**RAG**

**Retrieval Augmented Generation**

Retrieval-Augmented Generation (RAG) is a powerful technique that enhances the capabilities of large language models (LLMs) by combining external information retrieval with text generation. Instead of relying solely on a pre-trained model's knowledge, RAG leverages a retrieval component to fetch relevant data from external sources, making the AI's responses more accurate, dynamic, and informed.

**Components of RAG**

**Retriever**: Identifies and retrieves the relevant document

**Generator:** Take retrieved documents and the input query to generate coherent and contextually relevant response

**RAG:** A framework that combines the strengths of retrieval based systems and generation based models to produce more accurate and contextual relevant response

i.e an efficient way to customize an LLM model with our data

**User Query** → "Tell me about quantum computing."  
**Retriever** → Searches for relevant documents from a vector database.  
**Retrieved Documents** → Returns 2-3 top matching documents.  
**LLM Augmentation** → The model reads and integrates the retrieved knowledge.  
**Final Response** → The LLM generates an accurate, well-informed answer.

***Doubt: Is the answer is from external resources and llm pre trained data or only from external resources***

*the answer is typically a combination of both*

*If the retrieved documents provide a clear answer, the LLM mainly uses external data.*

*If the retrieved documents are not fully relevant, the LLM may rely partially on its pre-trained knowledge to fill gaps.*

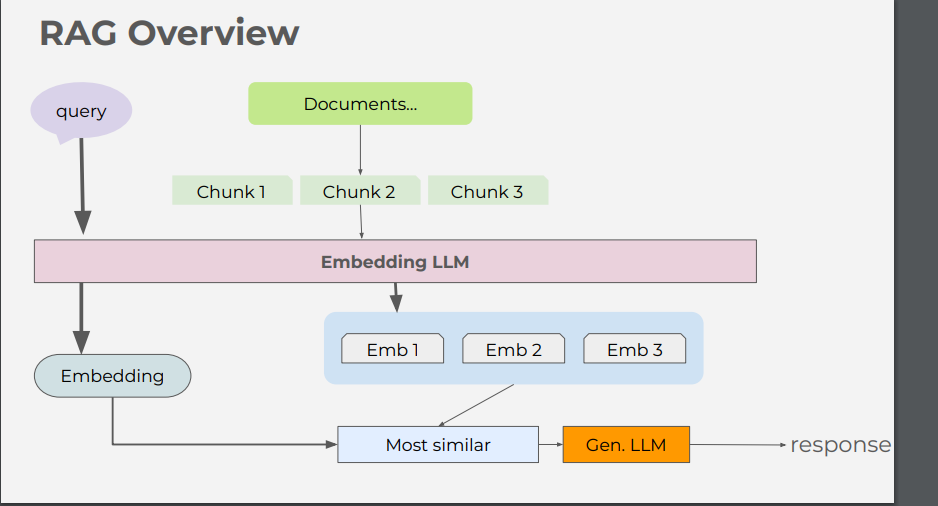
*If no useful documents are retrieved, the LLM falls back on its internal knowledge (which might be outdated or incomplete).*

***Can RAG Be Configured to Use Only External Data?***

*✅* ***Yes!*** *You can design RAG systems where the LLM* ***only uses retrieved documents*** *and ignores its pre-trained knowledge.*

* *This is done by* ***formatting prompts*** *to strictly rely on retrieved documents.*
* *Example: “Answer only based on the provided documents. If the answer is not found, say ‘I don’t know.’”*

**RAG Overview:**

****

**Step1: User query(input)**

User query is converted into an embedding(vector representation) using an embedding model. Since computers process only numerical data, transforming text into vectors allows for efficient similarity matching

**Step2:** **Chunking and embedding**

Documents are divided into chunks

Each chunk is converted into embeddings using the same embedding model that processed the query. These embeddings are stored in a vector database

**Step3: Retrieval**

The query embedding is compared with stored document embeddings

The system searches for most similar embeddings using the vector similarity search. The most relevant chunk is retrieved

