- The **original user query**

Step 2: Feed Prompt into the Language Model

- The **Generator is a pre-trained language model** like:
 - **T5 (Text-to-Text Transfer Transformer)**
 - **BART (Bidirectional and Auto-Regressive Transformers)**
 - **GPT (OpenAI's Generative Pre-trained Transformer)**
 - Or any decoder-based transformer model
- The generator receives the full prompt (context + query) and **generates the most likely sequence of tokens** (words or subwords) that form an answer.

Step 3: Token-by-Token Generation

The LLM doesn't generate the answer all at once. It generates it **token-by-token**, predicting what comes next based on:

- The original input (query + context)
- The tokens already generated so far

Step 4: Output Answer

Once generation completes, the model outputs the full answer.

Key Internals of the Generator

1. Decoder-Based Transformer

- GPT, BART, and T5 operate on a **decoder transformer architecture**, which excels at sequence generation.

2. Attention Mechanism

- The model uses **attention** to decide which parts of the context are most important when answering.

3. Language Modeling Objective

- The generator predicts the **next token** based on prior context and the query — trained using **maximum likelihood estimation**.

3. Knowledge Base / Corpus

The

Knowledge Base — sometimes called the **corpus** or **document store** — is the **source of truth** for your RAG system. It's the collection of **text documents** that your retriever will search to find relevant context to feed into the generator.

Types of Knowledge Sources

Source Type	Examples
-------------	----------

Internal	Company manuals, financial reports, legal contracts, support tickets, codebases
External	Wikipedia, news articles, academic papers, forums
Structured	Tables, databases, CSVs
Unstructured	PDFs, Word docs, websites, emails, chats

Step By Step Building of Knowledge Base

◆ Step 1: Document Collection

You start by **collecting raw data** from one or more sources:

- Upload internal documents (PDFs, DOCX, TXT)
- Scrape web pages (HTML)
- Use APIs (e.g., Wikipedia, Arxiv, etc.)
- Connect to databases or cloud storage

Tools used:

- `PyMuPDF`, `pdfminer` – for PDF reading
- `BeautifulSoup`, `Scrapy` – for web scraping
- `LangChain`, `Haystack` – document loaders
- `Google Drive`, `S3`, `MongoDB` – storage integration

◆ Step 2: Preprocessing the Documents

Before indexing, documents need **cleaning and structuring**. This step ensures uniform, high-quality data.

Tasks:

- Remove headers/footers, boilerplate text
- Normalize whitespace, punctuation
- Fix encoding issues
- Strip HTML tags if needed

- Convert to plain text

◆ Step 3: Chunking (Text Splitting)

Language models have a **token limit**, so entire documents can't be processed at once. Thus, we **split large texts into smaller, overlapping chunks** (a.k.a. "sliding windows").

🔨 Techniques:

- **Fixed-size chunks:** e.g., 500 tokens with 50-token overlap
- **Semantic splitting:** by paragraph, heading, sentence
- **Recursive Character Splitter:** tries sentences → paragraphs → tokens

🔄 Overlap helps preserve context across chunk boundaries.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50
)
chunks = splitter.split_text(document_text)
```

◆ Step 4: Embedding Each Chunk

Each text chunk is passed through an **embedding model** to convert it into a **vector (dense representation)**.

🔍 Popular embedding models:

- `text-embedding-3-small` / `3-large` (OpenAI)
- `all-MiniLM-L6-v2` (HuggingFace)
- `bge-base-en` , `Instructor` , `E5`

Each chunk becomes:

