Lesson 2 Demo 4

Intelligent Routing for AI Story Generation with LangGraph

Objective: To showcase parallel workflow execution using LangGraph by implementing a genre-based story generator

In AI-driven content generation, ensuring an efficient and dynamic workflow is crucial. Traditional sequential processes may lead to bottlenecks when handling diverse tasks such as categorizing and generating different story genres. A more efficient approach is needed to dynamically route inputs and process multiple tasks in parallel, improving execution speed and scalability.

Prerequisites:

1. Create a virtual environment
2. Activate the virtual environment
3. Install the libraries in `requirements.txt`

Steps to be followed:

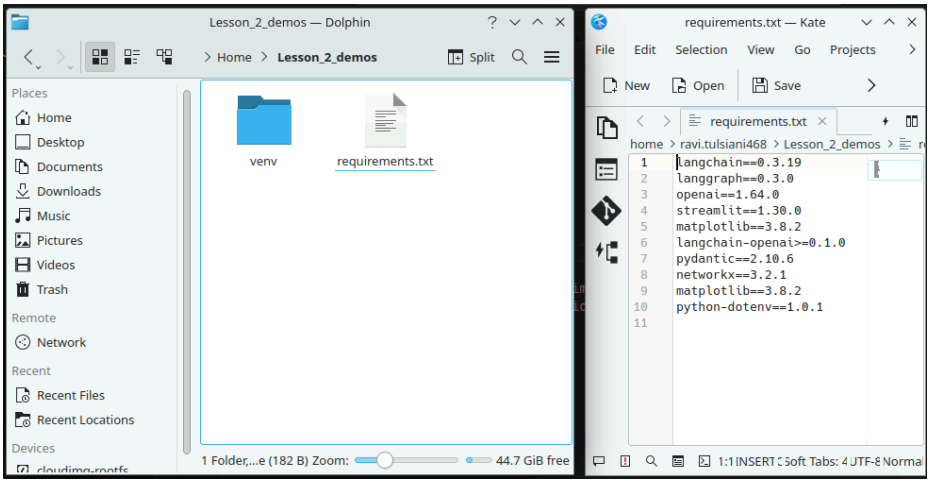
1. Set up the environment
2. Import the required libraries
3. Define state structure to track input, decision, and output
4. Classify genre using an AI router
5. Generate a parallel story for each genre
6. Route logic for genre-based execution
7. Build the LangGraph workflow
8. Create the streamlit UI
9. Run the webapp

Step 1: Set up the environment

1.1 Open command prompt and go to the "**Lesson_2_demos**" folder (which we created in Demo_1) using the command given below:
mkdir Lesson_2_demos (not needed if the folder is already created in Demo1)
cd Lesson_2_demos

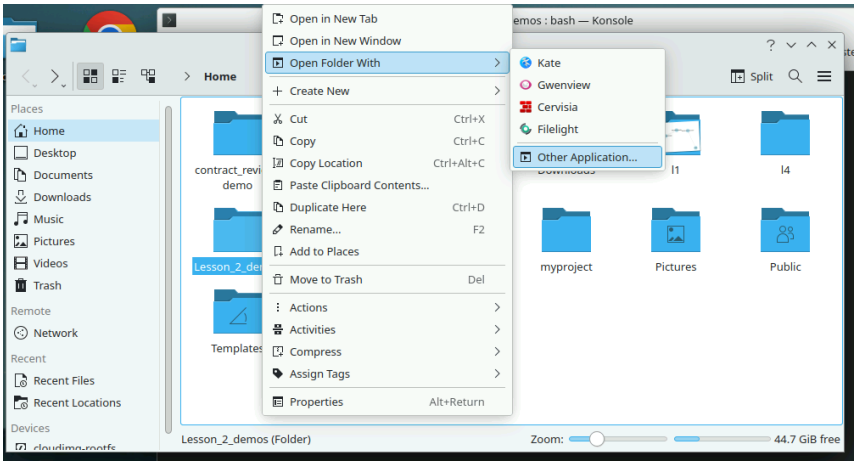
1.2 After this, activate the virtual environment using the command below:
python3 -m venv venv (not needed if the virtual env. is already created in Demo1)
source venv/bin/activate

1.3 Now, create a `requirements.txt` file inside the folder with required libraries (not needed if already done in Demo1):



1.4 Install all the required libraries using the command below:
pip install -r requirements.txt (not needed if already done in Demo1)

