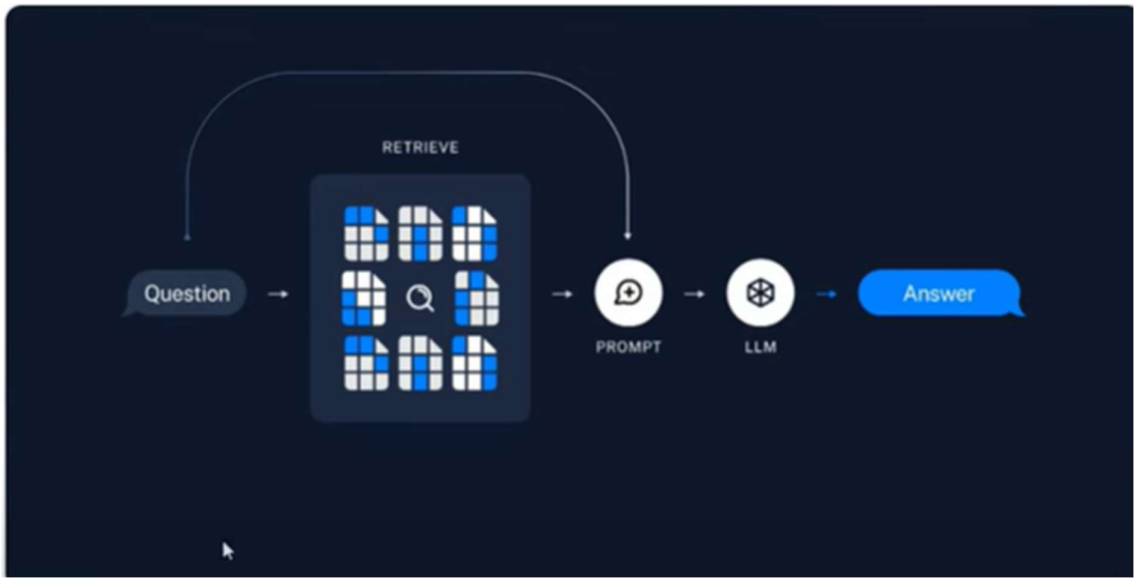
Conversational RAG based chatbot Using LangChain

Understanding RAG

Retrieval-Augmented Generation (RAG) is a technique that enhances the retrieval process from a custom vector database by appending the retrieved content (referred to as "context") to the prompt before sending it to a Large Language Model (LLM). This technique is particularly valuable in scenarios where:

- The LLM lacks knowledge about the question being asked.
- The LLM is not updated with the latest information to answer the query.
- Customized answers are required, referencing a personal or organizational knowledge base.

In such cases, the relevant content (context) is retrieved from a vector database and incorporated into the prompt to improve the LLM's response accuracy.



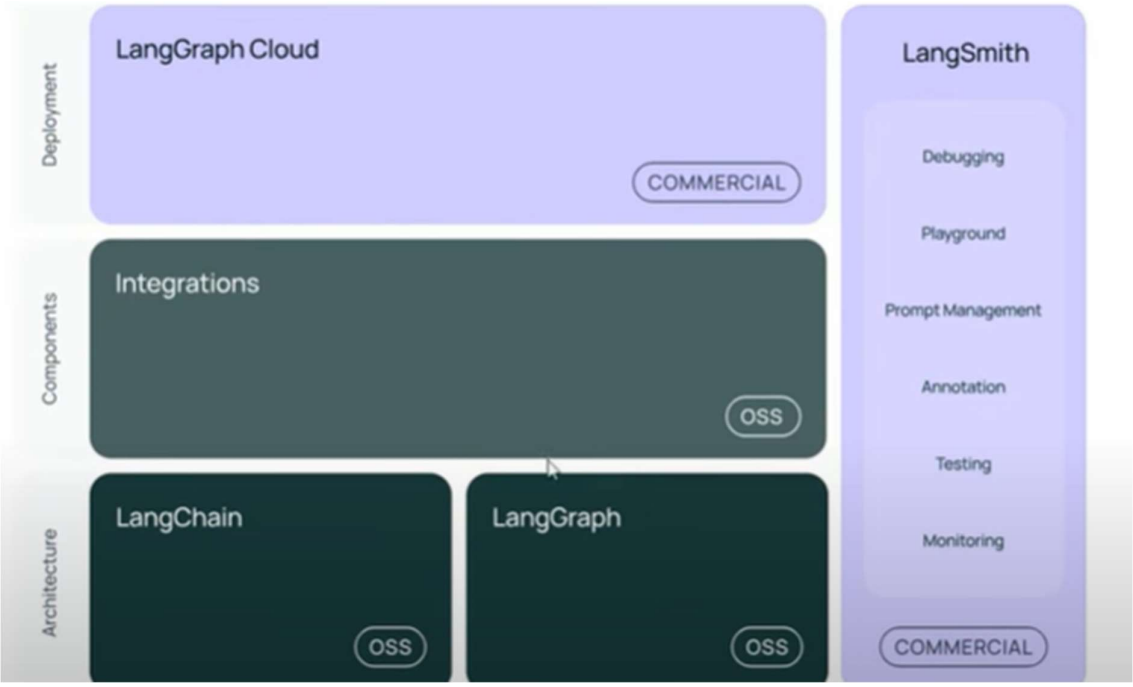
Introduction to Langchain and LCEL

LangChain is a framework designed to seamlessly integrate LLMs into applications. It provides flexible abstractions and an AI-first toolkit, making it a preferred choice for GenAI developers. LangChain includes several open-source libraries:

1. **LangChain-Core**:
    - Provides base abstractions for LLMs and supports the LangChain Expression Language (LCEL).
2. **LangChain-Community**:

- Offers community-driven integrations, such as "LangChain-Anthropic" and "LangChain-OpenAI."

