Building a RAG pipeline

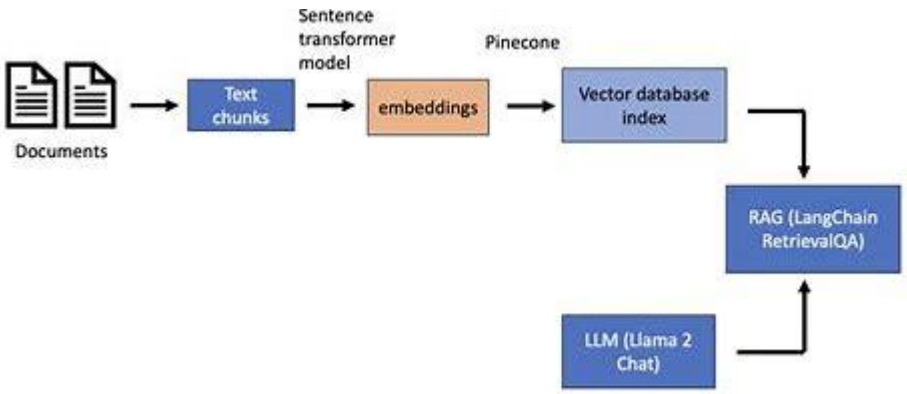
Authors:

1. *Vikram*
2. *Aman*

What is a RAG ?

A RAG (Retrieval-Augmented Generation) pipeline is an AI system that combines information retrieval with text generation to generate more accurate, context-aware responses. Instead of relying solely on a pre-trained LLM (which has a fixed knowledge base), a RAG pipeline retrieves relevant documents from an external knowledge source before generating a response.

This approach significantly reduces hallucination, improves factual accuracy, and allows the LLM to work with up-to-date or domain-specific information.



