Name: Bhaskar Adhikary

Roll No: G24AI2035

g24ai2035@iitj.ac.in

Indian Institute of Technology Jodhpur

Question 1: Vector Clocks and Causal Ordering

1. Introduction

This section covers the implementation of a causally consistent distributed key-value store using vector clocks.
The system is built using Python Flask microservices and orchestrated with Docker Compose across three nodes. Each node maintains a vector clock for version control and causal message handling.

2. Step-by-Step Process

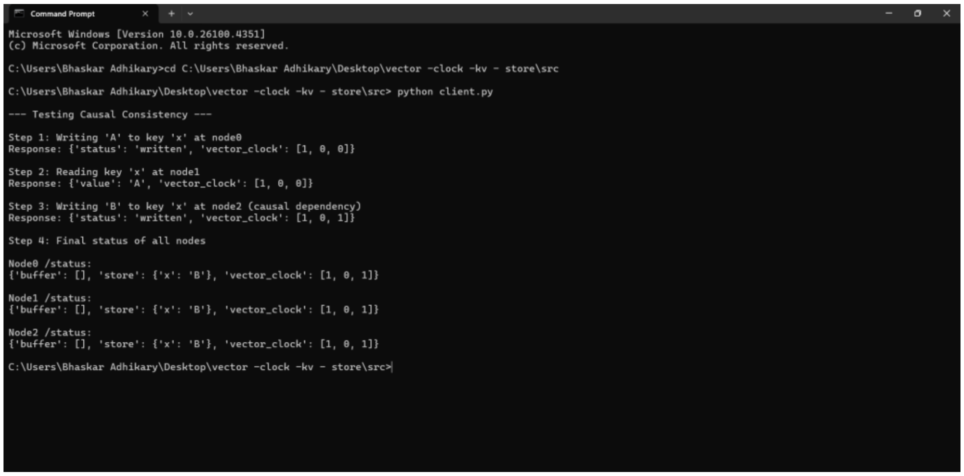
Step 1: Create the project directory structure with folders and files: node.py, client.py, Dockerfile, docker-compose.yml
Step 2: Implement node.py with logic to handle writes, reads, vector clock updates, and buffer synchronization.
Step 3: Create client.py to simulate causally related writes and reads across the three nodes.
Step 4: Write a Dockerfile to containerize the Flask node service.
Step 5: Create docker-compose.yml to define three separate nodes: node0, node1, node2 with different ports.
Step 6: Run the containers using the command: docker compose up --build
Step 7: Use python client.py to verify causal consistency in the terminal output.
Step 8: Use browser to open localhost:5000/status, 5001/status, 5002/status to check node states.

