

Retrieval Augmented Generation (RAG)

RAG

- Large Language Models (LLMs) are highly capable but encounter several issues like
 - creating inaccurate or irrelevant content (hallucinations)
 - using outdated information
 - operating in ways that are not transparent (blackbox reasoning).
- Retrieval-Augmented Generation (RAG) is a technique to solve these problems by augmenting LLM knowledge with additional domain specific data.

- A **hallucination** is when an LLM generates text that sounds correct but is **factually wrong or unsupported** by the prompt or input.

Why Do Hallucinations Happen?

- LLMs are **predictive models**, not fact checkers.
- They generate the **most likely next word**, based on training data — not always the *true* word.
- If training data has **gaps**, or the prompt is **ambiguous**, it guesses.

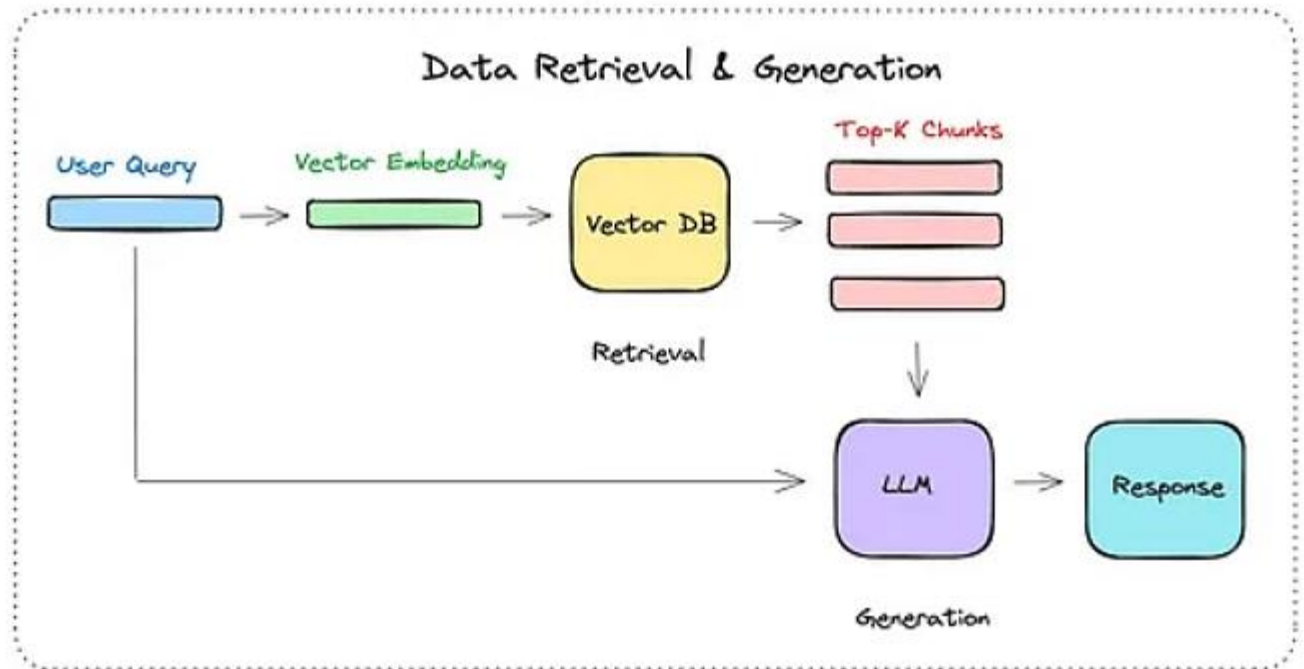
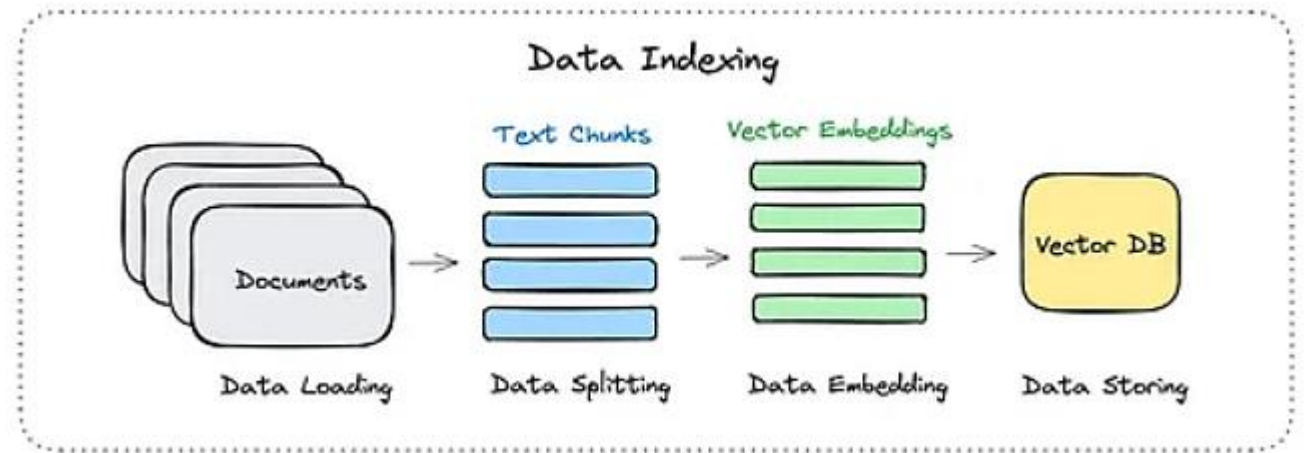
Why RAG?