flow diagram of RAG

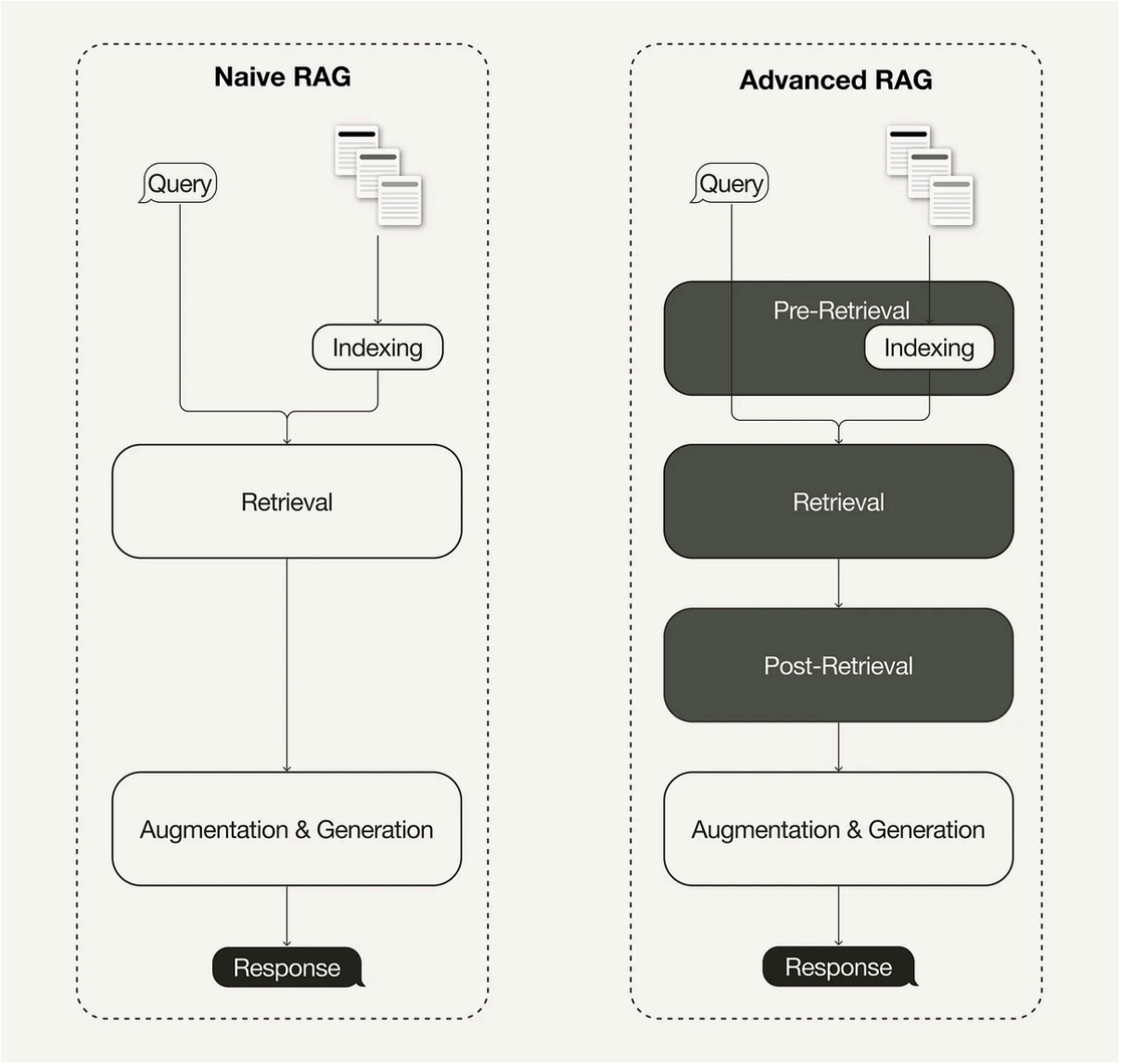
Types of RAG

1. **Naive RAG**

   The Naive RAG technique follows a process that includes indexing, retrieving, augmenting and generation of response.

2. Advance RAG

The Advance RAG technique follows a process that includes pre-retrieval (indexing, Query manipulation, Data modification), Retrieval(Searching and Ranking), Post-retrieval(Re-ranking, Filtering) and Generation(Enhancing, Customization).



Hallucination :

hallucination is a phenomenon wherein a large language model (LLM), often a generative AI chatbot or computer vision tool, perceives patterns or objects that are nonexistent or imperceptible to human observers, creating outputs that are nonsensical or altogether inaccurate.

1. Ollama module

- Ollama is a free tool that lets you run Large Language Models (LLMs) on your own computer.

- By running AI models locally, Ollama reduces latency, enhances performance, and allows for complete customization.

Use the DeepSeek LLM model. Which is run on Ollama server.

Steps to install Ollama tool and run DeepSeek Model

1. Code to install Ollama on Google Colab.

```
# Install Ollama
import os

!curl https://ollama.ai/install.sh | sh

!echo 'debconf debconf/frontend select Noninteractive' | sudo debconf-set-selections
!sudo apt-get update && sudo apt-get install -y cuda-drivers

# Set LD_LIBRARY_PATH so the system NVIDIA library
os.environ.update({'LD_LIBRARY_PATH': '/usr/lib64-nvidia'})
```

2. Run Ollama server

```
!nohup ollama serve &
```

3. Pull DeepSeek model

   Here I use the DeepSeek-r1: 1.5b model because it takes less time on the CPU, you can use as per your choice.

```
!ollama pull deepseek-r1:7b

!ollama pull deepseek-r1:1.5b
```

Next, Import necessary Modules and Packages.