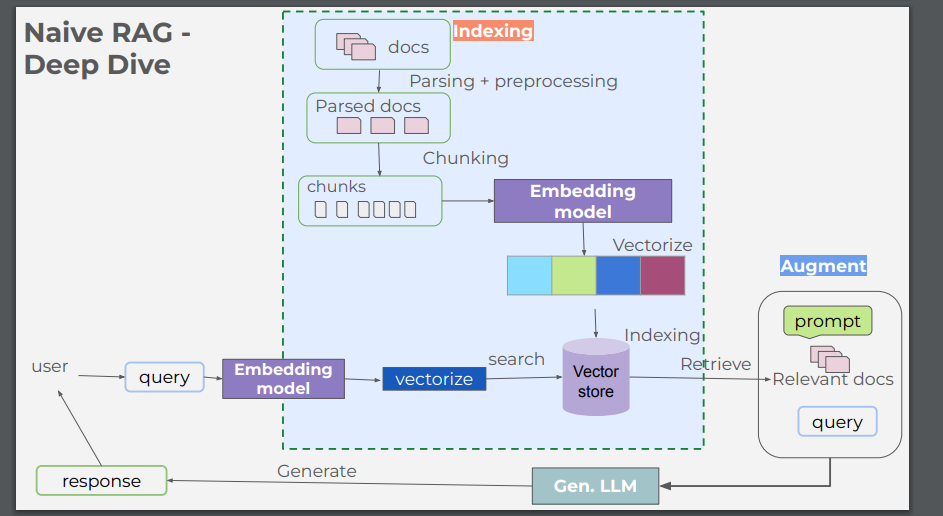
**Step 4:**

This retrieved document chunks are passed as additional context to generative LLMs . Now the LLM generates response based on its pretrained knowledge and retrieved chunk.

The final response is returned to the user

This approach reduces hallucinations because the LLM relies on real data.

Unlike traditional LLMs that rely solely on pre-training, RAG ensures that answers are updated dynamically.



**Advantages of ollama:**

1. Privacy concerns  
   → Running LLMs locally ensures that your sensitive data never leaves your machine—no risk of data leaks to external servers.
2. Ease of use  
   → Normally, setting up large language models requires technical knowledge (environments, dependencies). Ollama simplifies the setup with pre-packaged models and easy commands.
3. Cost efficiency  
   → Cloud-based LLM services (like OpenAI API) can become expensive. Ollama eliminates those costs by running models on your own hardware.
4. Latency reduction  
   → Local execution means no round trips to the cloud, resulting in much faster response times.
5. Customization  
   → You can fine-tune or modify models more freely, tailoring them to your specific needs without vendor limitations.

ollama run llama3.3 is to pull and run

/help gives all commands to operate

Ollama list-> shows all the models which I installed till now

When we have large number of parameters, the accuracy increases but also it requires more number of computational resources

Embedding length: the size of the vector representation for each token in input text

Higher the dimensional embeddings can capture the more nuanced meanings and relationships between the words

quantization: Q4\_K\_M: Quantization is a **compression technique** used to **reduce the size of neural network models** by lowering the precision of their weights.

Instead of storing weights as:

* **32-bit floats (high precision, high memory)**

It stores them as:

* **4-bit integers (low precision, low memory)**

**Meaning of Q4\_K\_M:**

* **Q4** → Quantized to **4 bits per weight**.
* **K\_M** → Specific quantization algorithm (variations based on how data is compressed & decompressed efficiently).

ollama create model-name -f ./Modelfile to create our own replication of particular model

A diagram of a software application

AI-generated content may be incorrect.

U ibased- msty app

**Using rest API**

import requests

import json

url="http://localhost:11434/api/generate"   #ollama rest api, even without this we can do with help of ollama python library

data={

    "model":"llama2:7b",

    "prompt":"tell m,e story for children",

    "max\_length":100,

}

response=requests.post(url,json=data, stream=True)

# check the response status

if response.status\_code == 200:

    print("Generated Text:", end=" ", flush=True)

    # Iterate over the streaming response

    for line in response.iter\_lines():

        if line:

            # Decode the line and parse the JSON

            decoded\_line = line.decode("utf-8")

            result = json.loads(decoded\_line)

            # Get the text from the response

            generated\_text = result.get("response", "")

            print(generated\_text, end="", flush=True)

else:

    print("Error:", response.status\_code, response.text)

