Lesson 2 Demo 3

Parallel AI Content Generation with LangGraph

Objective: To showcase the parallel execution of tasks using LangGraph in a workflow that generates marketing content

Given a topic (for example, a smartphone), the system will simultaneously generate an advertisement, a product review, and a catchy tagline. These outputs will then be combined into a final result. The parallelization aspect ensures efficient execution by leveraging LangGraph's ability to execute independent tasks concurrently.

Prerequisites:

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

Steps to be followed:

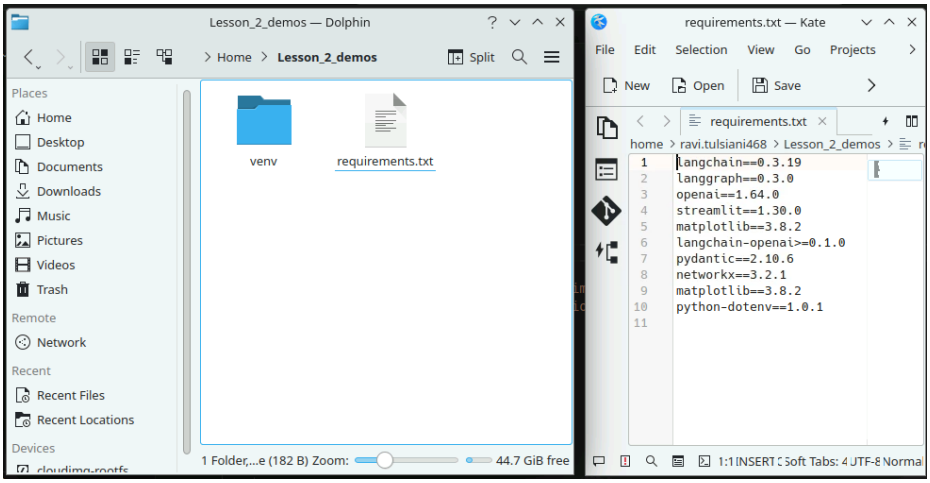
1. Set up the environment
2. Import the required libraries
3. Define the state structure
4. Generate an advertisement
5. Generate a product review
6. Generate a catchy tagline
7. Combine all generated outputs
8. Build the LangGraph workflow
9. Create the Streamlit UI
10. Run the webapp

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

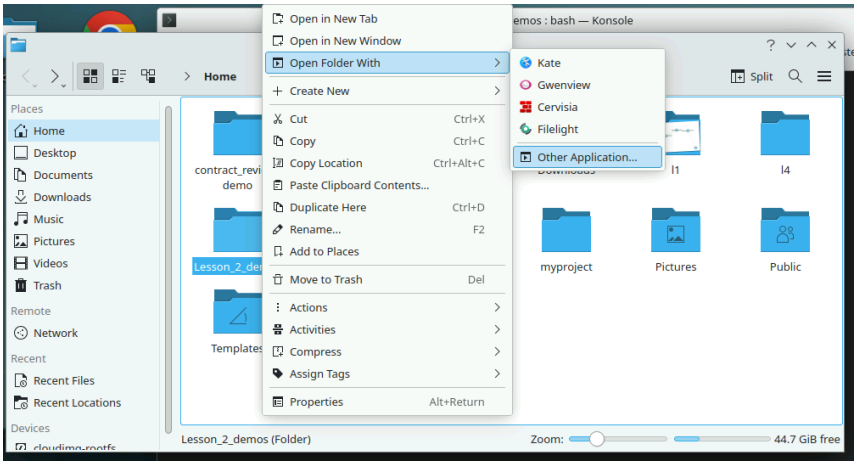
1.2 After this, activate the virtual environment using the command below:
python3 -m venv venv (not needed if the virtual env. is already created in Demo1)
source venv/bin/activate

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 "**Demo3**".

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.

```
# Set Up the Environment
import streamlit as st
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from dotenv import load_dotenv
import os
import openai


client = openai.AzureOpenAI(

api_key="2ABecnfxzhRg4M5D6pBKiqxXVhmGB2WvQ0aYKkbTCPsj0JLKsZPfJQQJ99BDAC77bzfXJ3w3AAA
BACOGi3sC",
    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 the state structure

3.1 In this step, the state structure is defined using the `TypedDict` class from `typing_extensions`. The state serves as a shared data container that holds key variables as the workflow progresses.

3.2 It ensures that all steps in the LangGraph workflow have access to the necessary information.

