Artificial Intelligence (Machine Learning & Deep Learning) [Course]

Week 19 – LangChain - Retrieval Augmented Generation (RAG)

[See examples / code in GitHub code repository]

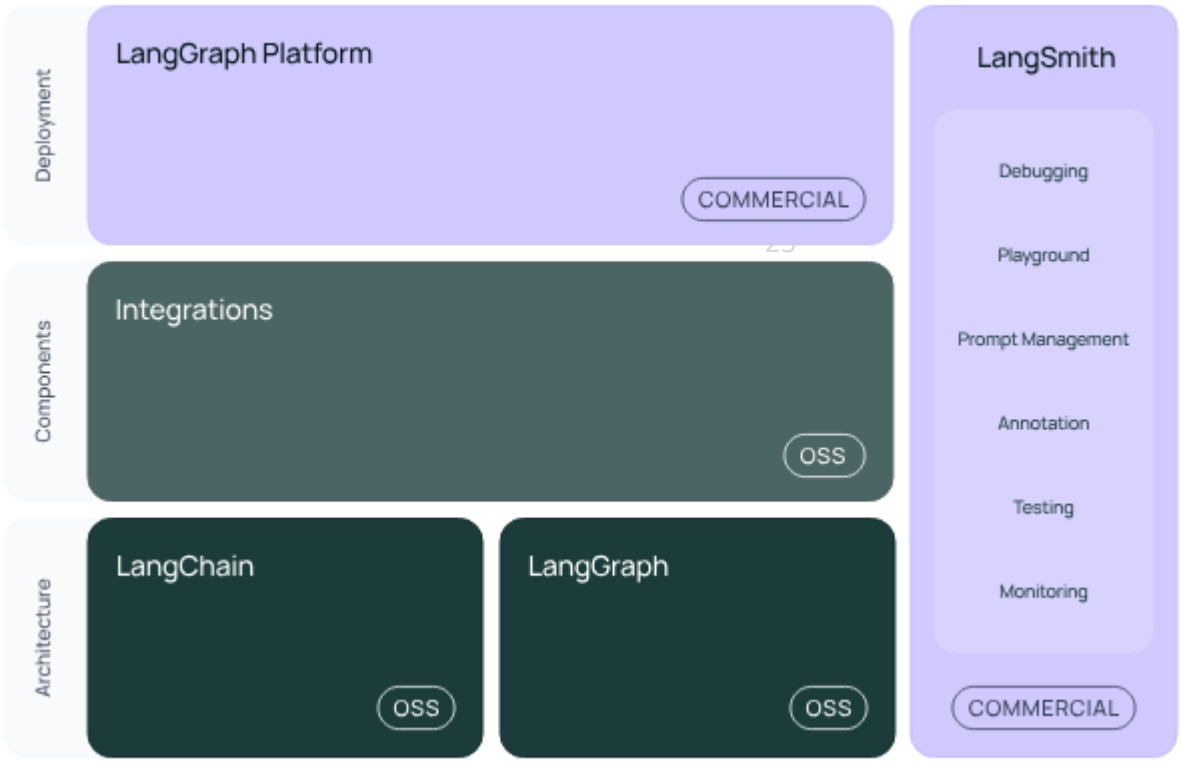
It is not about Theory, it is 20% Theory and 80% Practical – Technical/Development/Programming [Mostly Python based]

LangChain - Foundation

LangChain is a framework for developing applications powered by large language models (LLMs).

LangChain simplifies every stage of the LLM application lifecycle:

- **Development**: Build your applications using LangChain's open-source components and third-party integrations. Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.
- **Productionization**: Use LangSmith to inspect, monitor and evaluate your applications, so that you can continuously optimize and deploy with confidence.
- **Deployment**: Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Platform.

