To build the AI agent flow for the Espyr Hackathon’s DSA Preparation Coach, we need a system where multiple agents (Mentor Agent, Code Agent, Evaluation Agent, Persona Manager, and Orchestrator Agent) work simultaneously to guide users through solving DSA problems. The goal is to ensure seamless collaboration, real-time adaptability, and efficient state transitions while maintaining a generic, scalable architecture. Below is a detailed approach to designing the agent flow, focusing on simultaneous operation, as per the challenge document and your emphasis on AI-related work. Since you’ve requested no code for now, I’ll provide a conceptual framework, workflow design, and practical steps for implementation, tailored for your team’s backend-first approach.

**Core Principles for Simultaneous Agent Flow**

1. **Parallel Processing with Coordination**: Agents should operate concurrently where possible (e.g., Mentor Agent guiding the user while Persona Manager assesses skill level), but their actions must be synchronized to avoid conflicts (e.g., Code Agent only activates after Mentor approval).
2. **Event-Driven Architecture**: Use events to trigger agent actions, enabling asynchronous communication and reducing bottlenecks.
3. **State Management**: A finite state machine (FSM) or context-based system tracks the user’s progress, ensuring agents know when to act or wait.
4. **Dynamic Adaptation**: Agents adjust their behavior in real-time based on user input and skill level, requiring simultaneous monitoring and processing.
5. **Generic Design**: Agents should handle any DSA problem without hardcoded logic, relying on modular prompts and reusable workflows.

**Agent Roles and Simultaneous Responsibilities**

Based on the challenge document, here’s how each agent contributes to the flow and works concurrently:

1. **Mentor Agent**:
   * **Role**: Guides the user through problem-solving by asking for their approach, providing hints, and adapting tone.
   * **Simultaneous Tasks**:
     + Actively engages with the user via chat, processing their approach in real-time.
     + Listens for user inputs (e.g., text describing their approach) and generates responses using an AI model (e.g., GPT-4).
     + Communicates with the Persona Manager to adjust tone based on user skill level.
     + Signals the Orchestrator Agent when the user is ready to code (e.g., TOOL\_CALL: open editor).
2. **Code Agent**:
   * **Role**: Analyzes user code, runs test cases, detects bugs, and suggests optimizations.
   * **Simultaneous Tasks**:
     + Remains idle until triggered by the Orchestrator Agent but can preprocess test cases for the current problem in the background.
     + Once activated, it parses user code, runs tests, and generates feedback concurrently with the Mentor Agent’s monitoring of user progress.
     + Communicates with the Evaluation Agent to log performance metrics (e.g., bugs found, test case results).
3. **Evaluation Agent**:
   * **Role**: Provides a holistic session recap, assessing thought process clarity, hint usage, and code optimality.
   * **Simultaneous Tasks**:
     + Collects data in real-time (e.g., number of hints used, retries) while the Mentor and Code Agents interact with the user.
     + Prepares a performance summary in the background, ready to deliver when the user completes the problem.
     + Adjusts feedback based on input from the Persona Manager about the user’s skill level.
4. **Persona Manager**:
   * **Role**: Adjusts agent tone and behavior based on user skill level (beginner, intermediate, advanced).
   * **Simultaneous Tasks**:
     + Continuously analyzes user inputs (e.g., approach descriptions, code quality) to infer skill level.
     + Updates all agents with tone and complexity adjustments in real-time (e.g., motivational for beginners, challenging for advanced users).
     + Runs in parallel with other agents, ensuring their responses align with the user’s proficiency.
5. **Orchestrator Agent**:
   * **Role**: Manages transitions between agents and user states (approach → code → evaluation).
   * **Simultaneous Tasks**:
     + Listens for events from all agents (e.g., Mentor Agent’s TOOL\_CALL, Code Agent’s test results).
     + Coordinates agent activation/deactivation to prevent overlap (e.g., ensures Code Agent waits for Mentor approval).
     + Maintains the FSM to track the user’s current state and trigger transitions.