```
{
  "text": "Quantum computers use qubits...",
  "embedding": [0.34, -0.51, 0.72, ...],
  "metadata": {
    "source": "page_2.pdf",
    "section": "Quantum Introduction"
  }
}
```

◆ Step 5: Storing in a Vector Database

All chunk embeddings are stored in a **vector database** that enables **fast similarity search**.

Popular vector databases:

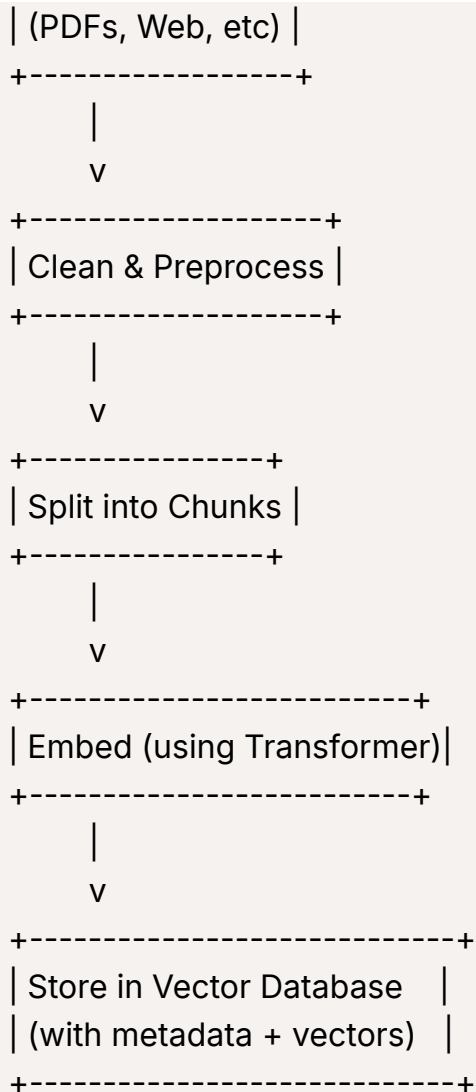
Name	Description
FAISS	Fast, local, open-source
Pinecone	Managed, scalable, cloud-native
Weaviate	Semantic + hybrid search
Milvus	High-performance, enterprise scale
Chroma	Open-source, local vector DB

◆ Step 6: Metadata Indexing (Optional but Powerful)

You can index metadata along with embeddings:

- `document_id`
- `page_number`
- `source_url`
- `tags` , `categories`

