LangChain Pipeline with OpenAI

🧭 Objective

We are building a **basic LangChain pipeline** using OpenAI's LLM (`gpt-4o`) to:

- Send prompts
- Process outputs
- Optionally use tracing with LangSmith
- (Optionally) serve the app via LangServe

1️⃣ Load Environment Variables

🔍 What?

Read API keys and config values from a `.env` file.

💡 Why?

This keeps sensitive keys out of your code.

1. You write your API keys in a `.env` file

You create this `.env` file in the **same folder** as your Python script.

This file looks like:

```
OPENAI_API_KEY=your-real-openai-key
LANGCHAIN_API_KEY=your-langchain-key
LANGCHAIN_PROJECT=my-cool-project-name
```

```
import os
from dotenv import load_dotenv
load_dotenv()
```

```
# Set OpenAI and LangChain environment variables
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")
```

`load_dotenv()` : This line **reads the `.env` file** and makes the keys available in your script.

`os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")` **:** This tells Python:
"
Look inside the `.env` file, get the `OPENAI_API_KEY`, and make it available for use."

(You do the same thing for LangChain (a tool to help with AI apps)

2️⃣ Load the OpenAI Chat Model

Purpose: Create an instance of OpenAI's latest model (`gpt-4o`).

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")  # Uses GPT-4 Turbo (fast & powerful)
print(llm)  # Prints model config (confirms setup)
```

Output:

```
client=<OpenAI Completions object>
model_name='gpt-4o'
openai_api_key=SecretStr('**********')
```

3️⃣ Send a Direct Prompt

Use `.invoke()` to send a single question and get a response.

```
result = llm.invoke("What is generative AI?")
```

```
print(result)
```

Key Notes:

- `invoke()` sends a single prompt to the model.
- Response includes **metadata** (token usage, model details).

4️⃣ Create a Prompt Template

🔍 What?

Use `ChatPromptTemplate` to define reusable prompt formats.

💡 Why?

Makes it easy to dynamically insert user input and apply structure to conversations.

This code sets up a **chat prompt template** for an AI model — basically a way to tell the model:

1. **What kind of role it should play** (like an expert).
2. **What the user will ask** (using a placeholder).

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI Engineer. Provide me answers based on th
e questions"),
    ("user", "{input}")
])
```

Key Notes:

- `system` sets the AI's behavior (e.g., "expert AI Engineer").

- `{input}` is a placeholder for dynamic user queries.

`ChatPromptTemplate.from_messages([...])`

- This creates a **conversation structure** for the AI.
- `("system", "...")`
    - Tells the AI **what kind of role it should play**.
    - Like: *"Pretend you're an expert AI Engineer."*
- `("user", "{input}")`
    - This is a **placeholder**. Whatever you pass in later as `"input"` will go here.
    - For example, if you ask *"What is a neural network?"*, that will replace `{input}`.
    - It tells LangChain:

      👉 "I'll give you the actual question later when I run this prompt."

- The **system** message sets the AI's tone and purpose.
- The **user** message is the question.

💡
- If you skip the `system` role, the AI may behave in a **generic** way.
    - Using `system` helps **customize behavior**: teacher, coder, chef, therapist, etc.

5️⃣ Build a Chain (Prompt → Model → Output Parser)

Purpose: Combine components into a **pipeline** for structured AI responses.

A Chain Without Parser (Raw AI Response):

```
chain = prompt | llm  # Pipe: Prompt → Model
response = chain.invoke({"input": "What is LangSmith?"})
print(response)
```

Output:

- A **detailed response** about LangSmith (logging, testing, monitoring).

It's using the **pipe operator (`|`)** to **connect two things**:

- A **prompt** (your message template)
- An **LLM** (a language model like OpenAI's GPT)

So it's saying:

> "Take this prompt, fill it in with input, then send it to the language model (LLM) to get a response."

6️⃣ Add an Output Parser

🔍 What?

Convert the LLM response into a plain string (if it's an object).

💡 Why?

Standardizes the output format for further use in apps or logs.

```
from langchain_core.output_parsers import StrOutputParser

output_parser = StrOutputParser()

# Full chain: prompt → LLM → parse response
```

```
chain = prompt | llm | output_parser

response = chain.invoke({"input": "Can you tell me about Langsmith?"})
print(response)
```

Output:

```
"LangSmith is a tool for debugging and monitoring LLM applications..."
```

🎯 Key Takeaways

✅ **LangChain simplifies AI workflows** (prompts, models, parsing).

✅ **LangSmith tracks interactions** (useful for debugging).

✅ **Chaining components** makes AI apps modular & reusable.