Data Engineer Take-Home (3 hours)

This assessment is designed to evaluate your proficiency in building production-grade data pipelines and backend services. You will be working with the GitHub Public Events API to build an end-to-end system that handles ingestion, raw storage, idempotent database loading, and advanced analytical modeling.

Overview

- **Role:** Data Engineer
- **Target Time:** 3–4 hours (though you have 4 days to submit).
- **Tech Stack:** Python, DuckDB, and an API framework of your choice (e.g., FastAPI, Flask).
- **AI Policy:** Usage of AI tools is expected and encouraged - we care about your ability to leverage modern tools to deliver high-quality code efficiently.

Evaluation Criteria

We prioritize a **production-ready mindset**. We will specifically look for:

- **Clean Engineering:** Modular code, error handling, and readability.
- **Correctness & Data Integrity:** Handling duplicates and ensuring data uniqueness.
- **Idempotency:** Designing the system so the load process is safe to run multiple times.
- **Advanced SQL:** Ability to work with nested/repeated data structures and complex window functions.

Part 1: Ingest Service (API)

Build a REST API service that acts as the entry point for raw data.

- **Endpoint:** POST /ingest.
- **Task:** Fetch the latest events from https://api.github.com/events and store them locally.
- **Background Processing:** The endpoint must return a response immediately. File I/O and fetching should happen in the background to avoid blocking the request.
- **Buffering:** Accumulate events in memory and write them to local files (JSON or JSONL) in batches (e.g., **150 events per file**).

Part 2: Database Persistence

Implement a process to move raw data into a structured environment using **DuckDB**.

- **Execution:** This can be a separate POST /load endpoint, a Python script, or a notebook.
- **Requirements:**
  - Load the raw JSON/JSONL files into DuckDB tables.
  - **Idempotency:** Ensure that re-running the load process does not create duplicate records.
  - **Data Modeling:** Choose a schema that efficiently supports the analytical queries in Part 3.

Part 3: Analytical & Modeling Layer

Using DuckDB, create the following views or tables. We expect **pure SQL** for these transformations where possible.

A) Top Repositories

Create a view of the **top 10 repositories** by event count. Your query should include a parameter for "days" (e.g., last 7 days), though it will run against whatever data you have ingested.

- **Required Fields:** repo_name, total_events, unique_users, push_events, first_event_at, last_event_at, and processed_at.

B) User Sessions

Model user activity into sessions based on the following logic:

- **Session Gap:** A new session begins after **30 minutes of inactivity**.
- **Output:** actor_login, session_id, session_start_at, session_end_at, and events_in_session.

C) Nested Data Manipulation

Demonstrate your ability to handle complex nested types.

1. **Base Table:** Create repo_events_nested where events are grouped into an ARRAY of STRUCTs per repository.
2. **Evolution:** Update this logic (or the table) to add a **new field inside the existing repeated struct** (e.g., an is_push boolean or event_score). The field must reside **within** the struct, not as a top-level column.

Part 4: Query API

Extend your service with endpoints to expose the data modeled in Part 3.

Endpoint	**Description**
GET /health	Basic service health check.
GET /top-repos?days=7&limit=10	Returns the top repositories based on the view from Part 3A.

GET /user-sessions?days=7&limit=50	Returns user session data from Part 3B.

Submission Instructions

- **Repository:** Host your code in a **private** GitHub repo and share it with: bironmatan.
- **Content:** Include the full Python project and a brief README explaining how to run the service and the load process.
- **Bonus:** Providing a Dockerized setup is highly encouraged.