EL HARKAOUI Chaymae -- 01-03-2025

LangChain

What is LangChain ?

LangChain is a -- Python - JavaScript - **Framework** designed to help building **applications powered by LLMs** such as **OpenAI's GPT** - **Google's Gemini** - **Meta's LLaMA** - and other similar models.

LangChain allows developers to :

- **Integrate** LLMs with **external data** -- APIs - databases - documents ...
- Implement **memory** in chat applications - so conversations are stateful
- Use **chains of LLM calls** to enhance reasoning and problem-solving.
- **Combine** different models - tools - agents for complex tasks.

LangChain makes it easy to build **AI-powered applications** like chatbots - document summarizers - question-answering systems - autonomous agents ...

LangChain was created in **October 2022** by **Harrison Chase** as an open-source framework to simplify working with LLMs. Initially, it focused on providing **wrappers for LLMs and prompt templates**, but as AI adoption surged especially after ChatGPT's launch - it rapidly evolved into a **full ecosystem**.

By early **2023** - LangChain introduced **memory - chains - agent capabilities** - enabling chatbots - AI assistants - automation workflows. Its integration with **vector databases** like FAISS and Pinecone and support for **multiple LLM providers** - OpenAI - Hugging Face - Cohere ... made it widely adopted for **retrieval-augmented generation RAG and enterprise AI applications**.

By **2024** - LangChain had become a **leading AI development framework**, allowing developers to build **scalable - intelligent AI-driven applications** with ease. Its future promises **more advanced AI agents - better on-premise model support - improved**

enterprise scalability - solidifying its role in the **next generation of AI-powered software**.

Key Concepts in LangChain

LangChain is built on five main **pillars** :

LLM Wrappers --- Easily interact with models like OpenAI's GPT - Claude - LLaMA ...
Prompt Management --- Helps with prompt engineering and formatting.
Memory --- Maintains context across multiple interactions.
Data Connectivity --- Retrieves and processes structured or unstructured data.
Agents & Chains --- Enables AI systems to make decisions and take actions.

Core Modules in LangChain

1 - Language Model Wrappers
2 - Prompt Templates
3 - Chains

LLM Wrappers

LLM Wrappers provide a **unified interface** for **interacting** with various **language models** - OpenAI GPT - Anthropic Claude - Cohere - Hugging Face models ...

Why to use ?

- Allow seamless **switching** between different models
- **Standardize** API calls and response handling
- **Reduce boilerplate code** when working with multiple LLM providers

Example without LangChain ❌

To call OpenAI's GPT model directly - we have to write custom API calls.

Problems

- Requires **manual** API handling
- Switching between models requires **rewriting code**

In [ ]:

```python
import openai

# Our OpenAI API key
openai.api_key = 'sk-projKuYvXPYyKUAbLyZd0oxyfxfpNqMCUA'

# Call the Chat API
response = openai.completions.create( model="gpt-3.5-turbo", prompt="What is LangChain?", max_tokens=100 )
```

```python
# Print the response
print(response.choices[0].text.strip())
```

Example with LangChain ✅

Advantages

- Less boilerplate code
- Easily interchangeable models
- Integrated with other LangChain tools

In [ ]:

```python
from langchain_community.chat_models import ChatOpenAI

# Initialize ChatOpenAI with our API key
llm = ChatOpenAI(model="gpt-3.5-turbo", openai_api_key="svrFJ0ix87AZiZ-CkMdYvXPYyKUAbLyZd0oxyfxfpNqMCUA")

# Prepare the message
messages = [ {"role": "user", "content": "What is LangChain?"} ]

# Invoke the model
response = llm.invoke(messages)

# Print the response
print(response)
```

Prompt Templates

Prompt Templates allow us to dynamically **format** and manage **prompts** in a **structured way**.

Why to use ?