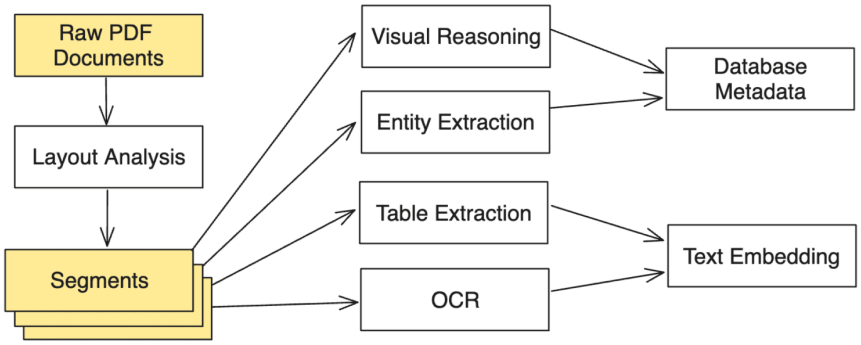
```
# Import necessary packages
import ollama  # Enables interaction with local large language models (LLMs)
import gradio as gr  # Provides an easy-to-use web interface for the chatbot

# Document processing and retrieval

from langchain_community.document_loaders import PyMuPDFLoader, TextLoader
# Extracts text from PDF files for processing

from langchain.text_splitter import RecursiveCharacterTextSplitter
# Splits text into smaller chunks for better embedding and retrieval

from langchain.vectorstores import Chroma  # Handles storage and retrieval of vector embeddings using ChromaDB

# Embedding generation
from langchain_community.embeddings import OllamaEmbeddings
# Converts text into numerical vectors using Ollama's embedding model

import re  # Provides tools for working with regular expressions, useful for text cleaning and pattern matching
```

```
# Import modules

from typing import List

from langchain.embeddings import HuggingFaceEmbeddings
from langchain.chains import RetrievalQA
from langchain.llms import HuggingFacePipeline
from langchain_community.llms import Ollama
from langchain.chains.llm import LLMChain
from langchain.chains.combine_documents.stuff import StuffDocumentsChain
from langchain.prompts import PromptTemplate

from transformers import pipeline

from google.colab import drive
```

- **We mount Google Drive because we use Google Colab to store data on Drive.**

2. Document Transformers

Document Transformer is a component that processes retrieved documents before passing them to the language model. It ensures that the retrieved context is structured, relevant, and optimized for generating high-quality responses.



3. Parsing Files

Code to load data from Google Drive:

```
def read_files():
  print("read all the files in directory")

  drive.mount('/content/drive')

  # Path to your PDF in Colab (adjust as needed)
  # pdf_path = '/content/credit_card.pdf'  # Example: PDF in your Drive
  pdf_path = '/content/WelZin.pdf'

  # PyMuPDFLoader initializes the PDF file
  loader = PyMuPDFLoader(pdf_path)
  # .load() method reads the content of the PDF and extracts its text
  documents = loader.load()
  return documents

read_files()

for doc in read_files():
   print('Print metadata like page number :', doc.metadata)
   print('Print page content : ', doc.page_content)
```

Once the data is loaded, you can transform them to suit your application, or to fetch only the relevant parts of the document. Basically, it is about splitting a long document into smaller chunks which can fit your model and give results accurately and clearly.

4. Splitting

- Text Splitter is a component that divides large documents into smaller, manageable chunks before they are stored in a vector database for retrieval.

- There are several “Text Splitters” in LangChain, you have to choose according to your choice. I chose “RecursiveCharacterTextSplitter”. This text splitter is recommended for generic text. It is parametrized by a list of characters. It tries to split the long texts recursively until the chunks are smaller enough.

Why we need to split

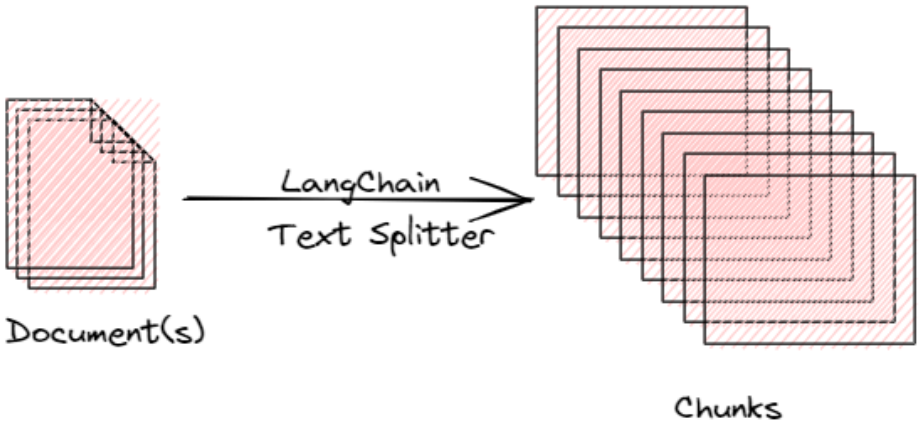
- LLMs have token limits (e.g., Mistral-7B has a context length limit).