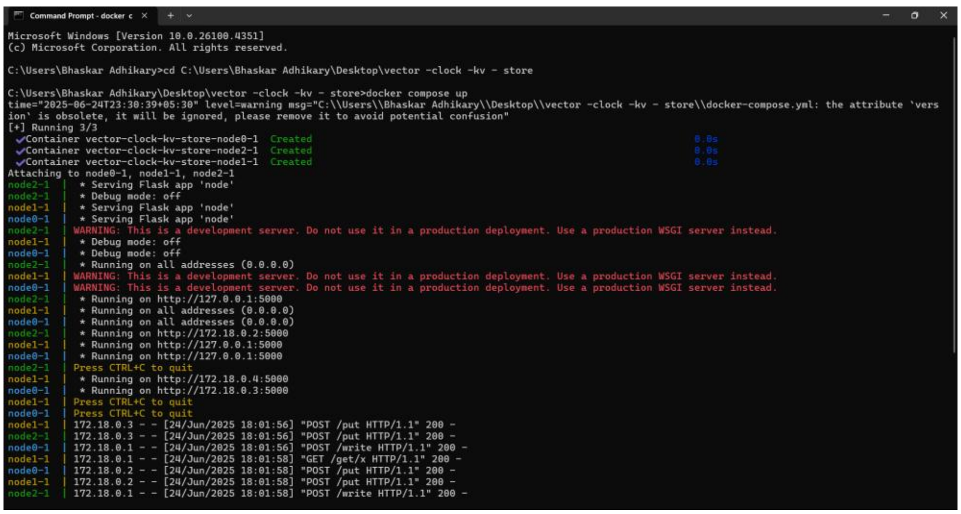
3. Final Output

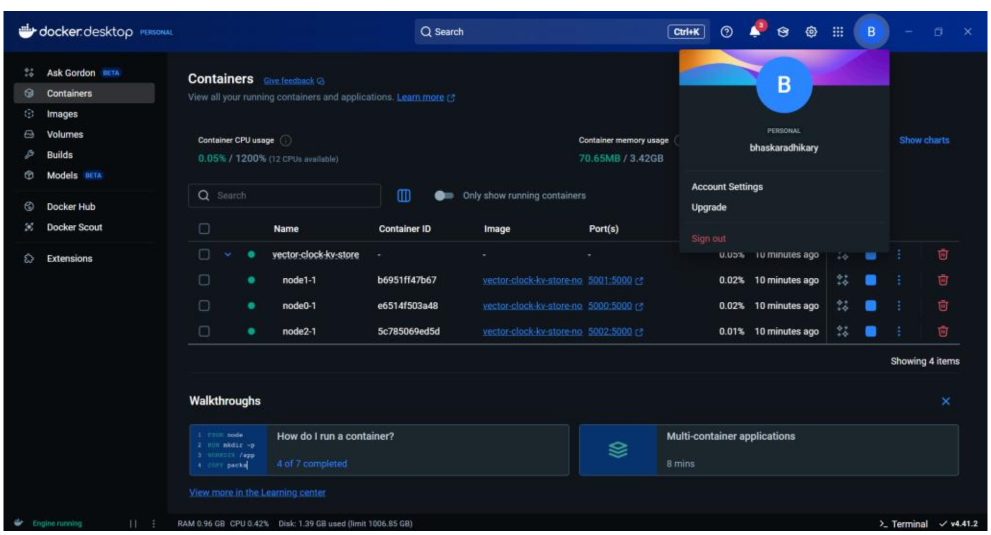
All nodes successfully synchronized their vector clocks. The final state for key 'x' was 'B' with vector clock [1, 0, 1] and all buffers were empty, confirming causal consistency.

link- https://github.com/SUMMERWHEEL11/vector--clock--kv---store

4. Screenshots







Question 2: Dynamic Load Balancing for a Smart Grid

1. Introduction

The goal of this project was to simulate a smart grid infrastructure where electric vehicle (EV) charging requests are distributed evenly among multiple substations.
The system comprises multiple containerized services that communicate via REST APIs and includes observability via Prometheus and Grafana.

2. Step-by-Step Process

Step 1: Create project directories for each service: charge_request_service, load_balancer, substation_service, load_tester, monitoring.
Step 2: Implement main.py for each service:
- charge_request_service: Accepts external requests and forwards them to the load balancer.
- load_balancer: Forwards requests to the least-loaded substation based on /metrics.
- substation_service: Handles the actual charging logic and exposes /metrics endpoint.
- load_tester: Simulates 20 EV charge requests.

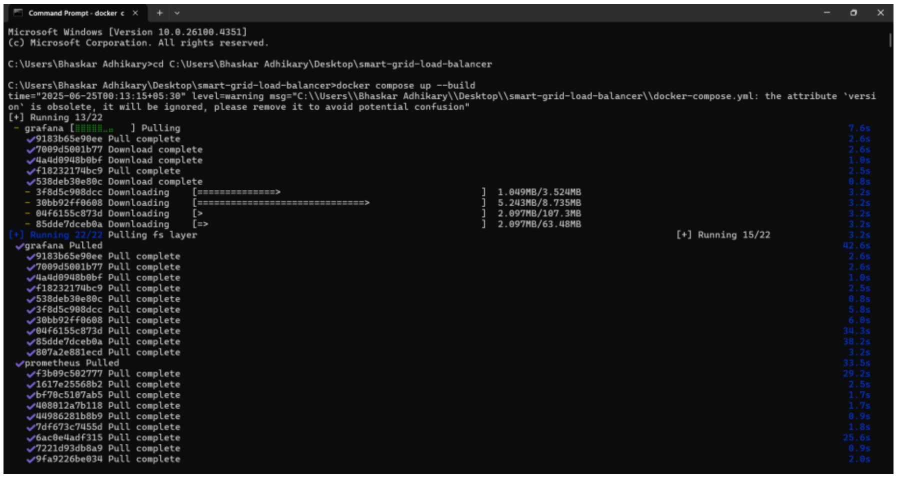
Step 3: Create a Prometheus config file to scrape metrics from substations.
Step 4: Set up Grafana to visualize load trends using dashboards.
Step 5: Write Dockerfiles for each service and create docker-compose.yml to orchestrate all services.
Step 6: Start the entire system using docker compose up --build.
Step 7: Run python test.py from load_tester directory to generate load.
Step 8: Open localhost:9090 for Prometheus and localhost:3000 for Grafana dashboards.