- **Prevent** redundant prompt writing
- Ensure **consistent structure** in requests
- Allow **parameterized inputs** for efficiency

Example without LangChain ❌

Problems

- Hardcoded prompts
- Difficult to scale for multiple topics

In [ ]:

```
user_input = "Quantum Computing"

prompt = f"Explain {user_input} in simple terms."

response = openai.ChatCompletion.create( model="gpt-4", messages=[{"role": "user", "content": prompt}] )

print(response["choices"][0]["message"]["content"])
```

Example with LangChain ✅

Advantages

- Reusability for different inputs
- Standardized prompt structures
- Easy modifications and scalability

In [ ]:

```
from langchain.prompts import PromptTemplate

template = PromptTemplate( input_variables=["topic"], template="Explain {topic} in simple terms.")

formatted_prompt = template.format(topic="Quantum Computing")

print(formatted_prompt)
```

```
Explain Quantum Computing in simple terms.
```

Chains

In LangChain - a **chain** means a **sequence of steps** that process **input** and generate **output**.

A **basic** chain **links** a **prompt** to an **LLM**- It links a **prompt template** with an **LLM** to form a **structured pipeline** for **generating responses**.

A **complex** chain can **combine multiple steps** - like retrieving data - applying logic - using different models...

Think of it like a conveyor belt : **User input query** → **Prompt Formatting** → **LLM Processing** → **Output Generation**

Chains allow you to **connect multiple components** - LLMs - memory - tools ... into a **single workflow**.

Example without LangChain ❌

If we want to take a user input - format a prompt - get an LLM response - we need to :

Problems

- Each step - formatting → sending → retrieving - is manually coded
- Hard to extend for complex workflows

In [ ]:

```python
user_input = "Neural Networks"

prompt = f"Explain {user_input} in simple terms."

response = openai.ChatCompletion.create( model="gpt-4", messages=[{"role": "user", "content": prompt}] )

print(response["choices"][0]["message"]["content"])
```

Example with LangChain ✅

Advantages

- Automates chaining of prompts and responses
- Easily extendable with memory - tools and multiple steps
- Cleaner - reusable code

In [ ]:

```
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

llm = OpenAI(model_name="gpt-4", openai_api_key="our_api_key")

template = PromptTemplate( input_variables=["topic"],template="Explain {topic} in simple terms." )

# A chain that connects a Langauge Model with a structured prompt
chain = LLMChain(llm=llm, prompt=template)

response = chain.run("Neural Networks")

print(response)
```

Memory

Memory allows LLMs to remember **previous conversations** and maintain context.

Why to use ?

- Enables stateful conversations
- Avoids repetition in chatbot applications
- Makes LLMs behave more like a human assistant

Example without LangChain ❌

Each message must contain context **manually**.

Problems

- We must manually track the conversation history

- Becomes inefficient for long interactions

In [ ]:

```
# This list defines a structured conversation history for the chatbot
# Each dictionary in the list represents a message in the conversation

messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello!"},
    {"role": "assistant", "content": "Hi! How can I help you?"},
    {"role": "user", "content": "What was my first message?"}
]

response = openai.ChatCompletion.create( model="gpt-4", messages=messages )

print(response["choices"][0]["message"]["content"])
```

Example with LangChain ✅

Advantages

- Automatic conversation tracking
- Supports long-term memory
- Scalable for chatbots and personal assistants

In [ ]:

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
from langchain.llms import OpenAI

llm = OpenAI(model_name="gpt-4", openai_api_key="our_api_key")

memory = ConversationBufferMemory()

conversation = ConversationChain(llm=llm, memory=memory)

print(conversation.run("Hello!"))
print(conversation.run("What was my first message?"))
```

Agents

Agents allow an **LLM** to **interact** with **external tools** and dynamically decide which tool to use

Why to use ?

- Enables AI-driven decision-making
- Connects LLMs with APIs - databases - web scraping
- Reduces the need for hardcoded responses

Example without LangChain ❌

Problems

- Manually coded logic for each tool
- Hard to scale with multiple tools

In [ ]:

```python
def get_weather(city):
    return f"The weather in {city} is sunny."