- Large documents can contain irrelevant information; splitting helps retrieve only the most relevant parts.

- Improves retrieval accuracy by enabling finer-grained searches instead of retrieving entire documents.

5. Chunking

Breaking down your large data files into more manageable segments.



Why do we need to break down our data?
To provide the LLM with precisely the information needed for the specific task.

Based on the complexity and effectiveness chunking divides into five types:

1. **Fixed Size Chunking**
   It breaks down the text into chunks of a specified number of characters, regardless of their content or structure.

2. **Recursive Chunking**
   We divide the text into smaller chunk in a hierarchical and iterative manner using a set of separators.

3. **Document Based Chunking**
   In this chunking method, we split a document based on its inherent structure.

4. **Semantic Chunking**
   All above three levels deal with content and structure of documents and necessitate maintaining constant value of chunk size. This chunking method aims to extract semantic meaning from embeddings and then assess the semantic relationship between these chunks.

   few concepts to know

   - **buffer_size:** configurable parameter that decides the initial window for chunks.
   - **breakpoint_percentile_threshold:** another configurable parameter. The threshold value to decide where to split the chunk
   - **embed_mode:** the embedding model used.

5. **Agentic Chunking**
   Agentic Chunking is an advanced method of text chunking in Retrieval-Augmented Generation (RAG) pipelines where the chunking strategy dynamically adapts to the content rather than following a fixed rule (like simple token or sentence-based chunking). The term "agentic" refers to the use of an agent-based approach where an AI model, heuristic, or

predefined logic intelligently decides how to segment the text for optimal retrieval and generation.

Here we Recursive chunking to improve retrieval performance.

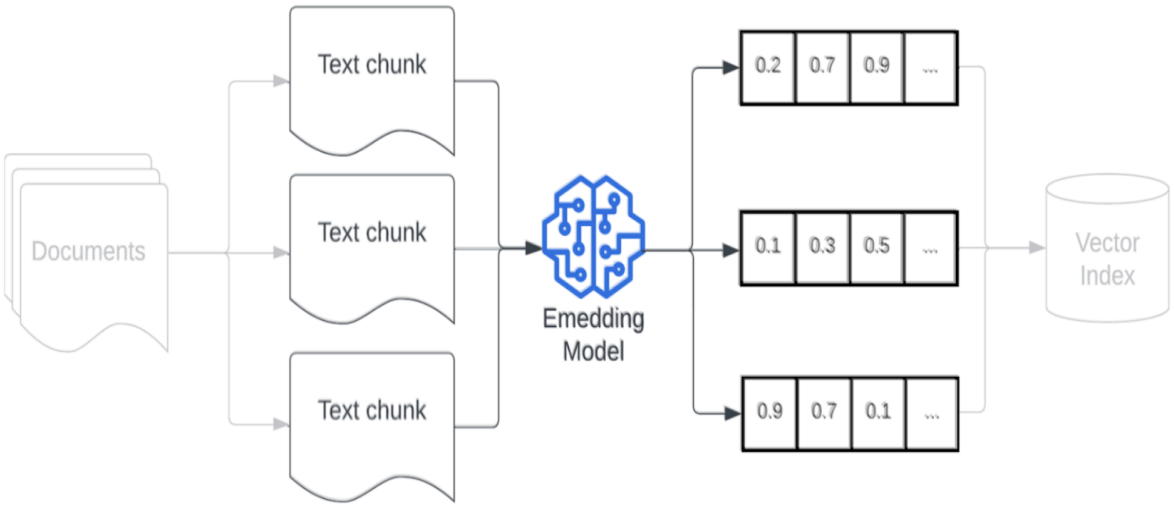
Code for chunking:

```
def chunk_data():
  print("chunking pdf data")

  # RecursiveCharacterTextSplitter splits the PDF into chunks of 500 characters,
  #keeping 100 characters overlap to keep context
  text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
  # Splits the documents into chunks and stores them in chunks object
  chunks = text_splitter.split_documents(read_files())
  return chunks

chunk_data()
```

- Use chuk_size = 500
- Use chunk_overlap = 100

6. Embedding

Embedding transforms each chunk into a dense vector, a numerical representation that captures the semantic meaning of the text. These vectors, typically generated by pre-trained language models, are then stored in a vector database.