3. **LangChain**:
   - Includes chains, agents, and retrieval strategies, forming the cognitive architecture of applications.
4. **LangGraph**:
   - Enables multi-actor applications using edges and nodes in a graph. It can be used independently or integrated with LangChain.
5. **LangServe**:
   - Facilitates the deployment of LangChain as REST APIs.
6. **LangSmith**:
   - A platform for debugging, testing, validating, and monitoring LLMs.



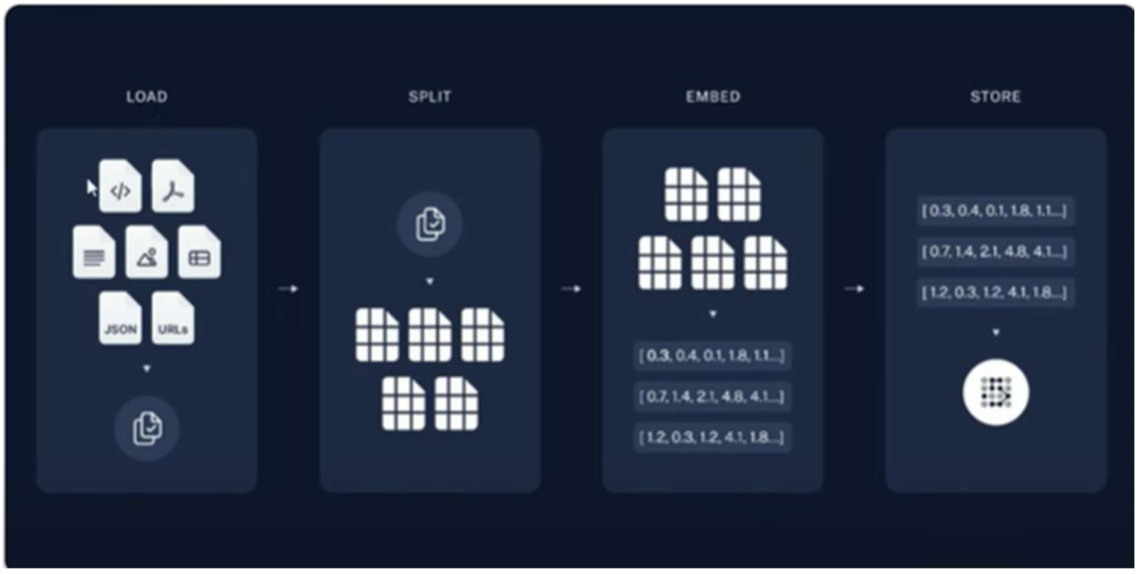
Summarized Flow

1. **Data Gathering**
2. **Data Processing**
3. **Document Splitting**:
   - Splitting data into chunks using chunking configurations.
4. **Vector Embeddings**:
   - Converting high-dimensional data into low-dimensional space to capture semantic relationships.

5. **Context Retrieval**:
    - Using the constructed vector store as a retriever.
6. **RAG Chain Construction**:
    - Combining retrieved context and the query into a prompt template for LLM invocation.

Document Processing And Vector Database Construction

A) Document Splitting: The documents need to be split into chunks and being applied vector embeddings so that they can be used for semantic search by storing into Vector Database. It is necessary to split the documents and compress them into embedded vectors which can then be indexed for context retrieval. Vectors capture the semantic meaning of the documents.




```python
from langchain_community.document_loaders import PyPDFLoader, Docx2txtLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
from typing import List
from langchain_core.documents import Document

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len
)
```

Langchain library provides Text splitters responsible for dividing the document into chunk which can be configured with fixed size and overlapping values.

*Chunk size and overlap values are necessary for getting the context right and precise and hence can be experimented with different values.

B) Vector Embeddings: Embeddings are necessary for translation of the document chunks into vectors which can then be stored in vector stores. Embeddings are necessary to convert the High dimensional data like images, videos etc into low dimensional space for capturing semantic relationships, improved model performance and dimensionality reduction.

```
embeddings = OpenAIEmbeddings()

# 4. Embedding Documents

document_embeddings = embeddings.embed_documents([split.page_content for split in splits])
```

C) Vector Database Construction: A vector database can be constructed once the chunking and the embeddings are realized. The document splits and embedding function are passed in the Vector store API to create a new chroma store.

```
from langchain_chroma import Chroma

embedding_function = OpenAIEmbeddings()
collection_name = "my_collection"
vectorstore = Chroma.from_documents(collection_name=collection_name, documents=splits, embedding=embedding_function, persist_directory="./chroma
#db.persist()

print("Vector store created and persisted to './chroma_db'")
```

Prompt Templates

The concept of having dynamic prompts which can take the value of “Question”, “Retrieved Context” and “Chat History” is managed using prompt templates. These templates have placeholders for the mentioned values which are passed at the time of invocation of LLM.