**Step-by-step Explanation:**

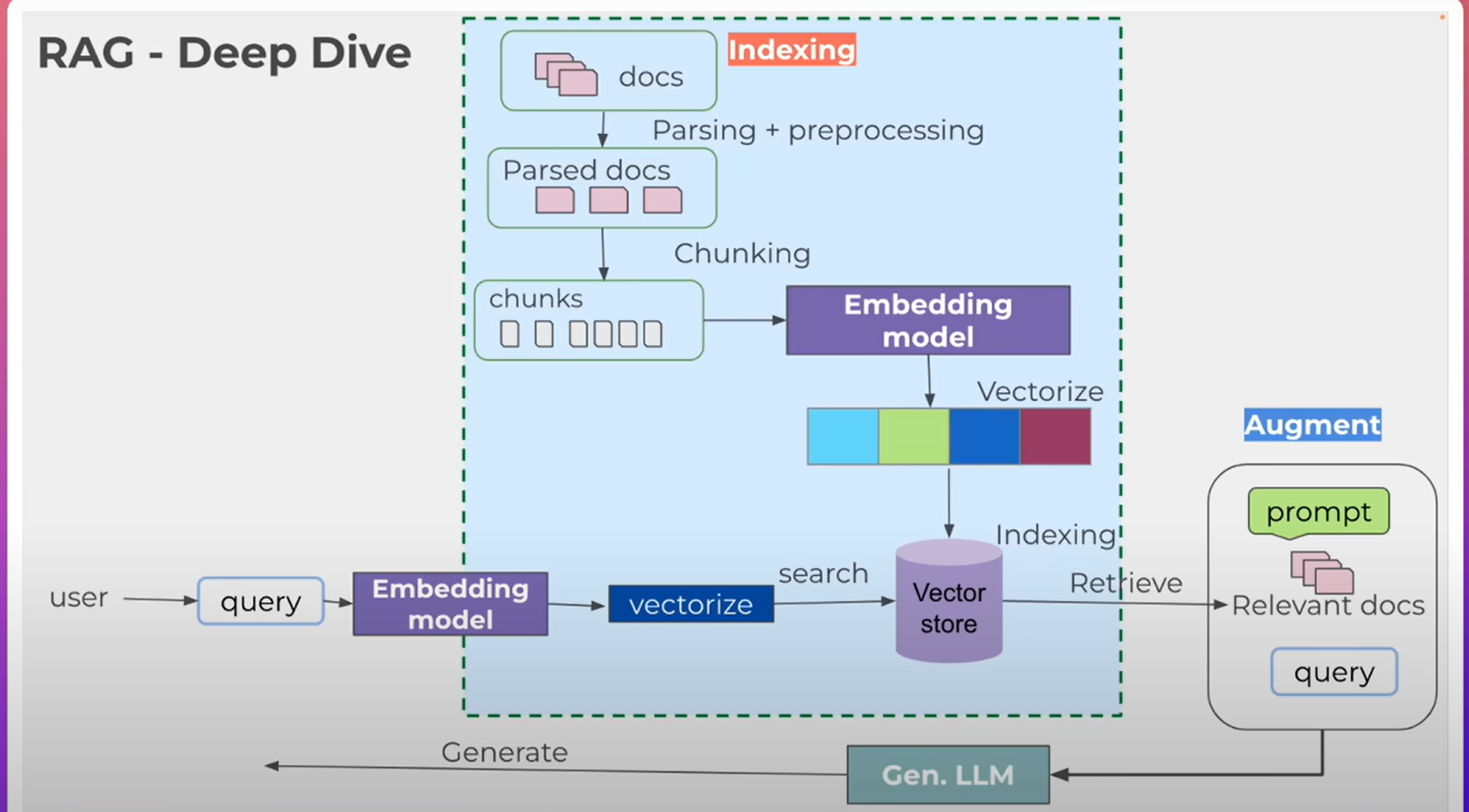
1. **Import libraries:**
   * requests → For making HTTP requests.
   * json → For handling JSON data.
2. **Define API URL and request data:**
   * url="http://localhost:11434/api/generate" → API endpoint for text generation.
   * data={"model":"llama2:7b", "prompt":"tell me a funny story"} → Specifies the model (llama2:7b) and prompt.
3. **Send a POST request (streaming response):**
   * response = requests.post(url, json=data, stream=True) → Sends request to the model and enables streaming.
4. **Check response status:**
   * if response.status\_code == 200: → Ensures the request was successful.
   * If not, prints the error.
5. **Process the streamed response:**
   * for line in response.iter\_lines(): → Reads the response line by line.
   * decoded\_line = line.decode("utf-8") → Converts bytes to a string.
   * result = json.loads(decoded\_line) → Parses JSON data.
   * generated\_text = result.get("response", "") → Extracts generated text.
   * print(generated\_text, end="", flush=True) → Prints text without extra new lines.

**Key Points (Short Summary)**

* **Imports necessary modules** → requests, json.
* **Defines API URL and request payload** → Specifies model and prompt.
* **Sends POST request with streaming enabled** → Efficient for long responses.
* **Checks response status** → Ensures successful request.
* **Processes streamed response** → Reads and prints generated text in real time.

