1. Objective

This project aims to design and implement a distributed smart grid system that dynamically balances Electric Vehicle (EV) charging requests across multiple substations. The primary objective is to maintain grid stability and efficiency by routing charging requests to the least loaded substation using real-time load data. Additionally, the system integrates observability through Prometheus and Grafana to monitor and analyze system performance under high load.

---

2. System Architecture

The system follows a **microservices-based architecture** composed of the following components:

- **Charge Request Service:** Entry point for EV charging requests (API Gateway).
- **Load Balancer:** Periodically polls each substation's `/metrics` endpoint and routes incoming requests to the least loaded substation.
- **Substation Service:** Simulates EV charging and exposes real-time load metrics in Prometheus format.
- **Monitoring Stack:**
    - **Prometheus:** Collects load metrics from substations.
    - **Grafana:** Visualizes load trends via a dashboard.

---

3. Implementation Overview

3.1 Substation Service

- Each substation maintains a `current_load` counter.
- Load increases on request start and decreases after a simulated charging delay.

- Metrics are exposed at `http://<host>:<port>/metrics` in Prometheus format.

3.2 Load Balancer Service

- Polls all substations at fixed intervals using `/metrics`.
- Parses the current load and routes new charging requests to the least loaded substation.
- Ensures fair distribution and prevents overloading.

3.3 Charge Request Service

- Exposes a `/charge` endpoint.
- Forwards incoming requests to the Load Balancer, which decides the target substation.

---

4. Monitoring Setup

- **Prometheus** is configured with a `prometheus.yml` file to scrape metrics from all running substation instances.
- **Grafana** is pre-loaded with a dashboard (`dashboard.json`) to visualize:
    - Real-time substation load.
    - Average and peak load during the test window.
    - Number of handled requests per substation.

---

5. Load Testing and Analysis

A load testing script simulates a "rush hour" scenario:

- **Test Duration:** 60 seconds

- **Total Requests:** 500 concurrent requests
- **Request Interval:** Randomized

Grafana Observations

- **Initial Phase:** All substations begin with equal load.
- **During Peak Load:** Load Balancer successfully distributes load based on real-time metrics.
- **Result:** No substation exceeded maximum capacity. Load was balanced fairly with minimal delay.

---

6. Docker and Compose Orchestration

All components are containerized and orchestrated via `docker-compose.yml`.

Substations are scaled using:

yaml
CopyEdit

```yaml
substation_service:
  deploy:
    replicas: 3
```

-
- Health checks and dependency declarations ensure proper startup sequence.
- Ports for Prometheus and Grafana are exposed for external access.

---

7. Repository Structure

css
CopyEdit

```
smart-grid-load-balancer/
```

```
├── charge_request_service/
│   ├── main.py
│   └── Dockerfile
├── load_balancer/
│   ├── main.py
│   └── Dockerfile
├── substation_service/
│   ├── main.py
│   └── Dockerfile
├── load_tester/
│   └── test.py
├── monitoring/
│   ├── prometheus/
│   │   └── prometheus.yml
│   └── grafana/
│       └── dashboard.json
├── docker-compose.yml
└── project_report.pdf
```

---

8. Challenges Faced

- Parsing Prometheus metrics efficiently with low latency.
- Handling race conditions when simultaneous requests are routed to the same substation.
- Synchronizing container startup order.

---

9. Future Improvements

- Integrate real-time alerting in Grafana for overload conditions.
- Use gRPC for low-latency inter-service communication.

- Implement predictive load distribution using historical data.