**Agent Flow Design for Simultaneous Operation**

To enable agents to work simultaneously, we’ll use an **event-driven, state-managed workflow** with **LangGraph** (as recommended in the document) for orchestration. Here’s a step-by-step breakdown of the flow, ensuring concurrency and coordination:

**1. System Initialization**

* **Trigger**: User selects a DSA problem from the web interface.
* **Agent Actions**:
  + **Orchestrator Agent**: Initializes the FSM, setting the state to “Problem Discussion.” Loads the problem’s metadata (e.g., problem statement, sample test cases).
  + **Persona Manager**: Begins with a default skill level (e.g., beginner) but starts analyzing user inputs to refine this assessment.
  + **Mentor Agent**: Displays the problem statement and prompts the user: “What’s your initial approach to solve this?”
  + **Code Agent**: Preprocesses the problem’s test cases in the background (e.g., generates edge cases like large inputs or duplicates).
  + **Evaluation Agent**: Initializes a session log to track metrics (e.g., hints used, retries).

**2. Problem Discussion Phase**

* **State**: Problem Discussion (FSM State 1).
* **Agent Actions**:
  + **Mentor Agent** (Primary): Engages in a conversational loop:
    - Processes user input (e.g., “I’ll try every pair to find the sum”).
    - Generates a response based on skill level (e.g., for beginners: “Good start! What if the input is very large? Hint: Can a HashMap help?”).
    - Increments hint counter and logs it with the Evaluation Agent.
  + **Persona Manager** (Concurrent): Analyzes the user’s approach:
    - If the user suggests brute-force, infers beginner level and signals Mentor Agent to use motivational tone.
    - If the user mentions data structures like HashMap, updates to intermediate level.
    - Sends real-time updates to all agents to adjust tone/complexity.
  + **Orchestrator Agent** (Concurrent): Listens for Mentor Agent’s signal to transition (e.g., user confirms approach or requests to code).
  + **Code Agent** (Background): Remains idle but prepares test case logic for the problem (e.g., edge cases for negative numbers).
  + **Evaluation Agent** (Background): Logs user interactions (e.g., approach clarity, hint requests) for later recap.

**3. Coding Phase**

* **State**: Coding (FSM State 2).
* **Trigger**: Mentor Agent issues TOOL\_CALL: open editor after validating the user’s approach.
* **Agent Actions**:
  + **Orchestrator Agent**: Transitions the FSM to “Coding” and activates the Code Agent.
  + **Code Agent** (Primary): Takes over:
    - Opens the Monaco Editor (via UI trigger).
    - Parses user-submitted code in real-time as they type or submit.
    - Runs predefined and dynamically generated test cases (e.g., edge cases like duplicates or large inputs).
    - Provides feedback (e.g., “Your code fails for negative numbers. Consider checking edge cases.”).
    - If logic is incorrect, signals Orchestrator to revert to Mentor Agent for further guidance.
  + **Mentor Agent** (Concurrent): Monitors user progress passively, ready to provide hints if the Code Agent flags issues.
  + **Persona Manager** (Concurrent): Analyzes code quality (e.g., time complexity, variable naming) to refine skill level assessment.
  + **Evaluation Agent** (Background): Logs code submissions, test case results, and bugs detected.
  + **Orchestrator Agent** (Concurrent): Coordinates feedback loops (e.g., Code Agent → Mentor Agent if user struggles).

**4. Evaluation Phase**

* **State**: Evaluation (FSM State 3).
* **Trigger**: Code Agent confirms the user’s solution passes all test cases or user opts to submit.
* **Agent Actions**:
  + **Evaluation Agent** (Primary): Generates a session recap:
    - Summarizes thought process clarity (based on Mentor Agent logs).
    - Reports hint usage and retries (from session log).
    - Assesses code optimality (e.g., time/space complexity from Code Agent).
    - Delivers feedback tailored to skill level (e.g., “Great use of two-pointers! Can you optimize for space?” for advanced users).
  + **Mentor Agent** (Concurrent): Provides motivational wrap-up or suggests next steps (e.g., “Try a harder problem next!”).
  + **Persona Manager** (Concurrent): Finalizes skill level assessment for the session and updates user profile for future interactions.
  + **Code Agent** (Background): Stores final code and test results for reference.
  + **Orchestrator Agent**: Closes the session or transitions to a new problem if the user continues.