Eg.

```
[ ]
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Define the prompt
prompt = ChatPromptTemplate.from_template("Tell me a short joke about {topic}")
```

LLM Invocation with value:-

```
prompt.invoke({"topic": "programming"})
```

RAG Chains

In the below example Chat type prompt template is imported from Langchain where the placeholders can be specified for the “Context from the retriever” and the “Questions asked”.

So, the prompt can be constructed using these values which is then fed to the LLM to obtain the final response.

```
from langchain_core.prompts import ChatPromptTemplate
template = """Answer the question based only on the following context:
{context}

Question: {question}

Answer: """
prompt = ChatPromptTemplate.from_template(template)
```

```
from langchain.schema.runnable import RunnablePassthrough
rag_chain = (
  {"context": retriever, "question": RunnablePassthrough()} | prompt
)
rag_chain.invoke("When was GreenGrow Innovations founded?")
```

As one can see the RAG chain realized through pipe “|” where the context obtained from the Retriever and the question will be passed at the time of invoking the chain via “rag_chain.invoke()” function. “RunnablPassthrough() function is nothing but the human chat/asked question at the time of invocation.

Combining everything together

Adding the LLM and the final response parser to the RAG chain will yield:-

```
rag_chain = (
    {"context": retriever | docs2str, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
question = "When was GreenGrow Innovations founded?"
response = rag_chain.invoke(question)
print(response)
```

Conversational RAG

As we have constructed a functional RAG chain in the previous steps, the next step is to enable the RAG chain to become more conversational so that follow up questions can be asked.

Consider an example where we have many documents about the various "International Organizations ". Now a user asks the chatbot.

Human: "Where is the next United Nations Session going to be held."

Chatbot: "It is going to be held in NewYork"

Human: When was the monument created. [At this point since we have many documents for the historical monument data, the RAG may provide us context for more than one monument as the same was not explicitly specified. Hence it becomes necessary to pass the chat history].

This brings us to the concepts of "History aware Retrievers and Contextual Prompts"

Contextual Prompts

The concept of contextual prompts makes it easy and accurate for the LLM to answer follow up questions. In the previous example we came across conversation like

Human: "Where is the next United Nations Session going to be held."

Chatbot: "It is going to be held in NewYork."

Now if a follow up question is asked:-

Human: "Where is it head quartered?" as a follow up question then It may return a list of documents with more than one Organization details. To refine this answer it is necessary that we maintain a contextualized prompt that takes the chat history and a contextualized system prompt as shown below:-

```
from langchain_core.prompts import MessagesPlaceholder
contextualize_q_system_prompt = (
    "Given a chat history and the latest user question "
    "which might reference context in the chat history, "
    "formulate a standalone question which can be understood "
    "without the chat history. Do NOT answer the question, "
    "just reformulate it if needed and otherwise return it as is."
)
```

```
contextualize_q_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", contextualize_q_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)
```

Now when the RAG chain is invoked with the dynamic inputs of Chat history and the question. The same can then be piped with the LLM and the StrOutputParser() function to create desired response.

```
contextualize_chain = contextualize_q_prompt | llm | StrOutputParser()
contextualize_chain.invoke({"input": "Where it is headquartered?", "chat_history": chat_history})
```

So the question “Where it is headquartered” will be converted into a more contextual prompt as:-

Normal Prompt : “Where it is headquartered”?

Contextualized Prompt : “Where is United Nation Organization headquartered?”

History Aware retriever

Langchain provides a similar class named : “create_history_aware_retriever” which takes the contextualized prompt, question and chat history as the input and creates a more contextualized retriever which is more accurate in response.

```
from langchain.chains import create_history_aware_retriever
history_aware_retriever = create_history_aware_retriever(
    llm, retriever, contextualize_q_prompt
)
history_aware_retriever.invoke({"input": "Where it is headquartered?", "chat_history": chat_history})
```

Normal Prompt : “Where it is headquartered”?

Contextualized Prompt : “Where is United Nation Organization headquartered?”

```
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain

qa_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful AI assistant. Use the following context to answer the user's question."),
    #  ("system", "Tell me joke on Programming"),
    ("system", "Context: {context}"),
    MessagesPlaceholder(variable_name="chat_history"),
    ("human", "{input}")
])

question_answer_chain = create_stuff_documents_chain(llm, qa_prompt)

rag_chain = create_retrieval_chain(history_aware_retriever, question_answer_chain)
```

The final rag_chain can be constructed with “create_stuff_documents_chain()” call and when invoked with the history_aware_retriever and the question_answer_chain we get the final response

```
rag_chain.invoke({"input": "Where it is headquartered?", "chat_history":chat_history})
```


```
rag_chain.invoke({"input": "Where it is headquartered?", "chat_history":chat_history})
```