Part 1: Candidate Brief

Hello! This challenge is designed to give us a sense of how you approach backend architecture, asynchronous processing, and API design. It's a starting point for a more in-depth conversation during our technical interview.

The Goal: AI Report Generation Service

Build a standalone API service that orchestrates a multi-step, "fan-out" process to generate a structured report. The service must be able to handle these potentially long-running jobs asynchronously without blocking the client.

Core Problem

The heart of this challenge is to design and build a robust, non-blocking API that manages the state of a complex background task. You will implement the orchestration logic on the backend, ensuring the system is efficient and resilient.

The Backend Workflow

1. A client sends a `POST` request to the service with a topic, for example, **"The Roman Empire"**.
2. The service must immediately accept the request, create a unique **job ID** for the task, and return that ID to the client with a `202 Accepted` status code.
3. **In the background**, the service begins the orchestration:
    - It first calls an AI service to generate a high-level outline for the topic.
    - Once the outline is received, it triggers a **concurrent fan-out** of API calls—one for each outline item—to generate the detailed content for that section.
4. The client can then use the job ID to poll a separate endpoint to check the status of the job and, once complete, retrieve the full, structured report.

API Requirements

You need to design and implement two API endpoints:

1. **Start a Job:** `POST /reports`
    - **Request Body:** `{ "topic": "The Roman Empire" }`
    - **Success Response (`202 Accepted`):** `{ "jobId": "some-unique-id" }`
2. **Get Job Status & Result:** `GET /reports/{jobId}`

- **Response (In Progress):** `{ ”jobId”: ”...”, ”status”: ”processing”, ”progress”: 0.25 }` *(Note: progress is a nice-to-have)*
- **Response (Completed):** `{ ”jobId”: ”...”, ”status”: ”completed”, ”report”: { ”topic”: ”The Roman Empire”, ”outline”: [...], ”sections”: { ”I. The Rise of the Republic”: ”...”, ”II. The Augustan Principate”: ”...” } } }`
- **Response (Failed):** `{ ”jobId”: ”...”, ”status”: ”failed”, ”error”: ”A description of what went wrong.” }`

Technical Requirements

- **Asynchronous Processing:** The `POST /reports` endpoint must return immediately. The entire AI generation process must happen in the background without blocking the HTTP request-response cycle.
- **State Management:** You will need to manage the state of each job (e.g., `processing`, `completed`, `failed`) and store its results. For this proof-of-concept, the state **does not need to be durable** across server restarts. However, be prepared to discuss how you would add persistence (e.g., using a database like Redis or PostgreSQL) in a production environment.
- **Type Safety:** If using TypeScript, we expect strong typing for your data models and API contracts. If using Python, we expect to see type hints.
- **API Key Management:** You will need to use your own API key (e.g., OpenAI, Anthropic). The key must be loaded from an environment variable and not hardcoded.

A Note on AI-Assisted Coding

We assume you’ll use AI-assisted editors and tools like **GitHub Copilot** or **Cursor** to accelerate your work. This is encouraged. However, be prepared to explain your architectural decisions, logic, and trade-offs.

Deliverables

1. **Source Code:** A link to a Git repository.
2. **README.md:** Clear instructions on how to install and run the service. Please include examples of how to use the API (e.g., with `cURL`).
3. **.env.example:** An example file listing the necessary environment variables.
4. **DEVLOG.md:** A brief commentary on your key architectural decisions (e.g., how you handled async processing, state management choices, etc.).