**Technical Implementation for Simultaneous Operation**

To enable agents to work concurrently while maintaining coordination, we’ll use the following tools and strategies, as suggested in the document:

1. **LangGraph for Orchestration**:
   * **Why**: LangGraph supports multi-agent workflows with asynchronous task execution, ideal for simultaneous agent operation.
   * **How**:
     + Define nodes for each agent (Mentor, Code, Evaluation, Persona, Orchestrator).
     + Use edges to represent state transitions (e.g., Problem Discussion → Coding).
     + Implement event-driven triggers (e.g., TOOL\_CALL: open editor) to activate agents.
     + Enable parallel execution by allowing non-blocking nodes (e.g., Persona Manager runs concurrently with Mentor Agent).
   * **Example Workflow**:
     + Mentor Agent node processes user input and emits a “hint\_provided” event.
     + Persona Manager node listens for user input events and updates skill level asynchronously.
     + Orchestrator node routes events to appropriate agents based on FSM state.
2. **Finite State Machine (FSM)**:
   * **Why**: Ensures agents only act when appropriate, preventing overlap (e.g., Code Agent doesn’t run tests during problem discussion).
   * **How**:
     + Define states: Problem Discussion, Coding, Evaluation.
     + Use a state manager (e.g., XState or custom context in React) to track the current state.
     + Agents subscribeto state changes via events (e.g., Mentor Agent triggers “ready\_to\_code” event).
     + Store state transitions and session data in a lightweight database (e.g., in-memory store or SQLite for prototype).
3. **AI Model Integration**:
   * **Why**: Powers agent intelligence for natural language processing, code analysis, and feedback generation.
   * **How**:
     + Use **GPT-4 API** for Mentor Agent’s conversational logic and hint generation.
     + Leverage **Hugging Face Transformers** for lightweight tasks like code parsing or test case generation if API credits are limited.
     + Design modular prompts for each agent:
       - Mentor Agent: “Given a user’s approach {input} and skill level {level}, provide a hint in a {tone} tone.”
       - Code Agent: “Analyze code {code} for problem {problem}, check for bugs, and suggest optimizations.”
       - Evaluation Agent: “Summarize session with metrics {hints\_used, retries, complexity} for skill level {level}.”
     + Run API calls asynchronously to minimize latency (e.g., Mentor Agent processes input while Code Agent prepares test cases).
4. **Event Bus for Communication**:
   * **Why**: Enables agents to share data and triggers without tight coupling.
   * **How**:
     + Implement a simple event bus (e.g., using Node.js EventEmitter or RxJS for React).
     + Define events like user\_input\_received, hint\_provided, code\_submitted, skill\_level\_updated.
     + Agents subscribe to relevant events (e.g., Persona Manager listens for user\_input\_received to assess skill).
5. **Background Processing**:
   * **Why**: Allows agents like Code Agent and Evaluation Agent to perform tasks (e.g., test case generation, session logging) without blocking user interaction.
   * **How**:
     + Use async/await or worker threads for non-blocking tasks (e.g., Code Agent generates test cases while Mentor Agent chats).
     + Cache problem metadata (e.g., test cases, expected complexity) to reduce redundant computation.

**Sample Simultaneous Workflow (Example Scenario)**

To illustrate how agents work together concurrently, let’s walk through a beginner user solving a “Two Sum” problem:

1. **User Starts Problem**:
   * **Orchestrator Agent**: Sets FSM to “Problem Discussion” and loads problem metadata.
   * **Mentor Agent**: Asks, “What’s your approach to find two numbers that sum to a target?”
   * **Persona Manager**: Assumes beginner level, monitors user input.
   * **Code Agent**: Prepares test cases (e.g., [2, 7, 11, 15], target=9) in the background.
   * **Evaluation Agent**: Initializes session log.
2. **User Responds**:
   * **User Input**: “I’ll check every pair of numbers.”
   * **Mentor Agent**: Processes input via GPT-4, responds: “Good start! What’s the time complexity? Hint: Can a HashMap make it faster?”
   * **Persona Manager**: Detects brute-force approach, confirms beginner level, signals Mentor to use motivational tone.
   * **Evaluation Agent**: Logs hint usage (+1 hint).
   * **Orchestrator Agent**: Listens for Mentor’s next action.
3. **User Refines Approach**:
   * **User Input**: “Maybe use a HashMap to store numbers and check for complements.”
   * **Mentor Agent**: Responds: “Great idea! What’s the complexity now?” Triggers TOOL\_CALL: open editor.
   * **Persona Manager**: Updates skill to intermediate based on HashMap usage.
   * **Orchestrator Agent**: Transitions FSM to “Coding,” activates Code Agent.
   * **Code Agent**: Opens Monaco Editor, waits for user code.
   * **Evaluation Agent**: Logs approach improvement.
4. **User Submits Code**:
   * **Code Agent**: Parses code, runs test cases (e.g., fails on duplicate numbers), responds: “Your code fails for [3, 3], target=6. Check duplicates.”
   * **Mentor Agent**: Concurrently offers a hint: “How can you handle duplicate numbers in your HashMap?”
   * **Persona Manager**: Monitors code quality, confirms intermediate level.
   * **Evaluation Agent**: Logs test failure and retry.
   * **Orchestrator Agent**: Reverts to Mentor Agent if user needs more guidance.
5. **User Submits Final Code**:
   * **Code Agent**: Confirms all test cases pass, calculates complexity (O(n)).
   * **Evaluation Agent**: Generates recap: “Clear approach after 2 hints, passed all tests, optimal O(n) solution.”
   * **Mentor Agent**: Congratulates: “Awesome job! Ready for a harder problem?”
   * **Persona Manager**: Updates user profile to intermediate for future sessions.
   * **Orchestrator Agent**: Transitions to “Evaluation” state, closes session or loads next problem.

