DATA-236 Sec 21&71

HOMEWORK 2

Instructions:

- Please provide the code solution for each question along with its intended output. Ensure that the code and corresponding output screenshots are placed together, one below the other.

- Submission should be in PDF Format.

- Please name your submission file as {last_name}_HW2.pdf

Part 1. HTML & CSS (4 points) - Artist Liberty.
Part 2. FastAPI(8 points)

1. Write the code to add a new book. The user should be able to enter the Book Title and Author Name. Once the user submits the required data, the book should be added, and the user should be redirected to the home view showing the updated list of books. (2 points)

2. Write the code to update the book with ID 1 to title:" Harry Potter", Author Name: "J.K Rowling". After submitting the data, redirect to the home view and show the updated data in the list of books. (2 points)

3. Write the code to delete the book with the highest ID. After submitting the data, redirect to the home view and show the updated data in the list of books. (2 points)

4. Write the code to add search functionality. The user should be able to search for a book by name/title. When the user enters a name and searches, the list should update to show only the matching books. (2 points)

Part 3: Stateful Agent Graph

Objective: The goal of this assignment is to refactor your previous sequential agent script into a more robust, stateful graph using the langgraph library. This will implement the **supervisor pattern** from the lecture, creating a system that can dynamically route tasks and even loop back for self-correction.

Step 1: Understanding the Core Concepts

In your last assignment, you created a simple "waterfall" process: the Planner ran, then the Reviewer, then it ended. This is rigid. A graph-based approach allows for more complex flows, like loops and conditional paths.

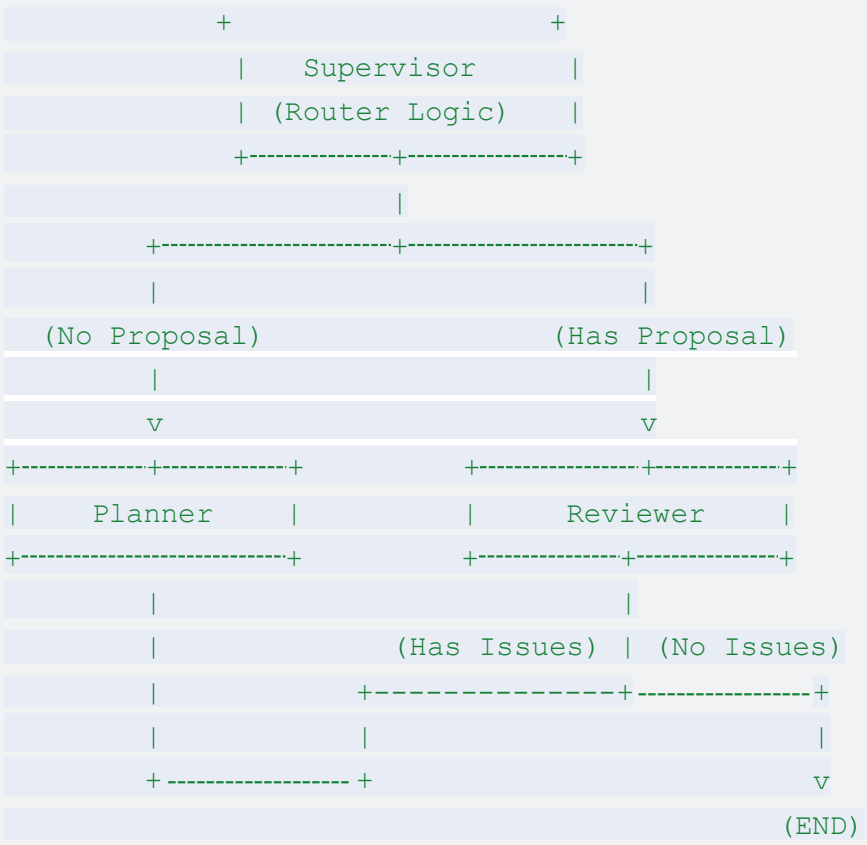
We will use langgraph to build this. Key concepts are:

- **AgentState:** A shared dictionary that acts as the "memory" for all agents. Every agent can read from and write to this central state.

- **Nodes:** These are the workers. Each of our agents (`Planner`, `Reviewer`) will become a node. A node is just a Python function that takes the current `AgentState` and returns a dictionary with updates.
- **Edges:** These are the arrows that connect the nodes, defining the flow of control. We'll use **conditional edges** to let a supervisor decide which path to take.

Here is the flow we're building:

None



Step 2: Setting up the State

First, define the shared `AgentState` using Python's `TypedDict`. This class will represent the memory of our system. It needs to hold the initial inputs, the outputs from each agent, and a turn counter to prevent infinite loops.

```
None

from typing import TypedDict, Dict, Any

class AgentState(TypedDict):
    title: str
    content: str
    email: str
    strict: bool
    task: str
    llm: Any
    planner_proposal: Dict[str, Any]
    reviewer_feedback: Dict[str, Any]
    turn_count: int
```

Step 3: Creating the Agent Nodes

Next, convert your `Planner` and `Reviewer` logic into standalone functions. Each function must:

1. Accept `state: AgentState` as its only argument.
2. Perform its task (e.g., call the LLM).
3. Return a dictionary containing only the keys of the `AgentState` it wants to update.

Example for `planner_node`:

```
None

def planner_node(state: AgentState) -> Dict[str, Any]:
    print("---NODE: Planner ---")
    # ... (your existing planner logic) ...
    proposal = ... # The JSON output from the LLM
    return {"planner_proposal": proposal}
```

Step 4: Building the Supervisor (The Router)

The supervisor is the "brain" of the operation. It doesn't do the work; it directs it. We split its logic into two parts:

1. **A State-Updating Node (`supervisor_node`):** This node's only job is to modify the state, like incrementing the turn counter.

2. **A Routing Function (`router_logic`):** This function *reads* the state and decides where to go next by returning a string (e.g., `"planner"`, `"reviewer"`, or `END`).

Step 5: Assembling the Graph

Now, wire everything together in your `main` function.

Step 6: Running and Testing

Finally, invoke your graph with the initial state and use the `.stream()` method to see the output from each step.

To test your correction loop, temporarily modify your `reviewer_node` to always return an issue, and watch the graph route the task back to the `planner`.

Helpful Resources

- **LangGraph Multi-Agent Collaboration:** LangGraph Docs
- **Python's `TypedDict`:** Python Docs