```
+-----+
| Raw Documents |
```



4. Embedding

What Is an Embedding?

An **embedding** is a **dense numerical representation** of a piece of text (word, sentence, paragraph, or document) in a **high-dimensional vector space**. The goal is to **capture the semantic meaning** of the text so that **similar meanings** are **close together** in this space.

Why Embeddings Matter in RAG

In a RAG pipeline:

- You **convert your documents** (during pre-processing) and **user queries** (at runtime) into **vectors**.
- These vectors are **compared using similarity measures** (like cosine similarity) to find the most relevant content.

Without good embeddings, retrieval will fail — even if your generator is powerful.

Step By Step Process of How Embeddings Works

1. Choose an EmbeddingModel to turn a sentence into meaningful vectors
2. Input a text chunk
3. Convert a vector embedding
4. Semantic Encoding means it encodes Keywords, Context, Meaning, Syntax and semantics combined
5. Store embeddings in a Vector store

5. Index

An **index** in RAG is a **specialized data structure** that allows fast retrieval of **relevant documents** (or chunks) from the knowledge base. It does this by storing and organizing **embeddings** (i.e., vectors) and/or **tokens** of documents in a way that supports **efficient search**.

Why Is Indexing Needed?

Without indexing, you'd have to **brute force** through every document vector for every user query. That's **slow** and **unscalable**, especially if you have:

- 100K+ chunks of data
- Multiple users querying at once
- Real-time response requirements

Indexing makes search fast, scalable, and accurate.

Types of Indexes in RAG

RAG systems usually use two main types of indexes depending on the retriever:

Index Type	Retrieval Method	Used In
Vector Index	Embedding similarity (e.g., cosine)	Neural retrievers
Inverted Index	Keyword overlap (e.g., TF-IDF, BM25)	Traditional IR (text search)

1. Vector Index (Neural Retrieval)




This is the **standard** in most modern RAG systems.

+ What It Stores

- Embedding vectors of each document chunk (e.g., 768 or 1536 dimensions)
- Optional: metadata (title, source, tags)

How It Works

When a user submits a query:

1. Query is embedded into a vector 
2. Vector  is **compared** with all stored vectors 
3. Similarity is calculated using a metric (e.g., cosine similarity)
4. Top-k most similar vectors are returned

Similarity Metrics

- **Cosine Similarity** (most common)
- **Euclidean Distance**
- **Dot Product**

Common Vector Index Structures

1. Flat (Brute Force)

- All vectors compared to query vector
- Very accurate but slow for large data

✅ Simple, ❌ Not scalable

2. IVF (Inverted File Index)

- Vectors grouped into clusters (like k-means)
- Search limited to top-n clusters

✅ Fast, ❌ May miss some results

3. HNSW (Hierarchical Navigable Small World)

- Graph-based nearest neighbor search
- Fast and accurate for millions of vectors

✅ Very fast + high recall

4. Product Quantization (PQ)

- Compress vectors to save space
- Used in large-scale FAISS

✅ Memory-efficient, ❌ Lower precision

2. Inverted Index (Lexical Retrieval)

This is the **classic** search engine approach (used in Elasticsearch, Whoosh, BM25, Lucene).