🔹 **Why streaming (stream=True) is important?**

* Allows real-time text generation, instead of waiting for the full response.



**1. INDEXING (Data Preparation & Storage)**

This is a **preprocessing stage** that happens **before** users ask any questions.

**🗂️ a. Raw Documents (docs)**

* These are the original unstructured documents you want the system to learn from — PDFs, web pages, reports, etc.

**🛠️ b. Parsing + Preprocessing**

* Converts documents into machine-readable format.
* Tasks: remove noise, normalize text, clean up formatting.

**✂️ c. Chunking**

* Documents are broken into **smaller chunks** (e.g., paragraphs, sentences).
* Why? Most LLMs (like GPT) have token limits — they can’t process full documents at once.
* Chunks maintain semantic meaning while staying small.

**🧭 d. Embedding Model**

* Each chunk is passed through an **embedding model** (e.g., OpenAI Ada, SBERT, HuggingFace, etc.).
* It converts chunks into high-dimensional **vectors** (numerical representations of meaning).

**🗃️ e. Vector Store**

* All the vectors are stored in a **vector database** (like FAISS, Pinecone, Weaviate, ChromaDB).
* Each vector is linked to its original chunk.

This completes **indexing**. You now have a searchable "knowledge brain."

**🔍 2. QUERYING (User Interaction)**

This is the **runtime flow** — when the user asks a question.

**💬 a. User Query**

* A user inputs a **natural language query**.

**🧠 b. Embedding Model**

* The query is also vectorized (converted into a vector) using the **same embedding model** as above.

**🧲 c. Vector Similarity Search**

* The query vector is used to **search the vector store**.
* The system retrieves the **most similar document chunks** — the ones most relevant to the query.

**🧩 3. AUGMENT + GENERATE**

This is where **Retrieval-Augmented Generation** shines.

**🧠 a. Augment the Prompt**

* The retrieved relevant chunks are combined with the user’s query to form a **rich prompt**.

**🤖 b. Generative LLM**

* The augmented prompt is fed into a **Large Language Model** (like GPT, LLaMA, Claude).
* The LLM uses this prompt to **generate a grounded, accurate response**.

A white background with black text

AI-generated content may be incorrect.

**What is LangChain?**

**LangChain** is a **framework** for building applications that use **Large Language Models (LLMs)** like GPT, Claude, LLaMA, etc. It’s especially useful when your app needs more than just a single LLM prompt.

**Why use LangChain?**

It helps you:

* Connect LLMs with **external data** (PDFs, websites, databases, etc.)
* Chain together **multiple prompts and tools**
* Build **RAG (Retrieval-Augmented Generation)** systems
* Add **memory**, **agents**, and **tools** to your LLM
* Prototype and test LLM workflows fast

A diagram of a document loading process

AI-generated content may be incorrect.

Embedding vectors keeps the content and meaning of the text

Text with similar content and meaning will have similar vectors

A diagram of a graph

AI-generated content may be incorrect.

**Left Side: Preprocessing**

* 📄 **Documents**: These are your input sources (e.g., PDFs, web pages, notes).
* 🔪 **Text Splits**: The documents are split into smaller, manageable chunks (e.g., 200-500 characters or tokens). This helps in more efficient embedding and retrieval.
  + Example: A 10-page PDF might be split into 50 text chunks.
* 🧠 **Embed**: Each split is passed into an **embedding model** (like OpenAI, HuggingFace, etc.) that converts it into a high-dimensional **vector**.

**🧠 Middle: Embedding**

* Each text split becomes a vector (a list of numbers) — this is the **Embedding Vector**.
  + For example: [-0.05, -0.0955, ..., 0.0722]
  + These numbers capture the **semantic meaning** of the text, allowing for similarity search later.

**💾 Right Side: Vector Store**

* These vectors are stored in a **vector database** (like FAISS, Chroma, Pinecone).
* Each vector is stored **alongside its corresponding original text**.
* When a user asks a question:
  + Their question is also embedded into a vector.
  + The system performs a **vector similarity search** to find the most relevant text splits based on closeness in vector space.