query = "What's the weather in Paris?"
if "weather" in query:
    response = get_weather("Paris")

print(response)
```

Example with LangChain ✅

Advantages

- Dynamically selects the correct tool
- Automates AI-driven decision-making
- Scalable for multiple tools - web search - APIs

In [ ]:

```python
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool

def get_weather(city):
    return f"The weather in {city} is sunny."

weather_tool = Tool(
    name="WeatherAPI",
    func=get_weather,
    description="Fetches weather data for a given city."
)

agent = initialize_agent(
    tools=[weather_tool],
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

response = agent.run("What's the weather in Paris?")
print(response)
```

Indexes

In LangChain - an **index** is a **structured way** to **store - organize - retrieve information** efficiently when working with **large datasets** or **document collections**.

Indexes are especially useful for retrieval-augmented generation RAG - where an LLM **fetches** relevant **information** before **generating a response**.

Why to use ?

- Efficiently **search large datasets** instead of scanning everything
- Improve response accuracy by **retrieving relevant documents** before calling an LLM
- Handle **knowledge retrieval** for applications like chatbots - question answering - document search ...

Types of Indexes in LangChain ?

- **Vector Index** - Embedding-Based
- **Keyword Index** - Text-Based
- **Structured Index** - SQL-Based

Vector Index

It Converts **text** into **vector embeddings** and **stores** them in a **vector database** like FAISS - Pinecone - Weaviate - ChromaDB ...
It Uses **similarity search** to find the most relevant data points.
It is Ideal for semantic search - Q&A - knowledge retrieval.

In [ ]:

```python
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

# Create an embedding model
embedding_model = OpenAIEmbeddings()

# Sample documents
documents = ["LangChain helps build LLM applications.", "Vector databases store embeddings for search."]

# Convert documents into a FAISS vector index
vector_index = FAISS.from_texts(documents, embedding_model)

# Retrieve relevant information
query = "What does LangChain do?"
similar_docs = vector_index.similarity_search(query)

print(similar_docs[0].page_content)
```

Keyword Index

It **Stores documents** in a simple **text-based format**.

Uses **traditional keyword matching** for **search**.

Best for smaller datasets where full-text search is enough.

In [ ]:

```
from langchain.indexes import VectorstoreIndexCreator
from langchain.document_loaders import TextLoader

# Load documents from a text file
loader = TextLoader("documents.txt")

# Create a keyword-based index
index = VectorstoreIndexCreator().from_loaders([loader])

# Query the index
response = index.query("What is LangChain?")
print(response)
```

Structured Index

It Uses **structured databases** to store and retrieve information.

Useful for retrieving structured data like user profiles - transactions - logs ...

In [ ]:

```
from langchain.sql_database import SQLDatabase
from langchain.chains import SQLDatabaseChain
from langchain.llms import OpenAI

# Connect to an SQL database
db = SQLDatabase.from_uri("sqlite:///my_database.db")

# Create a query chain
chain = SQLDatabaseChain(llm=OpenAI(), database=db)

# Ask a question that requires structured retrieval
response = chain.run("How many users signed up in January?")
print(response)
```

Example -- Chat Application with Streamlit

In [1]:

```
# Import necessary dependencies
import os  # Used to set environment variables
import streamlit as st  # Streamlit for creating the web app
from langchain.llms import OpenAI  # OpenAI's language model integration
from langchain.prompts import PromptTemplate  # For structuring prompts
from langchain.chains import LLMChain, SequentialChain  # Chains for managing model interactions
from langchain.memory import ConversationBufferMemory  # Memory to store conversation history
from langchain.utilities import WikipediaAPIWrapper  # Wikipedia API for fetching relevant data

# Set OpenAI API Key (ensure to keep this secret in production)
os.environ['OPENAI_API_KEY'] = 'sk-proj-xY1X5DjaugUzWHxj54qsZCRY-TR41QlcEoWPo5wuY8sXqf3fsXliWK00q78EzBNujpQGwZLGncT3


# Create the Streamlit app interface
st.title('🦜🔗 YouTube GPT Creator')  # App title
prompt = st.text_input('Plug in your prompt here')  # User input field