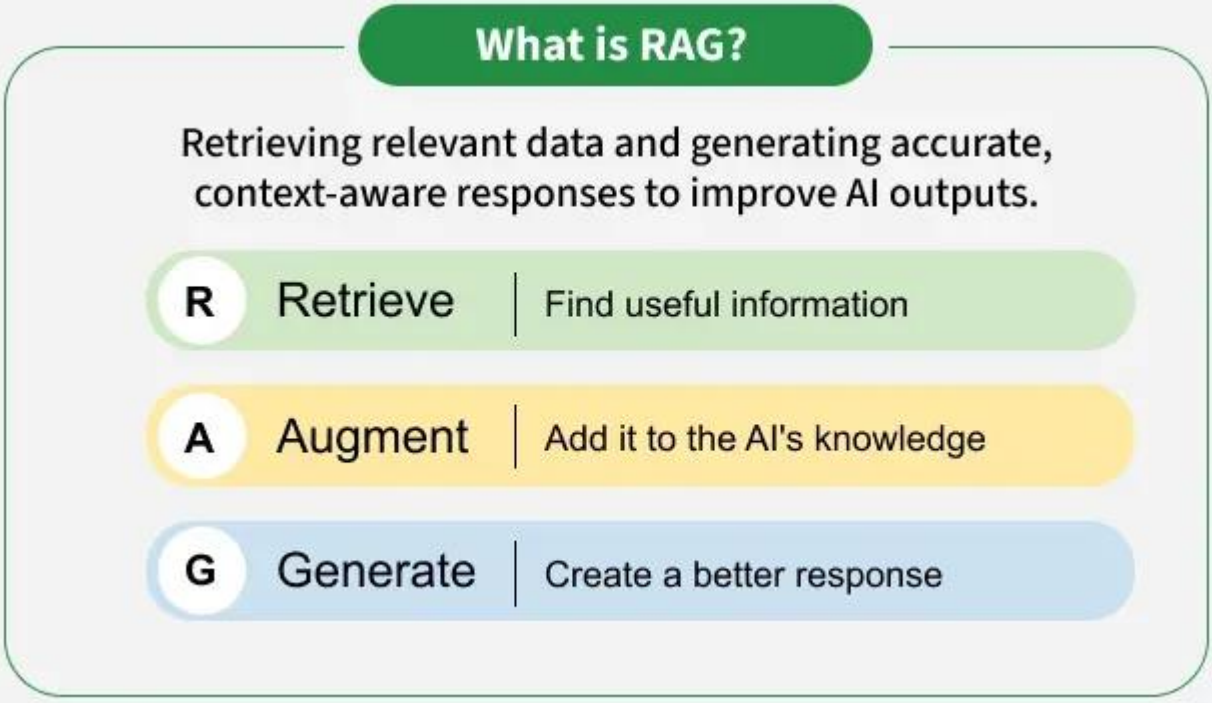


Reference:

https://python.langchain.com/docs/introduction/
https://www.langchain.com/
https://aws.amazon.com/what-is/langchain/

Retrieval-Augmented Generation (RAG) - Foundation

Retrieval-augmented generation (RAG) is an innovative approach in the field of natural language processing (NLP) that combines the **strengths of retrieval-based and generation-based models to enhance the quality of generated text.**



Why is Retrieval-Augmented Generation important?

In traditional LLMs, the model generates responses based solely on the data it was trained on, which may not include the **most current information or specific details required for certain tasks. RAG addresses this limitation by incorporating a retrieval mechanism that allows the model to access external databases or documents in real-time.**

Reference:

https://www.geeksforgeeks.org/nlp/what-is-retrieval-augmented-generation-rag/
https://aws.amazon.com/what-is/retrieval-augmented-generation/
https://cloud.google.com/use-cases/retrieval-augmented-generation?hl=en

LangChain-RAG - Coding – Development Case

Build a Retrieval Augmented Generation (RAG) App

One of the most powerful applications enabled by LLMs is sophisticated question-answering (Q&A) chatbots. These are applications that can answer questions about specific source information. These applications use a technique known as Retrieval Augmented Generation, or RAG.