- Combines **retrieval** and **generation** to answer queries.
- Useful when the base LLM does not have all the required knowledge.
- Example: Instead of asking GPT to summarize a niche article, RAG retrieves the article, then uses GPT to summarize it.
- Base models are limited to static training data.
- RAG introduces **dynamic, up-to-date, domain-specific knowledge** via retrieval.
- Great for tasks like: document Q&A, chat with PDFs, legal/medical assistants, customer support, etc.

Basic RAG Pipeline

Two phases:

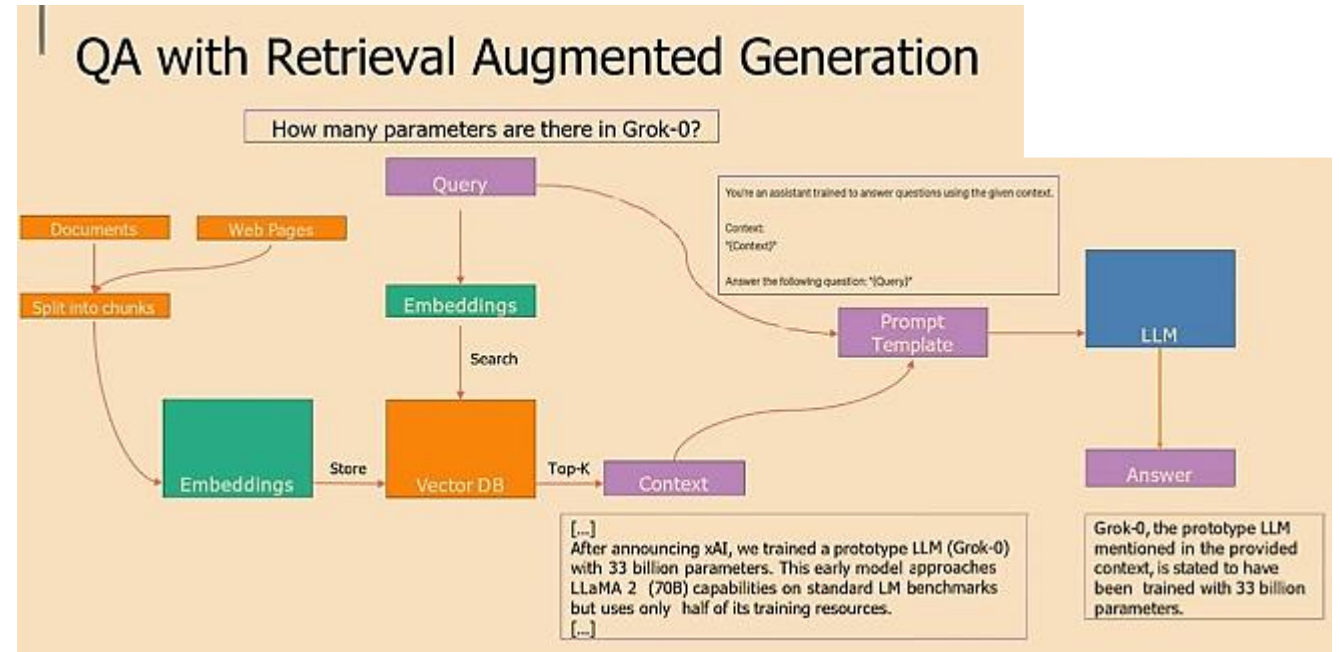
- Data Indexing
- Retrieval & Generation