# Define Prompt Templates
# Template for generating a YouTube video title based on a given topic
title_template = PromptTemplate(
    input_variables=['topic'],  # The expected input variable
    template='write me a youtube video title about {topic}')


# Template for generating a YouTube video script using Wikipedia research
script_template = PromptTemplate(
    input_variables=['title', 'wikipedia_research'],  # Uses title and research data
    template='write me a youtube video script based on this title TITLE: {title} while leveraging this wikipedia rese


# Define memory buffers to store conversation history
title_memory = ConversationBufferMemory(input_key='topic', memory_key='chat_history')  # Memory for title generation
script_memory = ConversationBufferMemory(input_key='title', memory_key='chat_history')  # Memory for script generati


# Initialize OpenAI language model with a specific temperature setting
llm = OpenAI(temperature=0.9)  # Higher temperature makes responses more creative
```

```python
# Create LLM chains for title and script generation
title_chain = LLMChain(llm=llm, prompt=title_template, verbose=True, output_key='title', memory=title_memory)
script_chain = LLMChain(llm=llm, prompt=script_template, verbose=True, output_key='script', memory=script_memory)


# Initialize Wikipedia API wrapper to fetch related content
wiki = WikipediaAPIWrapper()


# Check if the user has entered a prompt
if prompt:
    # Generate a video title using the input prompt
    title = title_chain.run(prompt)

    # Fetch related information from Wikipedia
    wiki_research = wiki.run(prompt)

    # Generate a video script using the title and Wikipedia research
    script = script_chain.run(title=title, wikipedia_research=wiki_research)

    # Display the generated title
    st.write(title)

    # Display the generated script
    st.write(script)

    # Expandable sections for viewing conversation history
    with st.expander('Title History'):
        st.info(title_memory.buffer)  # Show past generated titles

    with st.expander('Script History'):
        st.info(script_memory.buffer)  # Show past generated scripts

    with st.expander('Wikipedia Research'):
        st.info(wiki_research)  # Show Wikipedia research used for the script
```

```
2025-03-18 12:48:56.367 WARNING streamlit.runtime.scriptrunner_utils.script_run_context: Thread 'MainThread': missing
ScriptRunContext! This warning can be ignored when running in bare mode.
2025-03-18 12:48:57.147
  Warning: to view this Streamlit app on a browser, run it with the following
  command:

    streamlit run C:\Users\MTechno\AppData\Roaming\Python\Python311\site-packages\ipykernel_launcher.py [ARGUMENTS]
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Session state does not function when running a script without `streamlit run`
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
2025-03-18 12:48:57.151 Thread 'MainThread': missing ScriptRunContext! This warning can be ignored when running in ba
re mode.
C:\Users\MTechno\AppData\Local\Temp\ipykernel_20140\582130594.py:33: LangChainDeprecationWarning: Please see the migr
ation guide at: https://python.langchain.com/docs/versions/migrating_memory/
  title_memory = ConversationBufferMemory(input_key='topic', memory_key='chat_history')  # Memory for title generatio
n
C:\Users\MTechno\AppData\Local\Temp\ipykernel_20140\582130594.py:38: LangChainDeprecationWarning: The class `OpenAI`
was deprecated in LangChain 0.0.10 and will be removed in 1.0. An updated version of the class exists in the :class:`
~langchain-openai package and should be used instead. To use it run `pip install -U :class:`~langchain-openai` and im
port as `from :class:`~langchain_openai import OpenAI``.
  llm = OpenAI(temperature=0.9)  # Higher temperature makes responses more creative
C:\Users\MTechno\AppData\Local\Temp\ipykernel_20140\582130594.py:42: LangChainDeprecationWarning: The class `LLMChain
` was deprecated in LangChain 0.1.17 and will be removed in 1.0. Use :meth:`~RunnableSequence, e.g., `prompt | llm``
instead.
  title_chain = LLMChain(llm=llm, prompt=title_template, verbose=True, output_key='title', memory=title_memory)
```