+ What It Stores

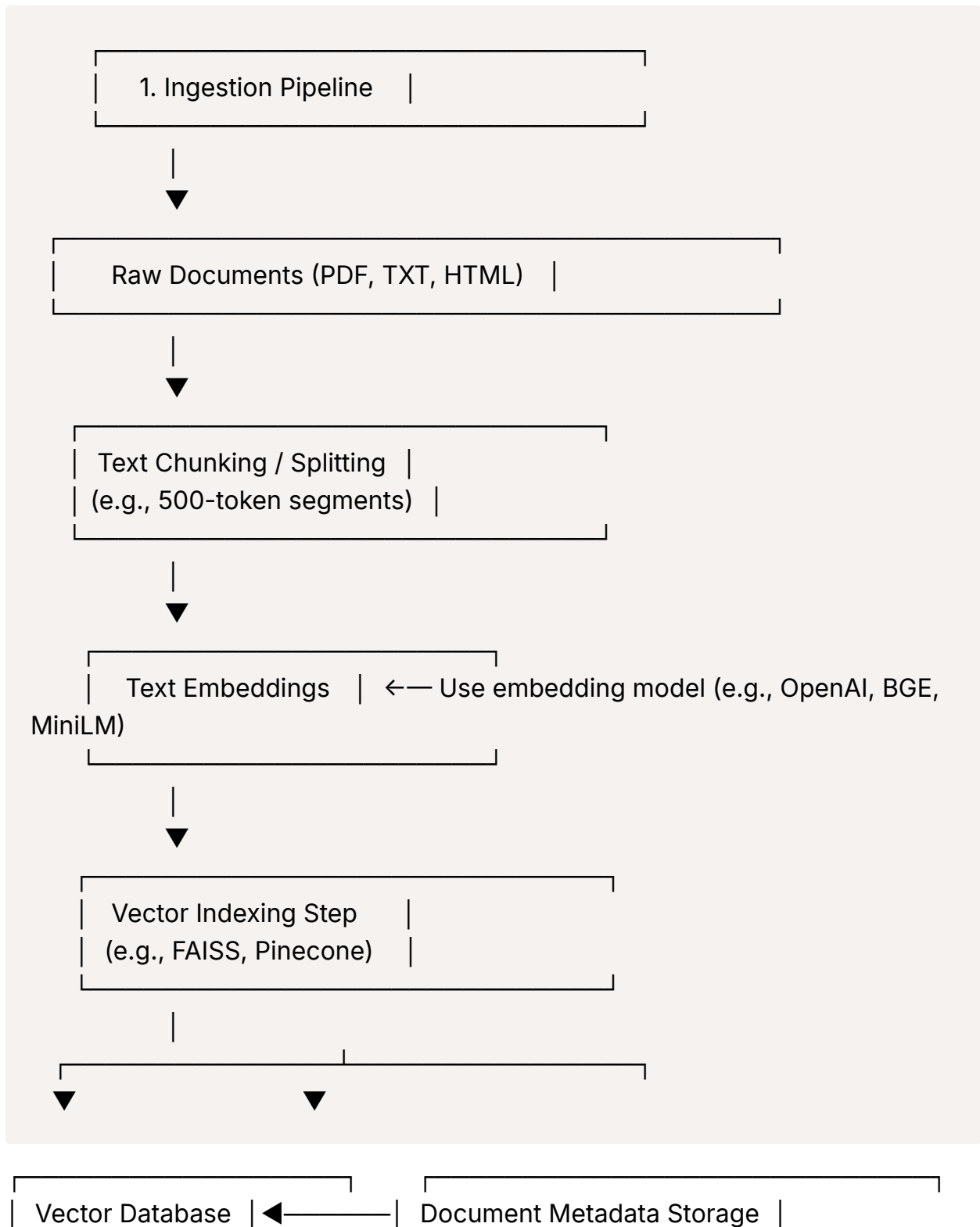
- A mapping of **terms** → **document IDs**
- Along with term frequency (TF), document frequency (DF)

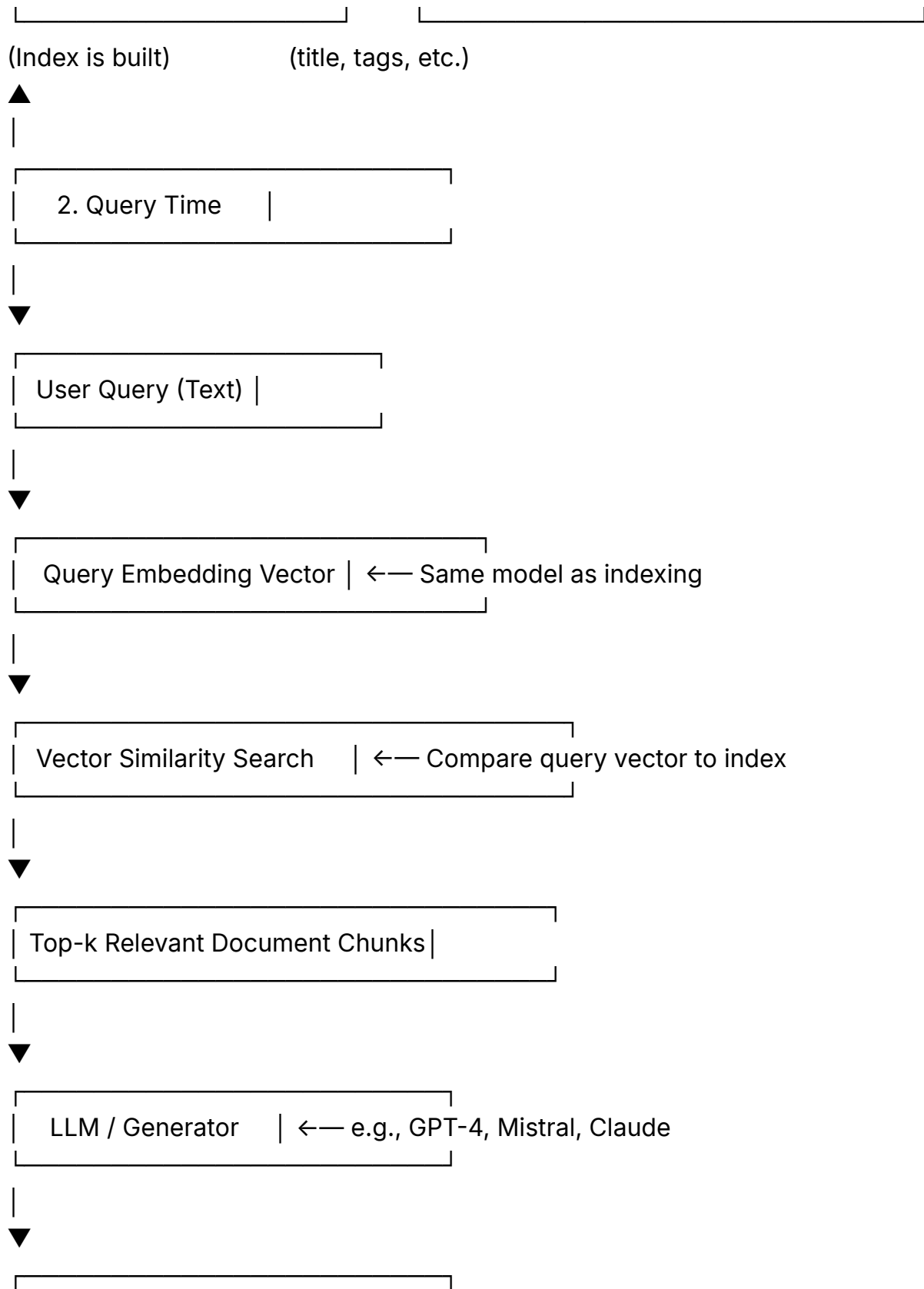
🔍 How It Works

When a user queries:

1. Search engine finds all documents with overlapping terms
2. Scores them using BM25 or TF-IDF
3. Returns ranked list of documents

$$\text{score}(d, q) = \sum [\text{IDF}(q_i) * (\text{tf} * (k + 1)) / (\text{tf} + k * (1 - b + b * \text{dl} / \text{avgdl}))]$$





| Final Answer to User |

6. Context

In RAG, **context** refers to the **retrieved information (text chunks)** that are passed along with the **user's query** to the **language model (generator)**. This information becomes the **"knowledge grounding"** for the model to generate a response.

Why Context Matters

LLMs like GPT-4 don't have persistent memory of your data unless you **give them context** at inference time. That's what the retrieved chunks provide.

Without relevant context:

- The model might **hallucinate**
- Answers may be **generic or incorrect**
- You **lose traceability** to a data source

With good context:

- Answers are grounded in real data
- You get **explainable, source-backed** output

7. Top-k Retrieval

It's the process of retrieving the **k most relevant chunks** from your indexed knowledge base based on their **similarity to the query**.

- **k** is a configurable parameter: common values are 3, 5, 10.
- The retrieved chunks will then be used as **context** (see above).

How It Works (Step-by-Step):

1. User inputs a query

→ e.g., "How does quantum entanglement work?"

2. Query is embedded into a vector

- using same embedding model as the corpus
- 3. **Compare query vector** to stored document vectors
 - using cosine similarity or other metric
- 4. **Sort by similarity score**
- 5. **Return top k documents**

8. Chunking

Chunking is the process of **breaking long documents into smaller pieces**, or "chunks", before embedding and indexing.

You can't embed or retrieve entire books — so we chunk them.

Why Is Chunking Important?

Reason	Explanation
✓ Better retrieval	More granular units help match relevant context
✓ Fits token limits	LLMs like GPT-4 have max token limits (e.g., 8k, 32k)
✓ Embedding accuracy	Large blocks confuse embedding models
✓ Scalable search	Finer indexing = faster, more precise retrieval

Types of Chunking

Method	Description	Example
Fixed-size	Cut every 500 tokens	Chunks: [1–500], [501–1000]...
Sentence-based	Break by sentence boundaries	NLP + regex
Sliding window	Overlap between chunks	Chunk1: 0–500, Chunk2: 400–900
Metadata-aware	Use sections, headers	Chunk by title, subheading

9. Query Expansion / Reformulation

Query Expansion or Reformulation is the process of **rewriting, rewording**, or **augmenting** a user's original question to make it **more suitable for retrieval**. The

goal is to **increase recall** — i.e., to find **more or better-matching documents** from the index.

Why Do We Need It?

Embeddings or keyword search engines may **fail** to match documents that:

- Use **different terms** or **synonyms** (e.g., "flu" vs "influenza")
- Use **abbreviations** or **spelling variants**