Basic RAG Pipeline

Two phases:

- Data Indexing
- Retrieval & Generation



Data Indexing Process

- 1.Data Loading:** This involves importing all the documents or information to be utilized.
- 2.Data Splitting:** Large documents are divided into smaller pieces, for instance, sections of no more than 500 characters each.
- 3.Data Embedding:** The data is converted into vector form using an embedding model, making it understandable for computers.
- 4.Data Storing:** These vector embeddings are saved in a vector database, allowing them to be easily searched.

Retrieval and Generation Process

1.Retrieval: When a user asks a question:

- The user's input is first transformed into a vector (query vector) using the same embedding model from the Data Indexing phase.
- This query vector is then matched against all vectors in the vector database to find the most similar ones (e.g., using the Euclidean distance metric) that might contain the answer to the user's question. This step is about identifying relevant knowledge chunks.
- **2. Generation:** The LLM model takes the user's question and the relevant information retrieved from the vector database to create a response. This process combines the question with the identified data to generate an answer.

Key Concept: Embedding ≠ Vectorstore

Embeddings = numerical meaning

Vectorstore = retrieval engine that stores and searches them

The real power comes from **connecting them to a language model prompt**.

- **LangChain / LlamaIndex** for RAG orchestration.
- **FAISS / Chroma / Weaviate** for vectorDB/ vectorstore.
- **Sentence Transformers / OpenAI Embeddings** for embedding.
- **OpenAI / Mistral / LLaMA / Mixtral** for LLM.

RAG copies answers from vectorstore	✗ No, the LLM generates responses from context
LLM ignores vectorstore and just answers	✗ RAG injects relevant info to guide LLM responses
LLM replaces search engine	✓ RAG combines search (retrieval) + reasoning (generation)

Vector DBs

A **vector database (Vector DB or vectorstore)** is a special type of database that stores **vector representations (embeddings)** of data and allows for **similarity search**.

- You take your **text chunks**, convert them to **vectors (embeddings)** using a model such as sentence-transformers.
- These vectors go into the vector DB.
- Later, a **query** is also turned into a vector.
- The DB finds the most **similar stored vectors** to that query using distance metrics (cosine similarity, L2, etc.).

Vector DB	Storage	Speed	Scalability	Metadata Support	Type	Notes
FAISS	In-memory	Very fast	Single node	Limited	Library	Great for prototyping, not distributed
Chroma	Lightweight local	Fast	Limited	Good	Local DB	Easy to use with LangChain
Pinecone	Cloud-hosted	Fast	Very scalable	Yes	Cloud SaaS	Paid, ideal for production
Weaviate	Cloud or local	Fast	Scalable	Strong support	Semantic DB	Schema-based, has hybrid search
Qdrant	Cloud or local	Fast	Scalable	Strong	Vector DB	Good open-source option
Milvus	Cloud or on-prem	Fast	Scalable	Yes	Industrial	More complex setup

Difference bw retriever and vector db?







The **retriever** is the interface or wrapper that tells the DB *how/when* to do it, actual similarity is identified by the vector db.

What is indexing in vector DBs?

Indexing is the process of organizing vector embeddings in a way that enables **fast and efficient similarity search**. Since embeddings are high-dimensional, we can't just brute-force compare everything — it's slow for large datasets.