This tutorial will show how to build a simple Q&A application over a text data source. Along the way we'll go over a typical Q&A architecture and highlight additional resources for more advanced Q&A techniques. We'll also see how LangSmith can help us trace and understand our application. LangSmith will become increasingly helpful as our application grows in complexity.

Practical Development Case Study

Reference:
https://python.langchain.com/docs/tutorials/rag/

Sample Code:

https://colab.research.google.com/github/langchain-ai/langchain/blob/master/docs/docs/tutorials/rag.ipynb
https://github.com/langchain-ai/langchain/blob/master/docs/docs/tutorials/rag.ipynb

Typical RAG application

A typical RAG application has two main components:

Indexing: a pipeline for ingesting data from a source and indexing it. *This usually happens offline.*

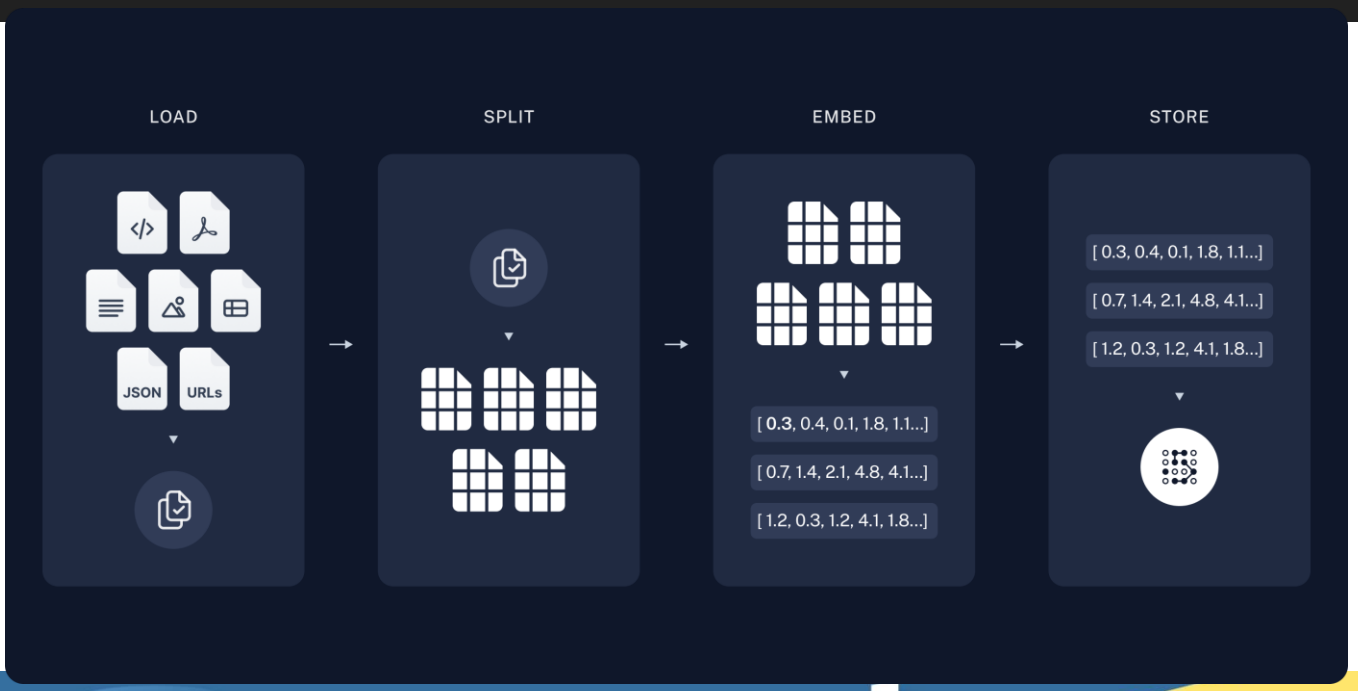
Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model.

Part 1 - 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.

