Assignment:

Generate an end to end Langgraph Application for automating the workflow of the architecture of orchestrator and synthesizer(Think of different usecase) using open source LLM models and make sure to debug with the help of Langsmith(Attach the zip file and explain your fundamental approach an dproblem statement )

To build an end-to-end **LangGraph application** for automating the workflow of an **Orchestrator and Synthesizer** architecture using open-source LLM models, we can think of a use case where we automate the orchestration and synthesis of various tasks or processes within a complex system. For this example, let’s focus on a **data pipeline orchestration** system, where the orchestrator is responsible for coordinating various tasks such as data extraction, transformation, and loading (ETL), and the synthesizer is responsible for processing and generating a final report or summary.

**Problem Statement**

The problem we are addressing is automating a **data pipeline** in a modular way, where various tasks (e.g., data extraction from an API, transformation, and reporting) are orchestrated. The **Orchestrator** ensures that tasks are executed in the right sequence, while the **Synthesizer** combines the outputs of those tasks and provides a final, useful result, such as a summary or a report.

* **Orchestrator**: Coordinates different tasks in the pipeline (e.g., ETL).
* **Synthesizer**: Combines the results from the orchestrated tasks and synthesizes a final report.

We’ll utilize open-source LLMs like **GPT-Neo** or **GPT-J** to process and generate content, and **Langsmith** for debugging the orchestration process.

**Core Use Case**

We can think of a use case like:

1. **Data Extraction**: Collect data from different APIs (e.g., news data, stock data).
2. **Data Transformation**: Clean and process the data.
3. **Report Generation**: Combine the processed data and generate a summary or report.
4. **Debugging**: Use Langsmith to monitor and debug the pipeline, ensuring tasks are executed in sequence without errors.

**LangGraph Architecture**

The application consists of two main components:

1. **Orchestrator**: Handles task sequencing and flow control.
2. **Synthesizer**: Combines the outputs of orchestrated tasks into a final report.

**Step-by-Step Breakdown**

1. **Orchestrator Node**: This node will handle the orchestration of tasks, such as calling the data extraction, transformation, and report generation nodes in the correct sequence.
2. **Data Extraction Node**: This will simulate data extraction from a source (such as an API or database).
3. **Data Transformation Node**: This node will simulate cleaning and transforming the extracted data.
4. **Report Generation Node**: This node will synthesize the data into a report.
5. **Langsmith Debugging**: This will monitor the entire workflow, logging the inputs and outputs for debugging purposes.

**Code Implementation**

We will define and implement these nodes in LangGraph, utilizing **Langsmith** for debugging.

**1. Data Extraction Node**

from langgraph import Node

class DataExtractionNode(Node):

def run(self):

# Simulate data extraction from an external source (API, Database)

data = {"news": "Breaking news: LangGraph automates workflows!", "stock\_price": "Stock XYZ: $150"}

print("Data extracted successfully.")

return data

**2. Data Transformation Node**

class DataTransformationNode(Node):

def run(self, inputs):

# Simulate data transformation (e.g., cleaning and formatting)

data = inputs["data"]

transformed\_data = {

"news": f"Processed News: {data['news']}",

"stock": f"Processed Stock Price: {data['stock\_price']}"

}

print("Data transformed successfully.")

return transformed\_data

**3. Report Generation Node**

from transformers import GPT2LMHeadModel, GPT2Tokenizer

class ReportGenerationNode(Node):

def run(self, inputs):

transformed\_data = inputs["transformed\_data"]

# Load the LLM model for text generation

model\_name = "EleutherAI/gpt-neo-2.7B"

model = GPT2LMHeadModel.from\_pretrained(model\_name)

tokenizer = GPT2Tokenizer.from\_pretrained(model\_name)

# Prepare the prompt for the LLM to synthesize a report

prompt = f"Generate a summary report based on the following information:\n\nNews: {transformed\_data['news']}\nStock Price: {transformed\_data['stock']}"

inputs = tokenizer(prompt, return\_tensors="pt")

# Generate report

output = model.generate(\*\*inputs, max\_length=500, num\_return\_sequences=1)

generated\_report = tokenizer.decode(output[0], skip\_special\_tokens=True)

print("Report generated successfully.")

return {"report": generated\_report}

**4. Orchestrator Node**

class OrchestratorNode(Node):

def run(self):

# Orchestrating the execution of tasks in sequence

print("Starting the orchestration process...")

# Step 1: Data Extraction

data\_extraction\_node = DataExtractionNode()

data = data\_extraction\_node.run()

# Step 2: Data Transformation

data\_transformation\_node = DataTransformationNode()

transformed\_data = data\_transformation\_node.run({"data": data})

# Step 3: Report Generation

report\_generation\_node = ReportGenerationNode()

report = report\_generation\_node.run({"transformed\_data": transformed\_data})

print("Orchestration process completed.")

return report

**5. Final Output and Debugging with Langsmith**

from langgraph import Workflow

from langsmith import Langsmith

# Create LangGraph Workflow for Orchestrator and Synthesizer

class DataPipelineWorkflow(Workflow):

def define(self):

orchestrator\_node = OrchestratorNode(name="Orchestrator")

orchestrator\_node.connect() # There's no further node here; it's the final output.

# Initialize Langsmith debugger

debugger = Langsmith(api\_key="your-api-key")

# Initialize and run the workflow with debugging

workflow = DataPipelineWorkflow()

debugger.monitor(orchestrator\_node) # Monitor the orchestrator node for debugging

workflow.run(debugger=debugger)

**Detailed Workflow Breakdown**

1. **Data Extraction**:
   * The **DataExtractionNode** simulates fetching data from external sources (APIs, databases, etc.).
   * This step involves extracting news and stock price information.
2. **Data Transformation**:
   * The **DataTransformationNode** processes and transforms the raw data.
   * This could involve cleaning, filtering, or reformatting the data before it's used for the report.
3. **Report Generation**:
   * The **ReportGenerationNode** utilizes the GPT-Neo model (or any other open-source LLM) to synthesize the transformed data into a coherent report or summary.
4. **Orchestrator**:
   * The **OrchestratorNode** ensures that these tasks are executed in the correct sequence.
   * The orchestration manages the data flow, ensuring that each task depends on the successful completion of the previous one.
5. **Langsmith Debugging**:
   * **Langsmith** is used to monitor the inputs and outputs of each task and ensure that the flow of data and logic is correct.
   * If any task fails or produces unexpected outputs, Langsmith logs will help identify and debug the issue.

**Final Report**

The generated report is stored and can be accessed as output, which is a summary or result of the orchestrated tasks.

**Packaging and Zipping the Application**

1. **Folder Structure**:
2. /data\_pipeline\_workflow
3. - data\_extraction\_node.py
4. - data\_transformation\_node.py
5. - report\_generation\_node.py
6. - orchestrator\_node.py
7. - main.py
8. - requirements.txt
9. **requirements.txt**: To ensure all dependencies are captured:
10. langgraph==<version>
11. langsmith==<version>
12. transformers==<version>
13. openai==<version>
14. **Zip the Folder**: Use any file compression tool to zip the folder containing all the Python files and the requirements.txt.

**Conclusion**

This **LangGraph application** automates the orchestration and synthesis of a data pipeline with clear task separation:

* **Orchestrator** coordinates task execution.
* **Synthesizer** (Report Generation) creates a final output based on the orchestrated tasks.

The use of **Langsmith** ensures debugging and efficient monitoring of the application to guarantee smooth execution and provide insights into potential issues. This approach can be easily adapted to other workflows beyond the data pipeline use case.