So, vector DBs use **different indexing algorithms** to speed up this search while balancing:

- **Search speed**
- **Memory/storage usage**
- **Accuracy (recall)**

Index Type	Description	Speed	Accuracy	Use Case
Flat (Brute Force)	Compares the query vector to all stored vectors (e.g., <code>IndexFlatL2</code> in FAISS)	 Slow	 100% accurate	Small datasets
IVF (Inverted File Index)	Groups vectors into clusters. Compares only inside nearest clusters.	 Fast	 High, not 100%	Medium/large datasets
HNSW (Hierarchical Navigable Small World)	Builds a graph structure for efficient navigation	 Very fast	 High	Pinecone, Milvus, and some FAISS

If u use

```
index = faiss.IndexFlatL2(dimension)
```

You're using brute-force Euclidean distance. But if you use:

```
python

index = faiss.index_factory(d, "IVF100,Flat")
```

You're using **Inverted File Index with 100 clusters**, which will be faster on large datasets.

Inverted File Index Process:

Step 1: Given a query, you compute its vector representation (embedding).

Step 2: The query vector is compared to the cluster centroids (the center of each cluster, typically the **mean vector** of the embeddings in that cluster) to determine which cluster(s) are closest to the query vector.

Step 3: Once the closest cluster is identified, you then perform a **local search** within that cluster (using a fast search technique like **brute force** or **approximate nearest neighbors** within the cluster).

Step 4: The most relevant vectors (or chunks/documents) from that cluster are returned.

- Retrieval-Augmented Generation = **Retrieval** (via similarity search in vector DB) + **Generation** (via LLM)

The retriever is passive until the **user query arrives**. Once it does, it searches for the most relevant information **based on vector similarity** between:

- the prompt's vector, and
- the document chunks' vectors.

- User Prompt
- Embedding the Prompt & Retrieval
- Context + Prompt → Augmented Prompt

You combine:

- The original prompt
 - The retrieved chunks (text, not vectors)
 - Into a **final prompt** that is sent to the LLM.
- LLM Answer Generation
 - Uses both the prompt and context to generate a **structured, grounded, and relevant response**.
 - The base model **does not rely on parametric memory** alone — it gets help from the **retrieved documents**.

RAG vs Finetuning

Use RAG when:

- Your model needs access to **frequently changing** or **large corpora**.
- You want a **plug-and-play** knowledge injection without training.
- You're building **QA systems**, assistants, agents that depend on long external context.

Use Fine-tuning when:

- You have **domain-specific tasks** like classification, translation, summarization with style.
- You want the model to **learn latent patterns** or **linguistic quirks** of a domain.
- Your tasks aren't just about knowledge access — they're about **behavior** and **task-specific adaptation**.

Aspect	RAG (Retrieval-Augmented Generation)	Fine-Tuning
Speed to Deploy	✅ Fast (no model weights modified)	❌ Slower (needs training time)
Cost	✅ Cheaper (no training compute)	❌ Costly (GPU hours, storage)
Flexibility	✅ High (just change documents)	❌ Low (new task = new fine-tune)
Knowledge Updating	✅ Easy (just update vector DB)	❌ Hard (must re-train with new info)
Performance (on very task-specific domains)	🟡 Medium (depends on retriever + context length)	✅ High (can deeply learn the domain)
Hallucination Risk	✅ Lower (has factual grounding from retrieved context)	❌ Higher (only relies on internal weights)
Use Case Fit	✅ Great for general knowledge QA, chatbots, RAG agents	✅ Best for narrow domain tasks (e.g., legal doc classification, biomedical NER)

- Can RAG be used for classification or translation tasks?

Classification with RAG (Doc-Based or Few-shot Style)

RAG can help when:

- You're classifying long documents based on external *domain-specific* rules or definitions.
- You have **label definitions** or **supporting guidelines** stored in your vector DB.

Example:

Task: Classify legal cases into categories (e.g., Civil, Criminal, Corporate)

How RAG helps:

- You retrieve similar precedent cases or law snippets via FAISS/Chroma.
- Pass those chunks + the input case into the LLM.
- Ask: *“Based on the retrieved precedents, classify this case.”*

So here, **retrieval gives the LLM the label reasoning**, making classification better, especially in **low-resource** or **ambiguous** settings.

- <https://medium.com/@drjulija/what-is-retrieval-augmented-generation-rag-938e4f6e03d1>
- https://youtu.be/iN0JDhCgf_Y?si=Wgq32WJAO_E-Yuci
- <https://humanloop.com/blog/rag-explained>
- <https://medium.com/@hassan.tbt1989/build-a-rag-powered-llm-service-with-ollama-open-webui-a-step-by-step-guide-a688ec58ac97>