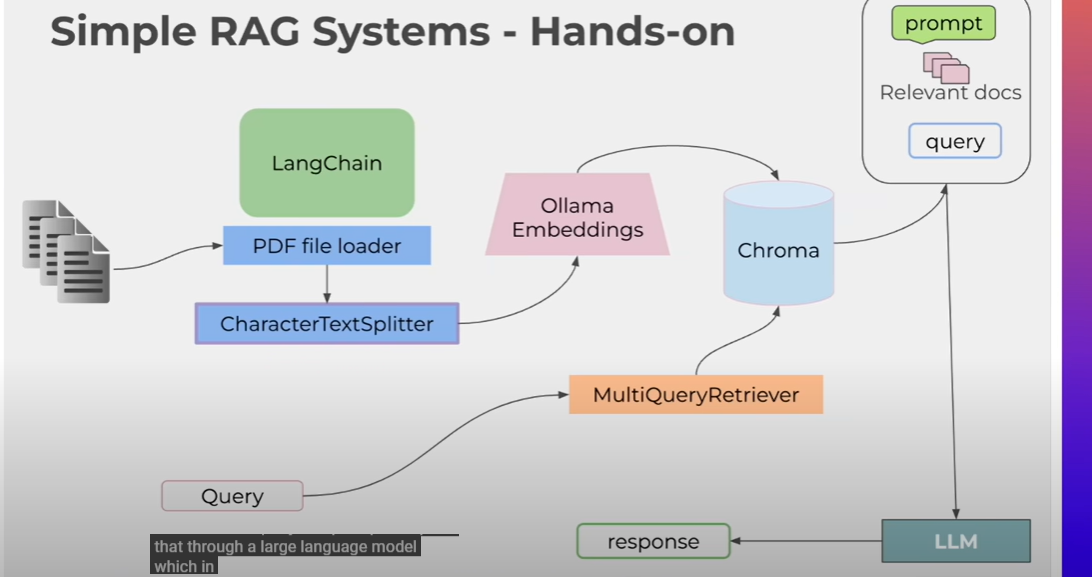
A blue rectangular with green squares and black text

AI-generated content may be incorrect.A diagram of a process

AI-generated content may be incorrect.

**Combined Vector Store + LLM Workflow**

1. **Query/Question (Input)**  
   A user asks a question or inputs a query.
2. **Embedding (Convert to Vector)**  
   The query is passed through an **embedding model**, which converts it into a numerical vector representation.
3. **Vector Store Search**  
   The query vector is compared with all stored vectors in the **vector store (index)** to find the most similar entry.  
   This is done using similarity metrics like cosine similarity.
4. **Pick the Most Similar Entry**  
   The most relevant document or text chunk associated with the closest vector is retrieved.
5. **LLM Processing**  
   The retrieved text chunk **and** the original query are passed to a **Large Language Model (LLM)**.
6. **Result/Answer (Output)**  
   The LLM generates a final **answer** based on both the query and the retrieved context.



**Document Ingestion Flow**

1. **PDF Documents**  
   Input files are PDFs that contain the knowledge base.
2. **LangChain + PDF File Loader**
   * **LangChain** is used as the orchestration framework.
   * The **PDF file loader** reads and loads content from the PDF documents.
3. **CharacterTextSplitter**
   * The loaded PDF content is split into smaller text chunks (e.g., paragraphs or sentences) using the CharacterTextSplitter.
   * This helps manage the token limits and improves chunk-level semantic search.
4. **Ollama Embeddings**
   * Each text chunk is converted into a numerical vector using **Ollama Embeddings**, a local embedding model (like nomic-embed-text or similar).
5. **Chroma**
   * The vectorized chunks are stored in a **Chroma** vector database (a vector store).
   * Chroma allows fast similarity search over the embedded chunks.

**🔍 Query + Retrieval Flow**

1. **Query**
   * The user enters a **query** (e.g., a question).
2. **MultiQueryRetriever**
   * The query is handled by **MultiQueryRetriever**, which may:
     + Reformulate the query into multiple variants (for better retrieval coverage).
     + Search the vector store for relevant chunks.
   * It sends these variants to **Chroma** to retrieve the most relevant text chunks.
3. **Prompt + Relevant Docs**
   * The retrieved documents (relevant chunks) and the original query are **combined into a prompt**.
4. **LLM (Large Language Model)**
   * The prompt is passed to the **LLM** (which could be a local model via Ollama or cloud-based like OpenAI).
   * The LLM uses the context to generate a **response** to the query.
5. **Response**
   * The final answer is returned to the user.

**Structure of Prompt:**

You are a helpful assistant. Use the following documents to answer the question.

Context:

<document chunk 1>

<document chunk 2>

...

<document chunk N>

Question: <user query>

Answer: