WEEK 2 / DAY 6

LangChain Deep Dive

Moving towards Modular Architecture.

By,

Shikha Tyagi

Founder - AI JAMIC (AI Research and Consulting)

Education: IIT Delhi (M.Tech.)

Where we stand

Week 1 Recap - Building Foundation

LLMs

We know how to call Gemini/GPT-4o via API.

Tools

We built functions (currency, weather) for the model to use.

RAG

We built memory systems with Vector DBs.

Starting today's session we will move towards using frameworks for building Agentic AI Application

Why it is helpful -

- Manually formatting strings for prompts.
- Manually parsing JSON outputs.
- Manually managing chat history lists.
- If we switch from OpenAI to Anthropic, we rewrite everything.

LangChain

LangChain is a framework for developing applications powered by language models.

Composability

Building blocks (Chains) that stack together.

Model Agnostic

Switch LLMs by changing 1 line of code.

Built-in Utilities

Document loaders, splitters, and parsers ready to go.

Langchain: Open Source Frameworks

https://github.com/langchain-ai/langchain/tree/master/libs

Quiz: what is first thing we pass to LLM?

Respond in Chat

Prompt Templates

Hardcoding strings is bad practice. Use **PromptTemplates**.

```
from langchain_core.prompts import ChatPromptTemplate


prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful tutor."),
    ("human", "Explain {topic} in simple words for a {age}-year-old.")
])

messages = prompt.format_messages(topic="gravity", age=10)
print(message)
```

{topic} is static or dynamic?

Prompt Templates

Every prompt can have two parts: static & dynamic

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful tutor."),
    ("human", "Explain {topic} in simple words for a {age}-year-old.")
])
```

{topic} and {age} are dynamic. Which will be replaced based on user input

How to work with rapidly evolving LLM Space

LangChain standardizes the interface.

- ChatOpenAI
- ChatAnthropic
- ChatGoogleGenerativeAI
- HuggingFaceHub

They all use the same method: .invoke()

https://docs.langchain.com/oss/python/integrations/providers/overview

How to work with rapidly evolving LLM Space

Sample Code

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("user", "{input}")
])

model = ChatOpenAI()

# This is an LCEL chain
chain = prompt | model

# Now you can invoke it
response = chain.invoke({"input": "What is LangChain Expression Language?"})
```

Output Parsers

LLMs output **AIMessage** objects (metadata + content).

Parsers extract just what you need.

- StrOutputParser: Just the text string.
- JsonOutputParser: Enforces and extracts JSON.

Output Parsers

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser

# 1. Define the model, prompt, and parser
model = ChatOpenAI()
template = PromptTemplate(template="Write a 5 line summary on the following text: {text}",
input_variables=['text'])
parser = StrOutputParser()

# 2. Create the chain
chain = template | model | parser

# 3. Invoke the chain to get a clean string result
result = chain.invoke({'text': 'Your input text here...'})
print(result)
```

LCEL: The Core Syntax

LangChain Expression Language (LCEL) uses the Unix Pipe | operator.

Input | Prompt | Model | Output Parser

```
chain = prompt | model | parser
```

Quick Check

Let's test your understanding of basics of Langchain.

What is the main purpose of LangChain Expression Language (LCEL)?

a) To store embeddings
b) To build chains using a simple, declarative syntax
c) To create databases
d) To visualize model outputs

In a ChatPromptTemplate, template variables are used to:

a) Store model weights
b) Dynamically fill values at runtime
c) Control GPU usage
d) Convert messages to embeddings

In an LCEL chain, the model component is responsible for:

a) Parsing the final output
b) Rendering HTML
c) Generating responses based on the formatted prompt
d) Storing the chat history

What does StrOutputParser() do in a LangChain pipeline?

a) Converts model output into images
b) Parses model output into plain text
c) Compresses the output
d) Runs the model faster

Which LCEL expression correctly connects a prompt, a model, and a parser?

a) prompt + parser + model
b) model | prompt | parser
c) prompt | model | StrOutputParser()
d) StrOutputParser() | prompt | model

Adding Memory

Conversations have history. Lanchain has RunnableWithMessageHistory to add memory to conversations.

Hands On

Chain vs Agent

Definition:
A **Chain** is a sequence of steps that takes input → processes it → produces output. Each step can be a prompt, a model call, or a function.

Characteristics:

- Predefined workflow.
- Deterministic: always follows the same sequence.
- Usually one main task (e.g., summarization, question answering).
- Can be **LCEL pipeline** like: `PromptTemplate | Model | Parser`.

Chain vs Agent

Definition:
An **Agent** is a higher-level system that can **decide dynamically which actions or tools to use** based on the input.
It can call **multiple chains, tools, APIs**, or even run Python code, and decide the order **at runtime**.

Characteristics:

- Flexible and intelligent: chooses actions dynamically.
- Can use **tools** (search engines, calculators, APIs, custom functions).
- Good for **complex tasks** or multi-step problem solving.
- Uses **LLM reasoning + tools** to decide next step.

Chain vs Agent

Feature	Chain	Agent
Workflow	Static, predefined sequence	Dynamic, chooses actions at runtime
Flexibility	Low	High
Tools usage	Usually none	Can use multiple tools/APIs
Use case	Summarization, QA, simple tasks	Multi-step reasoning, RAG, tool usage
Complexity	Simple	More complex
LCEL compatibility	Fully compatible	Compatible but may require tools setup

Let us build Agents with LangChain

Let us build Agents with LangChain

In Langchain

“`create_agent`” provides a production-ready agent implementation.

```
from langchain.agents import create_agent

agent = create_agent(
    "gpt-5",
    tools=tools
)
```

The Anatomy of a Tool

A "Tool" in LangChain is more than just a function. It acts as an API endpoint for the LLM.

- **Name:** How the model calls it (e.g., search).
- **Description:** The "Prompt" for the tool. Tells the LLM *when* to use it.
- **Args Schema:** Type validation (Pydantic) to ensure inputs are correct.

The @tool Decorator

In Day 3, we wrote JSON schemas manually. LangChain generates them for us using the @tool decorator and Python type hints.

Types of Agents

1. Tool Calling Agent

Best for: GPT-4o, Claude 3.5, Gemini.

Uses the model's native "Function Calling" API.

Reliable, structured, and fewer parsing errors.

2. ReAct Agent

Best for: Older models or open-source (Llama 2).

Uses prompt engineering ("Thought: ..., Action: ...") to simulate reasoning. More verbose.

When should you use an Agent instead of a Chain?

A. When you want the process to be faster.

B. When the sequence of steps is not known in advance.

C. When you are using a Vector Database.

D. When you want to save money.

Hands-On Lab

Rebuilding the "Currency and Weather" Agent

Don't Reinvent the Wheel

LangChain ships with 100+ pre-built integrations called **Toolkits**.

Google Search

Connects to SerpApi/Google to get real-time search results.

Wikipedia

Automatically searches and summarizes Wikipedia pages.

Python REPL

An agent that writes and executes its own Python code to solve math.

Summary: Day 6

- **LCEL:** The syntax for piping components together.
- **Chains:** For predictable, linear sequences.
- **Agents:** For dynamic, tool-using reasoning loops.
- **Abstraction:** LangChain hides the messy "glue code" so you can focus on logic.

Coming Up: Day 7

Building a **Production-Grade RAG** App with LangChain.

We will combine vector stores, retrievers, and history into a "Chat with PDF" tool.