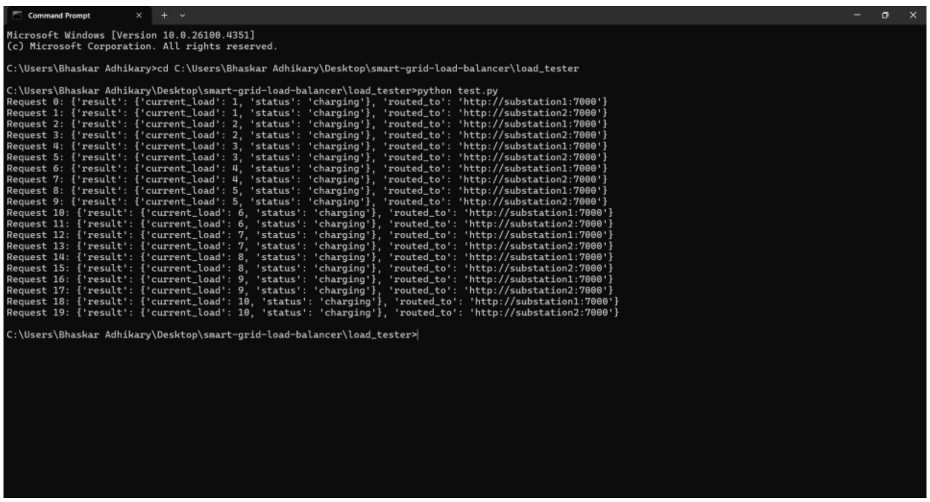
3. Final Output

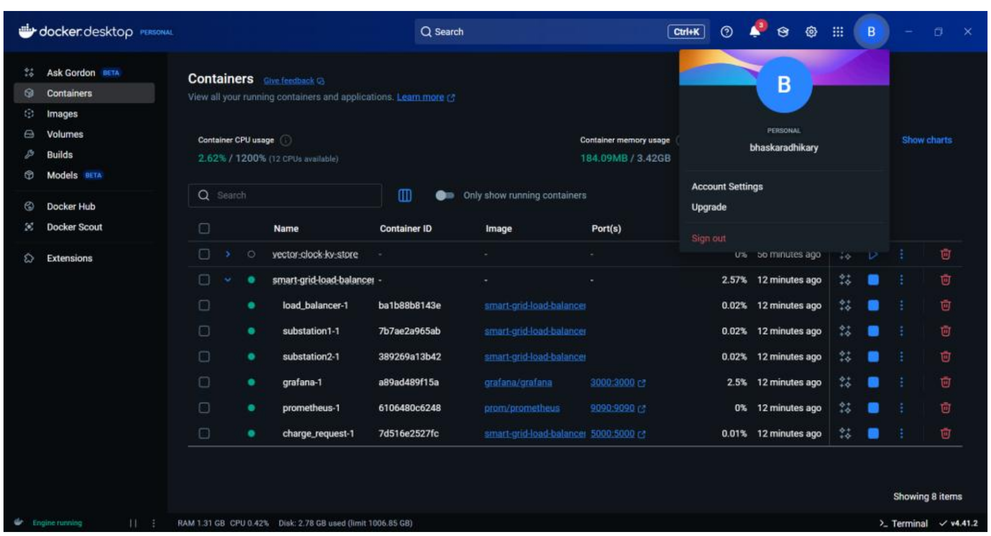
The requests were evenly distributed across substation1 and substation2. Each handled exactly 10 out of 20 requests. Metrics were visible in Prometheus and Grafana.

