**Assignment**:

Generate an end to end Langgraph Application for automating the workflow of Coding Peer Reviews using open source LLM models and make sure to debug with the help of Langsmith(Attach the zip file and explain your fundamental approach )

To build an end-to-end LangGraph application for automating the workflow of **Coding Peer Reviews** using open-source LLM models (e.g., GPT-J, GPT-Neo, etc.) and debugging with Langsmith, we need to go through the following steps:

1. **Define the Workflow**: We'll automate the coding peer review process. The basic workflow will consist of:
   * **Input Collection**: Collect code from the user and specify the areas of review (e.g., code quality, readability, performance).
   * **Code Analysis**: Use an LLM to analyze the code based on specific criteria like readability, efficiency, and potential issues.
   * **Review Report Generation**: Generate a peer review report based on the analysis.
   * **Output**: Provide the final review feedback.
   * **Debugging**: Use Langsmith to debug the workflow and ensure proper functioning.
2. **Set up and Install Dependencies**:
   * We'll need the **LangGraph** framework to orchestrate the workflow.
   * Use **open-source LLM models** like GPT-J or GPT-Neo for analyzing the code.
   * **Langsmith** will be used for debugging.

Install necessary libraries:

pip install langgraph langsmith openai transformers

1. **Create LangGraph Application**: Below is the breakdown of each step in the workflow.

**1. Input Collection (Code Submission):**

This node collects code from the user and specifies the areas to review, such as readability, performance, and security.

from langgraph import Node

class CodeInputCollectionNode(Node):

def run(self):

# Simulating user input

code = input("Enter your code for review:\n")

review\_areas = input("Enter areas of review (e.g., readability, performance, security):\n")

return {

"code": code,

"review\_areas": review\_areas

}

**2. Code Analysis (Peer Review with LLM):**

This node uses an open-source LLM (e.g., GPT-Neo or GPT-J) to analyze the code based on the specified review areas.

from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Code Analysis Node (Peer Review using LLM)

class CodeAnalysisNode(Node):

def run(self, inputs):

code = inputs["code"]

review\_areas = inputs["review\_areas"]

# Load pre-trained model and tokenizer

model\_name = "EleutherAI/gpt-neo-2.7B" # Open-source LLM

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

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

# Prepare the prompt

prompt = f"Review the following code based on the areas: {review\_areas}.\n\nCode:\n{code}\n\nProvide feedback on readability, performance, and security."

# Tokenize the prompt

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

# Generate review

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

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

return generated\_review

**3. Review Report Generation:**

This node generates a formatted peer review report based on the code analysis feedback.

# Review Report Generation Node

class ReviewReportGenerationNode(Node):

def run(self, inputs):

review = inputs["review"]

# Formatting the peer review report

review\_report = f"# Peer Review Report\n\n### Code Review Feedback\n\n{review}\n\n### Conclusion\nEnsure you address the feedback provided for better code quality."

return review\_report

**4. Output (Final Peer Review Report):**

This node outputs the peer review report to a file or terminal.

# Output Node (Final Report)

class OutputNode(Node):

def run(self, inputs):

with open("peer\_review\_report.md", "w") as f:

f.write(inputs["review\_report"])

print("Peer review report saved as peer\_review\_report.md")

**5. LangGraph Workflow Setup:**

Now, we combine everything into a LangGraph workflow, defining how the nodes are connected.

from langgraph import Workflow

class CodePeerReviewWorkflow(Workflow):

def define(self):

input\_node = CodeInputCollectionNode(name="Code Input")

analysis\_node = CodeAnalysisNode(name="Code Analysis")

report\_node = ReviewReportGenerationNode(name="Review Report Generation")

output\_node = OutputNode(name="Output")

# Define the node relationships

input\_node.connect(analysis\_node)

analysis\_node.connect(report\_node)

report\_node.connect(output\_node)

# Initialize and run the workflow

workflow = CodePeerReviewWorkflow()

workflow.run()

**6. Debugging with Langsmith:**

Langsmith is used to debug and monitor the workflow execution. You can track each node and monitor the inputs and outputs to identify any issues in the process.

from langsmith import Langsmith

# Initialize Langsmith for debugging

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

# Monitor each node during the execution

debugger.monitor(input\_node)

debugger.monitor(analysis\_node)

debugger.monitor(report\_node)

debugger.monitor(output\_node)

# Run the workflow with debugging

workflow.run(debugger=debugger)

**Final Steps:**

1. **Save all the code**: Save each class (CodeInputCollectionNode, CodeAnalysisNode, ReviewReportGenerationNode, OutputNode) in separate Python files.
2. **Create requirements.txt**: To capture all dependencies:
3. langgraph==<version>
4. langsmith==<version>
5. transformers==<version>
6. openai==<version>
7. **Zip the folder**: Create a directory containing all the Python files, requirements.txt, and any other necessary files (like a configuration or environment setup), then zip the folder.

**Debugging and Validation with Langsmith:**

* **Langsmith** will allow you to track inputs and outputs at each step of the workflow, ensuring that the peer review process is executed smoothly.
* If an error occurs (e.g., an unexpected code format or incomplete feedback), Langsmith will provide error logs and help identify where the issue lies in the workflow.

**ZIP File:**

I cannot directly provide a ZIP file here, but you can create a ZIP of your project structure as follows:

1. **Folder structure**:
2. /coding\_peer\_review
3. - code\_input\_collection\_node.py
4. - code\_analysis\_node.py
5. - review\_report\_generation\_node.py
6. - output\_node.py
7. - main.py
8. - requirements.txt
9. **Zip the entire folder**: Use any zip tool to create a compressed file for deployment.

**Summary of the Workflow:**

1. **Code Input Collection**: Users provide the code and specify areas to review (readability, performance, etc.).
2. **Code Analysis**: An open-source LLM like GPT-Neo analyzes the code and provides feedback based on the input review areas.
3. **Review Report Generation**: The feedback is formatted into a structured peer review report.
4. **Output**: The peer review report is saved to a file (e.g., markdown).
5. **Debugging**: Langsmith is used to monitor and debug the entire process, providing insights into each node's performance.

This approach automates the peer review process and allows for easy debugging and refinement through Langsmith.