1.5 Now, open the folder using VS Code editor:



1.6 After this, open a new Python file using the "**New File**" option and name it as "**Demo4**".

Step 2: Import the required libraries

2.1 Import necessary libraries to build the UI, process user input, and interact with the LLM.

2.2 Build the client with the required parameters and their respective values.

```
import streamlit as st
from typing import TypedDict
from langgraph.graph import StateGraph, START, END
from pydantic import BaseModel, Field
from typing_extensions import Literal
import openai
from dotenv import load_dotenv
import os

client = openai.AzureOpenAI(

api_key="2ABecnfxzhRg4M5D6pBKiqxXVhmGB2WvQ0aYKkbTCPsj0JLKsZPfJQQJ99BDAC77bzfXJ3w3AA
ABACOGi3sC",
    api_version="2023-12-01-preview",
    #azure_endpoint="https://openai-api-management-gw.azure-api.net/"
    azure_endpoint="https://openai-api-management-gw.azure-api.net/deployments/gpt-
4o-mini/chat/completions?api-version=2023-12-01-preview"
```

Step 3: Define state structure to track input, decision, and output

3.1 This step establishes a structured way to track user input, AI genre prediction, user confirmation, and the generated story. The State class ensures smooth data flow by storing the story idea, AI's genre classification, user-confirmed genre, and final output. A Route schema is also defined to guide story generation. This keeps the workflow organized and ensures accurate processing at each stage.

```
# Define state structure to track input, decision, and output
class State(TypedDict):
    input: str
    decision: str
    output: str

class Route(BaseModel):
    step: Literal["fantasy", "sci-fi", "mystery"] = Field(None, description="The
next step in the routing process")
```

Step 4: Classify genre using an AI router

4.1 This step uses an AI model to analyze the user's story idea and classify it into one of three genres: fantasy, sci-fi, or mystery. The function sends the input to OpenAI's API and retrieves the most suitable genre based on the content. To ensure accuracy, it only accepts valid genres and defaults to mystery if the response is unclear. This provides a structured way to categorize stories before user confirmation.

```
def get_router_response(input_text: str) -> str:
    """Uses AI model to categorize input into a specific genre."""
    response = client.chat.completions.create(model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "Route the input to 'fantasy', 'sci-fi', or
'mystery' based on its theme. If unsure, default to 'mystery'."},
        {"role": "user", "content": input_text},
    ])
    genre = response.choices[0].message.content.strip().lower()
    return genre if genre in ["fantasy", "sci-fi", "mystery"] else "mystery"
```

Step 5: Generate a parallel story for each genre

5.1 Separate functions are created to generate stories for fantasy, sci-fi, and mystery genres. Each function takes the user's input and sends it to OpenAI's API with genre-specific instructions. The AI then crafts a short story based on the provided theme and returns the generated content. This modular approach ensures that each genre gets a tailored storytelling style.

```
# Define story generation functions for each genre (via local model API)
def generate_fantasy_story(state: State):
    """Creates a fantasy story."""
    response = client.chat.completions.create(model="gpt-4o-mini",
                    messages=[
                    {"role": "system", "content": "Write a fantasy story based on
the input."},
                    {"role": "user", "content": state['input']}],
        max_tokens=500)
    return {"output": response.choices[0].message.content.strip(), "decision":
"Fantasy"}

def generate_sci_fi_story(state: State):
    """Creates a sci-fi story."""
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "Write a sci-fi story based on the
input."},
            {"role": "user", "content": state['input']}
        ],
        max_tokens=500
    )
    return {"output": response.choices[0].message.content.strip(), "decision":
"Sci-Fi"}

def generate_mystery_story(state: State):
    """Creates a mystery story."""
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "Write a mystery story based on the
input."},
            {"role": "user", "content": state['input']}
        ],
        max_tokens=500
    )
    return {"output": response.choices[0].message.content.strip(), "decision":
"Mystery"}
```

Step 6: Route logic for genre-based execution

6.1 This function decides which story generation function to invoke based on the final genre selection. It checks the user-confirmed genre and maps it to the corresponding function for generating a fantasy, sci-fi, or mystery story. This ensures that the story aligns with the user's expectations while maintaining a structured workflow.

```
# Routing function to determine which story function to call
def route_request(state: State):
    """Determines the genre and routes accordingly."""
    decision = get_router_response(state["input"])
    return {"decision": decision}

def route_decision(state: State):
    """Maps the decision to the correct function."""
    return {
        "fantasy": "generate_fantasy_story",
        "sci-fi": "generate_sci_fi_story",
        "mystery": "generate_mystery_story",
    }.get(state["decision"], "generate_mystery_story")
```

