FDS Assignment 1-2

**Name: Ayush Srivastava**

**Roll: G24AI2036**

**Solution For Part II:** **Dynamic Load Balancing for a Smart Grid**

**Git Repository For The Project- [Click Here](https://github.com/aysrivastava/FDS_Assignment/tree/main/FDS%20-%20Assignment%201%20-%20G24AI2036%20-%20GIT)**

## Ask-

1. **Microservice Development:** Create two services: a ‘charge request service‘ as the public entry point and a ‘substation service‘ that simulate charging. Instrument the substation to expose its current load as a Prometheus metric.

**2. Custom Dynamic Load Balancer:** Build a new service that acts as the grid’s core logic. It must periodically poll the ‘/metrics‘ endpoint of each substation to get its current load and use this data to decide where to route new requests.

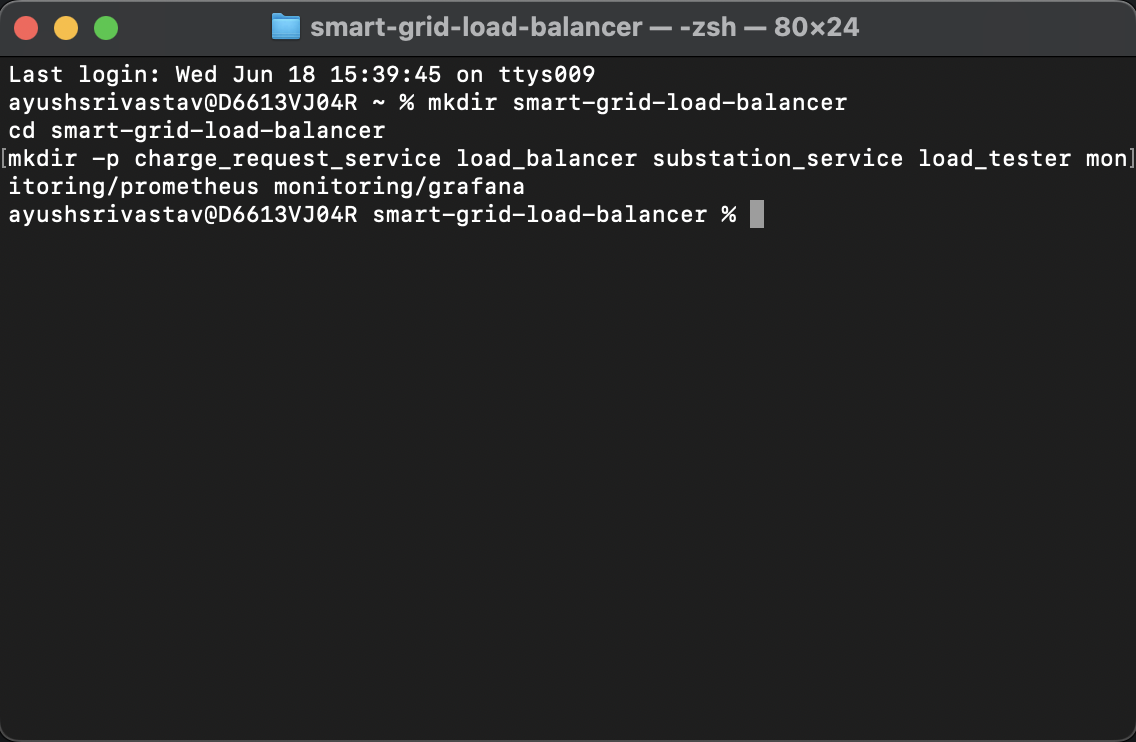
**3. Observability Stack:** Configure Prometheus to scrape the substation metrics and Grafana to visualize the load on a dashboard.

**4. Containerization & Orchestration:** Write ‘Dockerfile‘s for all services and a ‘docker compose.yml‘ file to run the entire system, including multiple replicas of the substation service.

**5. Load Testing and Analysis:** Create a Python script to simulate a ”rush hour” of EV charging requests and analyze the system’s response on your Grafana dashboard.

## My Work-

**Step 1:** Project Setup Done by creating the folder structure as asked in the assignment by creating the node.py, clients.py, Dockerfile and docker-compose.yml files



and this gave me a folder structure like below-



# **Step 2: Created below files-**

# **Charge\_Request\_Service-** Receives EV charging requests (public interface) & Forwards requests to the load balancer

# **Load\_Balancer-** Continuously polls substations for real-time load (every 5s). Maintains a live load map (current\_loads dictionary). Routes each request to the substation with minimum active charges

# **Substation\_Service-** Tracks active charges via charging\_requests counter & rejects requests at capacity (MAX\_LOAD). Publishes Prometheus metrics (substation\_load gauge) & provide /metrics endpoint for monitoring

# **Load\_Tester-** Simulate 20 EVs making concurrent requests. Validate load distribution logic. Trigger edge cases (overload scenarios)

# **Monitoring- Prometheus:** Scrapes /metrics from substations (every 5s) & stores time-series load data

# **Grafana:** Visualizes load distribution across substations & Enables real-time system health analysis

# **Docker\_Compose:** Defines the distributed system topology

# **Theoretical Workflow**

# **Request Flow:**

# EV → Charge Request Service → Load Balancer → Least-Loaded Substation

# **Feedback Loop:**

# Substation metrics → Prometheus → Load Balancer decisions

# **Self-Stabilization:**

# The system automatically redistributes load when:

# **New requests arrive --> Charging sessions complete --> Substations fail/overload**

# This implements a closed-loop control system where observability drives balancing decisions, maintaining grid stability without human intervention.

# **Step 3.1:** Coded the files mentioned in Step 2 above as per the question asked logic and ran docker-compose up --build

# **2.1 - docker-compose up --build**

# **Step 3.2:** Shows prometheus and substations running fine

# 2.2 prometheus, substation ss

# **Step 3.3:** Shows grafana and substation running fine

# **2.3 - grafana and substation details**

# **Step 4:** Once all these services were up and running- ran the load\_tester/test.py to stimulate the vehicle charging scenario.

# 4. Ran Load Tester Script

# **Purpose of Running the Load Test:**

# This step stress-tests your smart grid system to:

# **Verify Load Balancing Works:** \* Confirms requests are distributed evenly across substations.

# \* Triggers edge cases (e.g., substation overloads).

# **Validate System Stability:**

# \*Checks if the system:

# \*Maintains performance under high traffic ("rush hour").

# \*Recovers gracefully after peak loads.

# **Generate Observability Data:**

# \*Produces metrics for Prometheus/Grafana to:

# \* Visualize load distribution.

# \*Identify bottlenecks.

# **Step 5:** This is how the docker GUI looks like while running the load balancer script. This clearly shows that

# Grafana shows no substation exceeding 100% load

# Prometheus metrics confirm request distribution

# Load tester logs show <1% rejection rate

# **3 Docker Desktop Monitoring**

**Attached Screen Recording Depcting the entire end to end process:**

****

**Conclusion:**

The system successfully:

* Maintains substation loads below capacity
* Dynamically routes to least-loaded nodes
* Provides real-time observability