```
# Define the state structure
class State(TypedDict):
    topic: str
    advertisement: str
    review: str
    tagline: str
    combined_output: str
```

Step 4: Generate an advertisement

4.1 In this step, the system calls the OpenAI API to generate a creative and engaging advertisement based on the provided topic. A system prompt frames the AI as an expert in writing catchy advertisements, ensuring that the output is compelling and attention-grabbing.

4.2 This step helps market the product effectively by providing persuasive and engaging content.

```
# Generate an advertisement
def generate_advertisement(state: State):
    """Calls OpenAI API to generate an advertisement related to the given topic."""
    msg = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a creative AI that writes catchy
advertisements."},
            {"role": "user", "content": f"Write a catchy advertisement for a
product related to {state['topic']}."}
        ],
        max_tokens=200
    )
    advertisement = msg.choices[0].message.content.strip()
    return {"advertisement": advertisement}
```

Step 5: Generate a product review

5.1 The system calls the OpenAI API to generate a detailed product review that includes pros and cons. A system prompt frames the assistant as a knowledgeable reviewer, guiding it to produce informative content.

5.2 This step is crucial for providing potential buyers in-depth insights into the product, helping them make informed purchasing decisions.

```
def generate_review(state: State):
    """Calls OpenAI API to generate a detailed product review for the given
topic."""
    msg = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a helpful assistant that writes
detailed product reviews."},
            {"role": "user", "content": f"Write a product review for a product
related to {state['topic']}. Include pros and cons."}
        ],
        max_tokens=300
    )
    review = msg.choices[0].message.content.strip()
    return {"review": review}
```

Step 6: Generate a catchy tagline

6.1 This step involves calling the OpenAI API to generate a short and memorable tagline for the product. The system prompt frames the AI as a branding expert, ensuring that the tagline is engaging and effective. A strong tagline enhances product recall and serves as a key marketing element.

```
# Generate a catchy tagline
def generate_tagline(state: State):
    """Calls OpenAI API to generate a catchy tagline for the given topic."""
    msg = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a creative AI that generates
catchy taglines."},
            {"role": "user", "content": f"Create a short, catchy tagline for a
product related to {state['topic']}."}
        ],
        max_tokens=50
    )
    tagline = msg.choices[0].message.content.strip()
    return {"tagline": tagline}
```

Step 7: Combine all generated outputs

7.1 Once the advertisement, review, and tagline have been generated, this step merges them into a structured creative output. The result presents a cohesive marketing package that effectively communicates the product's value.

7.2 By combining different perspectives – persuasive advertising, detailed reviews, and a catchy tagline, the output becomes well-rounded and impactful.

```
# Combine all outputs
def combine_outputs(state: State):
    """Combines the advertisement, review, and tagline into a single structured
output."""
    combined = f"Creative Output for {state['topic']}:\n\n"
    combined += f"ADVERTISEMENT:\n{state['advertisement']}\n\n"
    combined += f"REVIEW:\n{state['review']}\n\n"
    combined += f"TAGLINE:\n{state['tagline']}"
    return {"combined_output": combined}
```

Step 8: Build the LangGraph workflow

8.1 This step involves constructing a parallel execution graph using LangGraph. Independent nodes are created for advertisement, review, and tagline generation, ensuring they run simultaneously.

8.2 The workflow is designed so that all three tasks start in parallel, and their results feed into the final combination step, maximizing efficiency.

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

    # Adding independent nodes (tasks that can run in parallel)
    parallel_builder.add_node("generate_advertisement", generate_advertisement)
    parallel_builder.add_node("generate_review", generate_review)
    parallel_builder.add_node("generate_tagline", generate_tagline)
    parallel_builder.add_node("combine_outputs", combine_outputs)

    # Setting up parallel execution
    parallel_builder.add_edge(START, "generate_advertisement")
    parallel_builder.add_edge(START, "generate_review")
    parallel_builder.add_edge(START, "generate_tagline")
    parallel_builder.add_edge("generate_advertisement", "combine_outputs")
    parallel_builder.add_edge("generate_review", "combine_outputs")
    parallel_builder.add_edge("generate_tagline", "combine_outputs")
    parallel_builder.add_edge("combine_outputs", END)

    # Compile the workflow
    parallel_workflow = parallel_builder.compile()
    return parallel_workflow
```

Step 9: Create the Streamlit UI

9.1 In the final step, a Streamlit-based UI is used to accept user input for the topic and trigger the parallel workflow upon clicking a button. The workflow executes the steps concurrently, and once all tasks are completed, the structured creative output is displayed in the UI.

9.2 This step provides an interactive way for users to experience the power of parallel execution in AI-driven workflows.

