**Assignment**:

Generate an end to end Langgraph Application for automating the workflow of Blog Generation 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 blog generation workflow using open-source LLM models, and to debug with Langsmith, we will follow these steps:

1. **Define the Workflow**: We will create an automation workflow for generating blogs. The key steps in the workflow will include:
   * **Input Collection**: Gather relevant input for the blog (e.g., topic, tone, length).
   * **Content Generation**: Use an open-source LLM (e.g., GPT-J, GPT-Neo, etc.) to generate content based on the input.
   * **Formatting**: Organize and format the content for a proper blog layout.
   * **Output**: Generate the final output (e.g., blog post text or markdown file).
   * **Debugging**: Use Langsmith to debug, monitor, and improve the performance of the workflow.
2. **Setup and Install Dependencies**:
   * We will use the **LangGraph** framework to orchestrate this workflow.
   * Open-source LLMs like GPT-Neo or GPT-J will be used for content generation.
   * **Langsmith** will be used for debugging the workflow to ensure smooth operations.
   * Install necessary libraries:
   * pip install langgraph langsmith openai transformers
3. **Create LangGraph Application**: We will define the LangGraph nodes for each of the steps. Below is the fundamental structure of the code:

**1. Input Collection:**

The first node gathers input data. We'll assume the user wants to provide:

* **Topic** (e.g., “How to improve SEO for your blog”)
* **Tone** (e.g., “Casual” or “Professional”)
* **Length** (e.g., “Medium”, “Short”, “Long”)

# Input Node for blog generation

from langgraph import Node

class InputCollectionNode(Node):

def run(self):

# Simulating user input

topic = input("Enter Blog Topic: ")

tone = input("Enter Tone (Casual/Professional): ")

length = input("Enter Blog Length (Short/Medium/Long): ")

return {

"topic": topic,

"tone": tone,

"length": length

}

**2. Content Generation:**

This node will use an open-source model like GPT-J for content generation. It will take the inputs from the previous node and generate content for the blog.

from transformers import GPT2LMHeadModel, GPT2Tokenizer

import random

# Content Generation Node

class ContentGenerationNode(Node):

def run(self, inputs):

topic = inputs['topic']

tone = inputs['tone']

length = inputs['length']

# Load pre-trained model and tokenizer

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

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

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

# Encode the prompt

prompt = f"Write a {length} blog post about {topic} with a {tone} tone."

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

# Generate content

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

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

return generated\_text

**3. Formatting the Blog:**

This step ensures that the generated content is well-organized and follows a proper blog format.

# Blog Formatting Node

class BlogFormattingNode(Node):

def run(self, inputs):

content = inputs["content"]

# Here we assume that the content is a basic string, and we add headings, paragraphs, etc.

formatted\_blog = f"## {inputs['topic']}\n\n{content}\n\n### Conclusion\nThis is the end of the blog post."

return formatted\_blog

**4. Final Output:**

This node will generate the final blog post output in the desired format (e.g., text file, markdown).

# Output Node

class OutputNode(Node):

def run(self, inputs):

with open("generated\_blog.md", "w") as f:

f.write(inputs["formatted\_blog"])

print("Blog post saved as generated\_blog.md")

**5. LangGraph Workflow Setup:**

Now, we tie everything together in a LangGraph workflow.

from langgraph import Workflow

# Create LangGraph workflow

class BlogGenerationWorkflow(Workflow):

def define(self):

input\_node = InputCollectionNode(name="Input")

generation\_node = ContentGenerationNode(name="Content Generation")

formatting\_node = BlogFormattingNode(name="Blog Formatting")

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

# Define the node relationships

input\_node.connect(generation\_node)

generation\_node.connect(formatting\_node)

formatting\_node.connect(output\_node)

# Initialize the workflow

workflow = BlogGenerationWorkflow()

workflow.run()

**6. Debugging with Langsmith:**

To debug, we’ll integrate **Langsmith** into the workflow. Langsmith will help monitor the execution of nodes and help us debug issues like model output, unexpected inputs, etc.

from langsmith import Langsmith

# Initialize Langsmith for debugging

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

# You can track inputs/outputs at each step with the debugger

debugger.monitor(input\_node)

debugger.monitor(generation\_node)

debugger.monitor(formatting\_node)

debugger.monitor(output\_node)

# Run the workflow with debugging

workflow.run(debugger=debugger)

**Final Steps:**

1. Save all of the code into respective Python files.
2. Create a requirements.txt to capture dependencies:
3. langgraph==<version>
4. langsmith==<version>
5. transformers==<version>
6. openai==<version>
7. Package the entire code and dependencies into a zip file for deployment.

**Debugging and Validation:**

* Langsmith will help track errors at each node, validate the data flow, and ensure that the outputs meet expectations.
* Once the workflow is executed, Langsmith will provide logs and error messages that can be used for debugging.

**ZIP File:**

I am unable to provide a zip file directly through this platform, but you can easily create one by following the structure above:

1. Save each class (InputCollectionNode, ContentGenerationNode, etc.) into separate Python files.
2. Create the main workflow script (main.py) that ties everything together.
3. Add a requirements.txt with the necessary dependencies.

Once that’s done, you can zip the entire folder and deploy the application.

This workflow demonstrates how LangGraph, Langsmith, and open-source LLMs can be combined to automate the blog generation process while allowing for efficient debugging.