**Key Considerations for Simultaneous Operation**

1. **Concurrency Management**:
   * Use asynchronous APIs (e.g., GPT-4 async calls) to prevent blocking.
   * Implement rate-limiting to manage API credit usage.
   * Use a queue for agent tasks to prioritize critical actions (e.g., Mentor Agent’s response over Evaluation Agent’s logging).
2. **Conflict Avoidance**:
   * Ensure the Orchestrator Agent enforces state-based rules (e.g., Code Agent only activates in “Coding” state).
   * Use locks or semaphores for shared resources (e.g., session log updates).
3. **Scalability**:
   * Design agents with modular prompts to handle any DSA problem.
   * Cache common test cases and problem metadata to reduce computation.
4. **Error Handling**:
   * Handle API failures gracefully (e.g., fallback to local Hugging Face models).
   * Log errors for debugging without disrupting user flow.
5. **Team Task Division**:
   * **Agent Logic**: Assign team members to design prompts for each agent (e.g., Mentor Agent prompts, Code Agent test logic).
   * **Orchestration**: Dedicate 1–2 members to implement LangGraph and FSM.
   * **AI Integration**: Assign members to integrate GPT-4 and test Hugging Face models.
   * **Testing**: All members test concurrent flows for different skill levels.

**Tools and Setup for Simultaneous Flow**

* **LangGraph**: For agent orchestration and event-driven workflows.
* **GPT-4 API**: For conversational logic and code analysis (use async calls).
* **Hugging Face Transformers**: For local fallback or lightweight tasks.
* **Node.js/React**: For backend event bus and minimal frontend.
* **Monaco Editor**: For code input, triggered by Orchestrator events.
* **SQLite/In-Memory Store**: For session logging and state persistence.

**Next Steps for Implementation**

1. **Define Agent Prompts (Day 1–2)**:
   * Create modular prompts for Mentor, Code, and Evaluation Agents.
   * Example: Mentor Agent prompt for beginners: “Given approach {input}, provide a simple hint in a motivational tone.”
2. **Set Up LangGraph (Day 2–3)**:
   * Define nodes and edges for agent interactions.
   * Implement event listeners for user\_input\_received, code\_submitted, etc.
3. **Implement FSM (Day 3–4)**:
   * Use XState or custom context to manage states (Problem Discussion, Coding, Evaluation).
   * Test state transitions with mock user inputs.
4. **Integrate AI Models (Day 4–5)**:
   * Connect GPT-4 API for primary tasks, test Hugging Face for fallback.
   * Optimize prompts for concurrency (e.g., batch API calls).
5. **Test Concurrent Flows (Day 5–6)**:
   * Simulate beginner, intermediate, and advanced user scenarios.
   * Verify agents work simultaneously without conflicts.
6. **Finalize and Document (Day 7)**:
   * Document agent flow in GitHub README.
   * Record video walkthrough showing concurrent agent interactions.

**Expected Outcomes**

* **Seamless User Experience**: Users receive real-time guidance, code feedback, and performance summaries without delays.
* **Efficient Agent Collaboration**: Agents work concurrently (e.g., Persona Manager updates skill level while Mentor Agent responds).
* **Scalable Architecture**: The system handles any DSA problem with generic prompts and test cases.
* **Team Learning**: All members gain hands-on experience with AI orchestration, prompt engineering, and state management.

This approach ensures the AI agents operate simultaneously, coordinated by the Orchestrator Agent and driven by an event-based, state-managed system. Let me know if you’d like to dive deeper into any specific agent’s logic, refine the workflow for a particular scenario, or discuss task assignments for your team!