Code for embedding:

```
# Embedding
embeddings = OllamaEmbeddings(model="deepseek-r1:1.5b")

print("Embedding Vector:", embeddings)
```

Here, We use OllamaEmbedding and DeepSeek-r1: 1.5b model

Different types of embeddings

1. Word Embedding:

- A word embedding is essentially a vector that represents a specific word in a given language. The dimensionality of this vector can range anywhere from a few hundred to a few thousand dimensions.
- Word embedding lies in their ability to capture semantic relationships between words.

2. Graph Embedding:

- Graph embeddings are another type of embedding that are used to represent graph data. Graphs are data structures that consist of nodes and edges, and they are commonly used to represent relationships between entities.
- A graph embedding is a low-dimensional vector that represents a node in a graph. The goal of a graph embedding is to preserve the structural information of the graph in the low-dimensional space. This means that nodes that are close to each other in the graph should also be close to each other in the embedding space.
-

3. Image Embedding:

Image embeddings are used to represent image data. The idea is similar to word embeddings: a high-dimensional image is represented as a low-dimensional vector. These embeddings can capture visual similarities between images, which can be useful for tasks like image recognition and image clustering.

7. Vector DB

Vector databases store data as high-dimensional vector embeddings, capturing semantic meaning and relationships. They utilize specialized indexing techniques like hashing, quantization, and graph-based methods to enable fast querying and similarity searches.

Code for Vector Store

```
vectorstore = Chroma.from_documents(
  documents=chunks, embedding=embeddings, persist_directory="./chroma_db"
)

print("Total Documents in Vector Store:", vectorstore._collection.count())

retrieved_docs = vectorstore.similarity_search("what is the joining date", k=4)
print("Retrieved Document:", retrieved_docs[1].page_content)

retriever = vectorstore.as_retriever()
```

Different type of Vector Database:

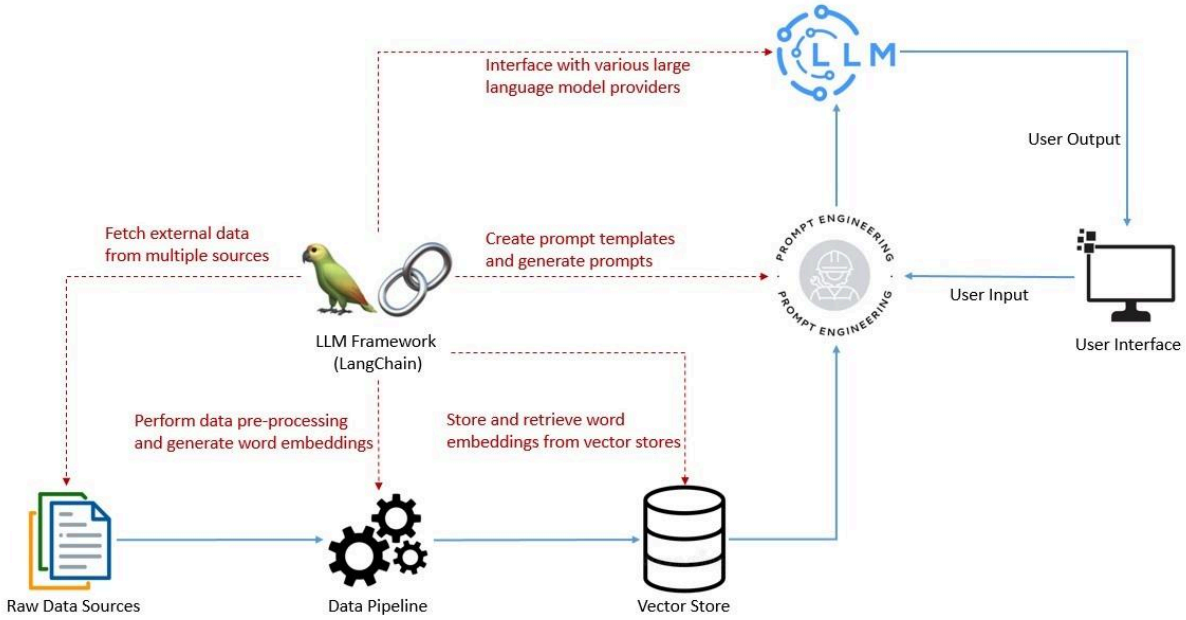
1. **Chroma**
   Chroma is an open-source embedding database that simplifies the development of large language model (LLM) applications. It supports LangChain (Python and JavaScript) and LlamaIndex, making it easy to manage text documents, convert text to embeddings, and perform similarity searches.

