Project Assignment: High-Scale Energy Ingestion Engine

Target Role: Backend Developer (Node.js/NestJS)

Experience Level: 2+ Years

Time Limit: 48 Hours

---

1. Executive Summary

Our Fleet platform manages a high-scale ecosystem of **10,000+ Smart Meters and EV Fleets**. To provide intelligence to Fleet Operators, we consume two independent data streams that arrive every **60 seconds** from every device.

The goal of this assignment is to build the core ingestion layer that handles these streams, correlates them, and provides fast analytical insights into power efficiency and vehicle performance.

---

2. Domain Context: Hardware & Logic

To build a successful system, you must understand the relationship between the two hardware sources:

- **The Smart Meter (Grid Side):** Measures **AC (Alternating Current)** pulled from the utility grid. It reports kwhConsumedAc, representing the total energy the fleet owner is billed for.
- **The EV & Charger (Vehicle Side):** The charger converts AC power into **DC (Direct Current)** for the battery. The vehicle reports kwhDeliveredDc (actual energy stored) and **SoC** (State of Charge/Battery %).
- **Power Loss Thesis:** In the real world, AC Consumed is always higher than DC Delivered due to heat and conversion loss. A drop in efficiency (e.g., below 85%) indicates a hardware fault or energy leakage.

---

3. Functional Requirements

A. Polymorphic Ingestion

Create a robust ingestion layer that recognizes and validates two distinct types of telemetry arriving via 1-minute heartbeats:

- **Meter Stream:** { meterId, kwhConsumedAc, voltage, timestamp }
- **Vehicle Stream:** { vehicleId, soc, kwhDeliveredDc, batteryTemp, timestamp }

B. Data Strategy (PostgreSQL)

Implement a database schema optimized for write-heavy ingestion and read-heavy analytics. Demonstrate the separation of:

- **Operational Store (Hot):** Fast access for "Current Status" (SoC, active charging).
- **Historical Store (Cold):** Optimized storage for billions of telemetry rows over time.

C. Persistence Logic: Insert vs. Upsert

Choose the correct operation for each data "temperature":

- **The History Path:** An **append-only (INSERT)** approach for every granular reading to build an audit trail for long-term reporting.
- **The Live Path:** A strategy (**UPSERT/Atomic Update**) ensuring the dashboard avoids scanning millions of rows to find a car's current battery percentage or a charger's last known voltage.

D. Analytical Endpoint

Implement GET /v1/analytics/performance/:vehicleId returning a 24-hour summary:

- Total energy consumed (AC) vs. delivered (DC).
- Efficiency Ratio (DC/AC).
- Average battery temperature.

---

4. Technical Constraints

- **Framework:** NestJS (TypeScript).
- **Database:** PostgreSQL.
- **Performance:** The analytical query **must not** perform a full table scan of the historical data.

---

5. Deliverables

1. **Source Code:** A GitHub repository link.
2. **Environment:** A docker-compose.yml to spin up the service and DB.
3. **Documentation:** A README.md explaining your architectural choices regarding data correlation and how the system handles 14.4 million records daily.