- Have **missing keywords** from the query
- Answer the question **indirectly**

10. Reranker

A **reranker** is a **second-pass model** that takes the **top-k retrieved documents** (from the initial vector search or keyword search), and **reorders** them based on **semantic relevance** to the query.

The initial retriever is fast but shallow. The reranker is slower but more accurate.



Why Do We Need It?

Embedding-based retrieval (e.g., cosine similarity in FAISS) is **efficient but approximate**. It:

- Often retrieves documents that **look close numerically** but may not be **truly relevant**
- Doesn't use the **full query-document pair** in scoring — it just compares vectors

Rerankers **read the actual text of the query and the documents** and produce a **more accurate score**.



How It Works (in Steps)

1. Retrieve Top-k Documents

→ Via vector similarity

2. Send Each Pair to Reranker Model

→ Each `(query, doc_i)` pair gets a **relevance score**

3. Reorder Documents by Score

4. Pass Reranked Top-k to LLM for Generation

11. Prompt (in RAG)

- The full input to the generator, which includes:
 - User query
 - Retrieved context
 - Possibly system instructions or formatting

12. Hallucination (in RAG)

A **hallucination** occurs when a language model generates **text that sounds plausible but is factually incorrect**, unsupported by the data, or even completely made up.

These errors can be:

- **Factual errors** (e.g., stating the wrong date or formula)
- **Invented citations** (e.g., fake references)
- **Inferred logic errors** (e.g., drawing a wrong conclusion)
- **Misattributions** (e.g., saying a quote is from Einstein when it's not)

Why Do LLMs Hallucinate?

Root causes include:

Cause	Description
Lack of source grounding	LLMs are trained on vast, general data but don't "know" if something is true in a specific context

Probability-based generation	LLMs predict the next token using probabilities — they may “guess”
Prompt ambiguity	Vague or missing context leads to confident, but incorrect answers
Training data gaps	If a topic wasn’t covered well during training, the LLM may “improvise”
Lack of memory or state	LLMs don’t retain persistent memory unless explicitly provided

The Pipeline for Anti-Hallucination

1. **User Asks a Question**
2. **Retriever finds relevant documents**
3. **LLM uses only those documents to generate an answer**
4. **Optionally: filter answer using factuality checks or LLM criticism**

Techniques RAG Uses to Minimize Hallucination

Method	How It Helps
Context Injection	Limits the LLM to respond only using retrieved facts
Top-k Retrieval	Ensures only the most relevant chunks are included
Chunk Quality	Chunking ensures small, accurate blocks are used
Prompt Constraints	Use instructions like: “Only answer based on the context below”
Reranking	Prioritizes high-relevance and high-trust chunks
Citations & Source Display	Adds transparency and allows verification

Limitations — RAG Can Reduce, But Not Eliminate

Even with RAG, hallucination **can still occur** if:

- Retrieval fails (e.g., wrong or no chunks found)
- Retrieved data is outdated or ambiguous

- The model **ignores the prompt constraints**
- The **token budget** cuts off important context