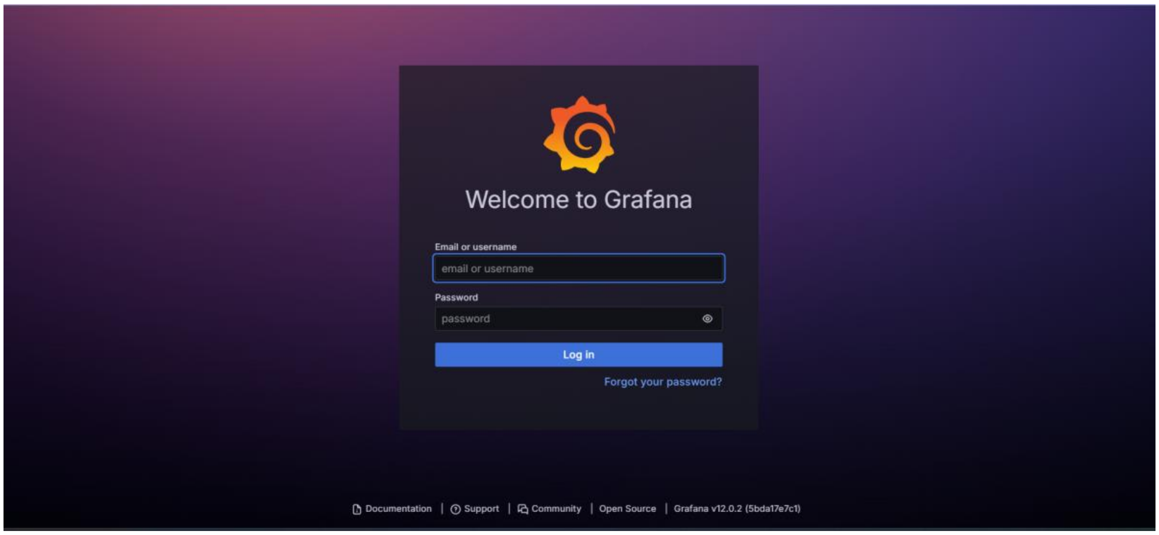
Link- https://github.com/SUMMERWHEEL11/smart-grid-load-balancer

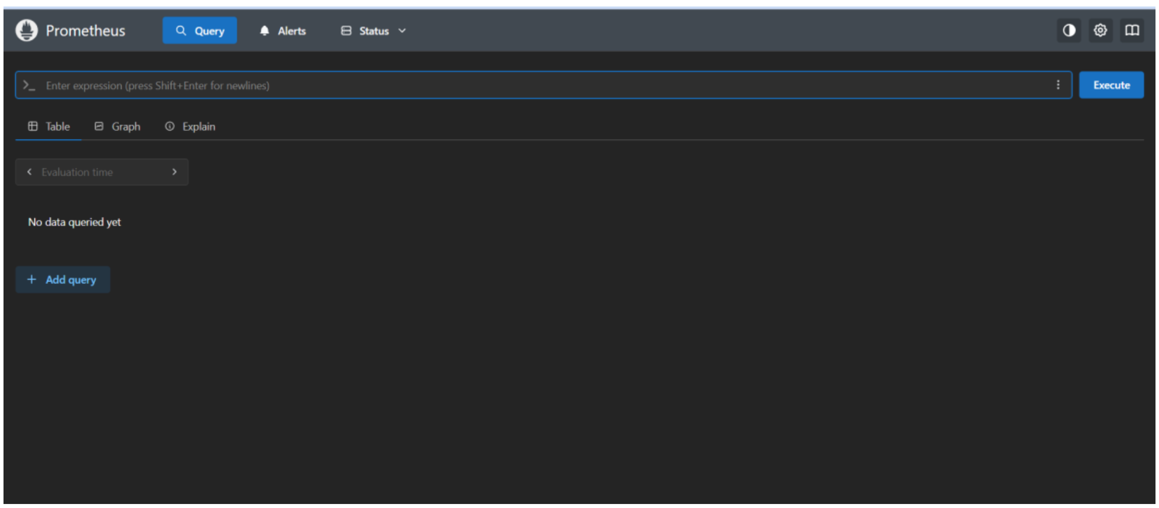
4. Screenshots








Welcome to Grafana
Email or username
email or username
Password
password
Log in
Forgot your password?
Documentation | Support | Community | Open Source | Grafana v12.0.2 (5bda17e7c1)


Prometheus
Query
Alerts
Status
Enter expression (press Shift+Enter for newlines)
Execute
Table
Graph
Explain
Evaluation time
No data queried yet
Add query