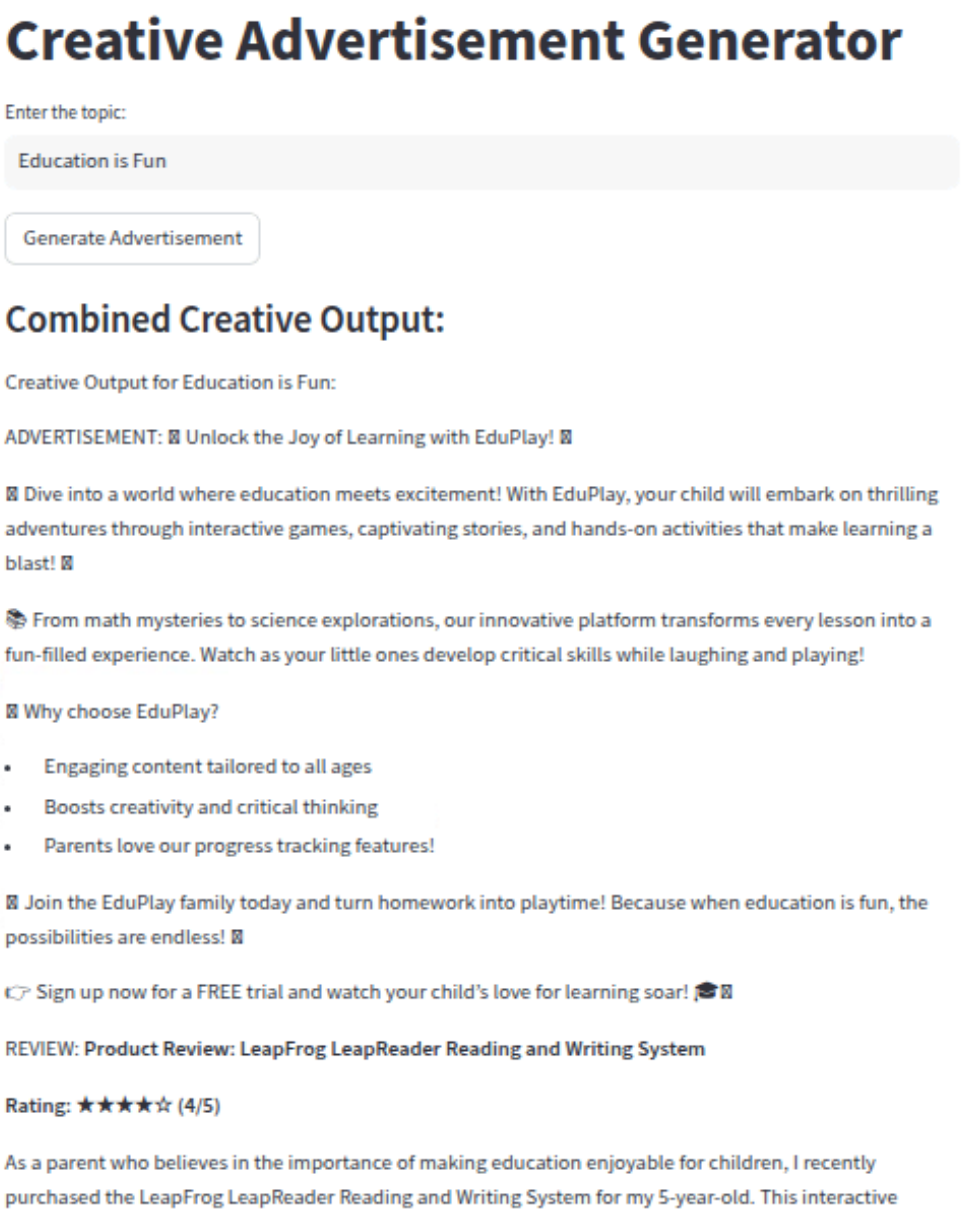
```
# Streamlit UI
def run_streamlit_app():
    """Handles Streamlit UI interactions and workflow execution."""
    st.title("Creative Advertisement Generator")
    topic = st.text_input("Enter the topic:")
    if st.button("Generate Advertisement"):
        parallel_workflow = build_workflow()
        state = parallel_workflow.invoke({"topic": topic})
        st.subheader("Combined Creative Output:")
        st.write(state["combined_output"])

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

Step 11: Run the webapp

11.1 Save the file and then run the streamlit webapp from command prompt using the command given below:
streamlit run Demo3.py

Output:



By following the above-mentioned steps, you have successfully built a parallel content generation workflow using LangGraph. This demo showcases how multiple tasks, such as creating an advertisement, a product review, and a catchy tagline, can run concurrently to produce comprehensive and cohesive marketing content efficiently. The integration of these outputs via a Streamlit UI not only streamlines the process but also highlights the power of parallel execution in enhancing productivity and content quality.