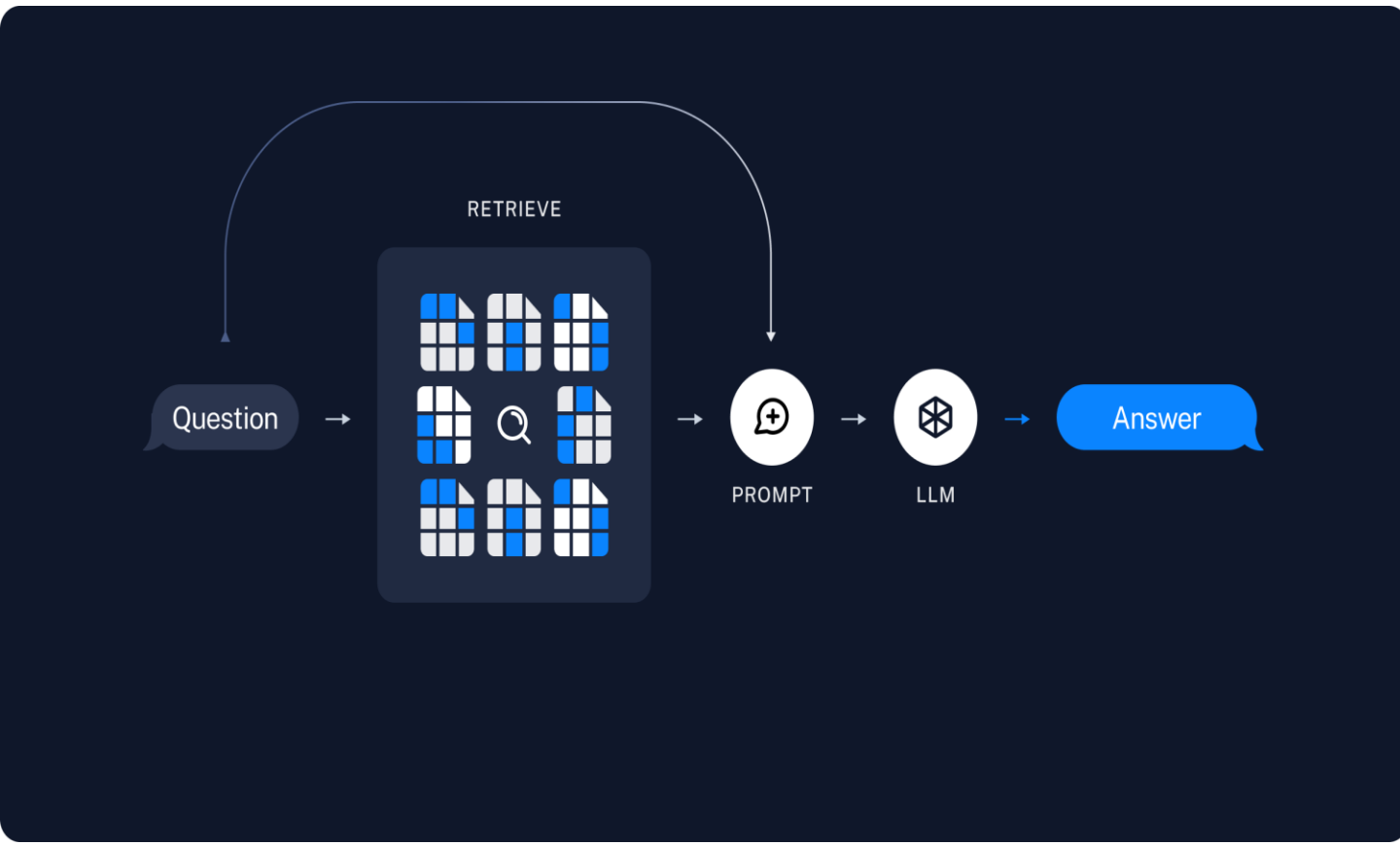


Typical RAG application

Part -2: Retrieval and generation

Retrieve: Given a user input, relevant splits are retrieved from storage using a Retriever.