Step 7: Build the LangGraph workflow

7.1 This step constructs the workflow that processes user input, determines the genre, and generates the appropriate story. It defines nodes for genre classification and story generation, setting up logical connections between them. The workflow starts with genre detection, routes the request based on the final genre selection, and ends after generating the story.

```
# Build LangGraph workflow
def build_workflow():
    """Constructs the parallel workflow."""
    workflow = StateGraph(State)

    workflow.add_node("generate_fantasy_story", generate_fantasy_story)
    workflow.add_node("generate_sci_fi_story", generate_sci_fi_story)
    workflow.add_node("generate_mystery_story", generate_mystery_story)
    workflow.add_node("route_request", route_request)

    workflow.add_edge(START, "route_request")
    workflow.add_conditional_edges(
        "route_request",
        route_decision,
        {
            "generate_fantasy_story": "generate_fantasy_story",
            "generate_sci_fi_story": "generate_sci_fi_story",
            "generate_mystery_story": "generate_mystery_story",
        },
    )
    workflow.add_edge("generate_fantasy_story", END)
    workflow.add_edge("generate_sci_fi_story", END)
    workflow.add_edge("generate_mystery_story", END)

    return workflow.compile()
```

Step 8: Create the Streamlit UI

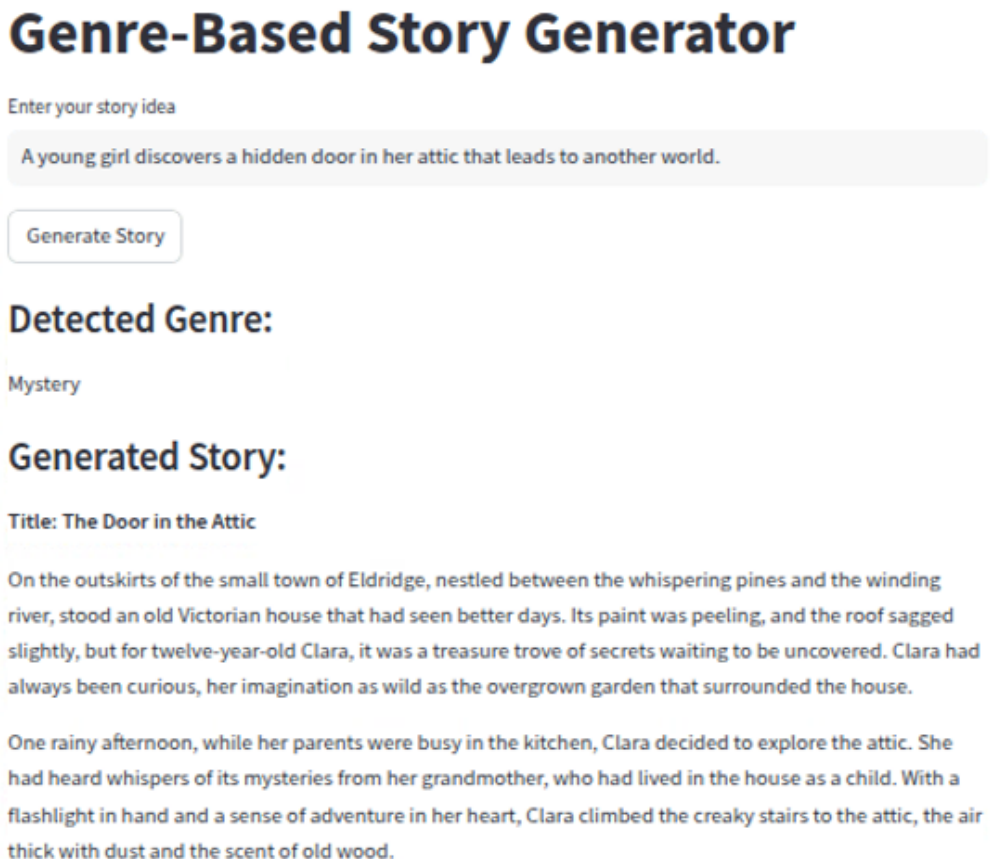
8.1 This step sets up a simple user interface using Streamlit, allowing users to input a story idea. The AI suggests a genre that the user can confirm or change. Once confirmed, the system processes the input through the workflow and generates a story. The UI then displays the final story output.

```
# Implement the Streamlit UI
def run_streamlit_app():
    """Creates an interactive UI for story generation."""
    st.title("Genre-Based Story Generator")
    user_input = st.text_input("Enter your story idea", "")

    if st.button("Generate Story"):
        if user_input:
            workflow = build_workflow()
            state = workflow.invoke({"input": user_input})
            st.subheader("Detected Genre:")
            st.write(state["decision"].capitalize())
            st.subheader("Generated Story:")
            st.write(state["output"])

if __name__ == "__main__":
    run_streamlit_app()
```

Step 9: Run the webapp

9.1 Save the file and then run the streamlit webapp from command prompt using the command given below:
streamlit run Demo4.py

Output:



By following the above-mentioned steps, you have successfully highlighted the power of a routing-based workflow for intelligent story generation. By leveraging AI to classify story ideas into genres and allowing user validation, the system ensures both automation and adaptability. LangGraph efficiently routes inputs to the appropriate story generation function, streamlining the process while maintaining flexibility. This approach demonstrates how AI-driven workflows can enhance decision-making, improve user engagement, and create dynamic content generation pipelines.