2. **Pinecone**
   Pinecone is a managed vector database platform designed for high-dimensional data. It offers features like real-time data ingestion, low-latency search, and integration with LangChain. Pinecone is highly scalable.

3. **Faiss**
   Faiss, developed by Facebook, is an open-source library for fast similarity search and clustering of dense vectors. It supports Python/NumPy integration and GPU execution, making it suitable for handling large vector sets that may exceed RAM capacity.

8. LangChain module

LangChain is a framework for developing applications powered by language models (LLMs) like GPT. It simplifies the integration of LLMs with other tools, such as data sources, APIs, and user interactions, to build robust AI applications.



Key Concepts in LangChain

1. Retrieval Chains:
Facilitate searching and retrieving relevant information from external data sources (e.g., vector databases) to augment the LLM's responses.

2. Memory:
Enable the LLM to remember information between user interactions for a more context-aware experience.

3. Agents:
Dynamically decide which tools to use and how to use them based on the user query.

Code for LLM:

```
# Build LLM chain and Q/A

retriever = vectorstore.as_retriever()

prompt = """
    Use the following context to answer the question.
    Context: {context}
    Question: {question}
    Answer:"""

QA_PROMPT = PromptTemplate.from_template(prompt)

llm = Ollama(model="deepseek-r1:7b")
llm_chain = LLMChain(llm=llm, prompt=QA_PROMPT)

combine_documents_chain = StuffDocumentsChain(llm_chain=llm_chain, document_variable_name="context")
```

9. QA Retrieval

QA Retrieval refers to the process of retrieving relevant documents or passages from a knowledge source (such as a vector database or document index) to assist in answering a user's query.

Code for QA retrieval:

```
qa = RetrievalQA(combine_documents_chain=combine_documents_chain, retriever=retriever)

response = qa("what is stipend offered int this internship offer letter")
print(response)
```

How QA Retrieval Works in RAG

1. **User Query Input:**
   The user provides a question (e.g., "What are the benefits of deep learning?").

2. **Embedding Generation:**
   The query is converted into an embedding using a pre-trained model (e.g., Hugging Face Embeddings).

3. **Retrieval from a Knowledge Base:**
   - The system searches for semantically similar documents in a vector database (e.g., FAISS, Chroma) or traditional text indices.
   - It returns the most relevant passages based on similarity scores.

4. **Context Injection into LLM:**
   The retrieved documents are fed into a large language model (LLM) as additional context.

5. **Answer Generation:**
   The LLM generates a response using both the retrieved documents and its own trained knowledge.

References

1. https://medium.com/@akriti.upadhyay/implementing-rag-with-langchain-and-hugging-face-28e3ea66c5f7
2. https://www.datacamp.com/tutorial/deepseek-r1-ollama
3. https://medium.com/ai-agent-insider/developing-rag-systems-with-deepseek-r1-ollama-66a520bf0b88
4. https://www.analyticsvidhya.com/blog/2025/01/rag-system-using-deepseek/
5. https://python.langchain.com/docs/introduction/#architecture