Generate: A ChatModel / LLM produces an answer using a prompt that includes both the question with the retrieved data



Practical Development Case Study

Reference:
https://python.langchain.com/docs/tutorials/rag/

Sample Code:

https://colab.research.google.com/github/langchain-ai/langchain/blob/master/docs/docs/tutorials/rag.ipynb
https://github.com/langchain-ai/langchain/blob/master/docs/docs/tutorials/rag.ipynb

Typical RAG application - **Document loaders**

Document loaders

DocumentLoaders load data into the standard LangChain Document format.

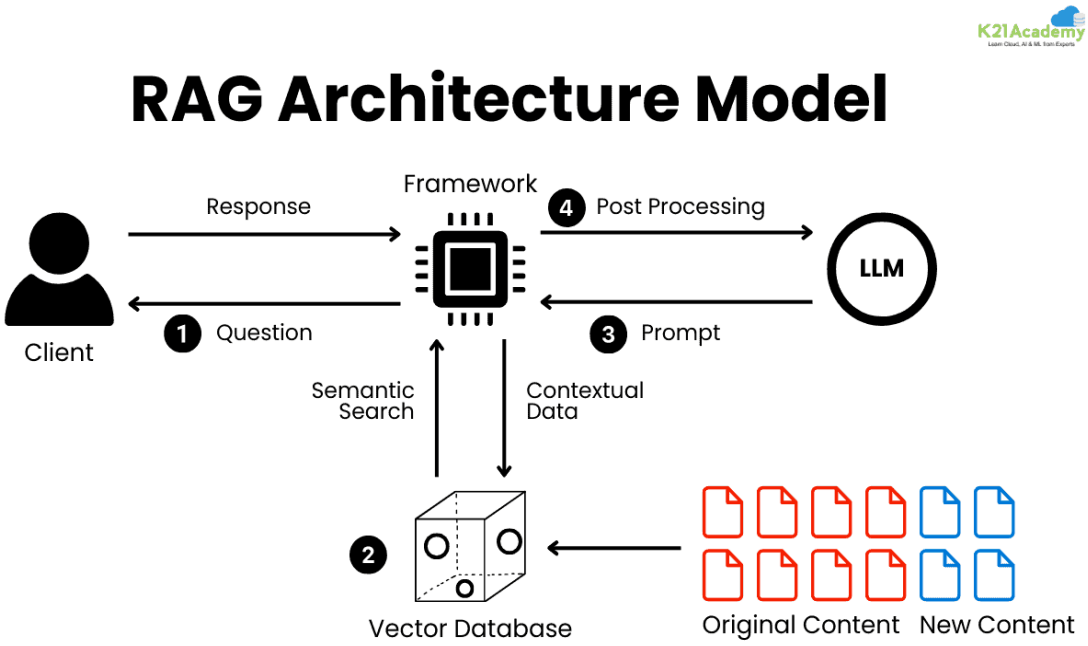
Each DocumentLoader has its own specific parameters, but they can all be invoked in the same way with the .load method. An example use case is as follows:

```
from langchain_community.document_loaders.csv_loader import CSVLoader

loader = CSVLoader(
    ...  # <-- Integration specific parameters here
)
data = loader.load()
```

Vector stores

A vector store stores embedded data and performs similarity search.



Practical Development Case Study

Document Loader - Reference:
https://python.langchain.com/docs/integrations/document_loaders/
Vector stores - Reference:
https://python.langchain.com/docs/integrations/vectorstores/

Thank you - for listening and participating

- ❑**Questions / Queries**
- ❑**Suggestions/Recommendation**
- ❑**Ideas…..?**

Shahzad Sarwar
Cognitive Convergence
https://cognitiveconvergence.com
shahzad@cognitiveconvergence.com
voice: +1 4242530744 (USA) +92-3004762901 (Pak)