FUNDAMENTALS OF DISTRIBUTED SYSTEMS

Assignment 1

Anjitha Ravikumar

G24AI2043

PG Diploma in Data Engineering

IIT Jodhpur

Github Link: https://github.com/anjitharavikumar/FDS-Assignment

Table of Contents

1. Introduction

This report documents the implementation of two distributed systems:

- **Vector Clocks and Causal Ordering** – used to track the partial order of events in distributed systems and ensure consistency across data replicas.
- **Dynamic Load Balancing in Smart Grids** – focused on simulating real-time distribution of EV charging loads across substations using monitoring and intelligent routing mechanisms.

Both systems were developed in **Python**, containerized with **Docker**, and orchestrated using **Docker Compose**.

2. Tools & Technologies Used

- **Docker** – Enables containerization of services to emulate distributed nodes
- **Docker Compose** – Manages and orchestrates multiple interdependent services
- **Python** – Used as the core programming language for implementing backend logic
- **Flask** – A lightweight web framework used to build RESTful APIs
- **Prometheus** – Collects and scrapes metrics from running services
- **Grafana** – Visualizes real-time data through interactive dashboards

3. Vector Clocks and Causal Ordering

3.1 Overview

Vector Clocks track event causality in distributed systems. Each node maintains counters representing events across all nodes, enabling conflict resolution and proper event ordering. This ensures causal consistency - if one event affects another, all nodes process them in the correct sequence. Essential for distributed databases and collaborative systems.

3.2 Objective

Implement a key-value store across 3 nodes that maintains causal consistency using vector clocks. The system should detect out-of-order messages and delay their processing until causal dependencies are satisfied.

3.3 Components

- **node.py** - Handles the distributed key-value store operations and implements vector clock synchronization logic
- **docker-compose.yml** - Configures and deploys the 3-node cluster with assigned port mappings and node identifiers
- **client.py** - Provides a testing interface for sending PUT/GET requests and inspecting vector clock states

3.4 Architecture

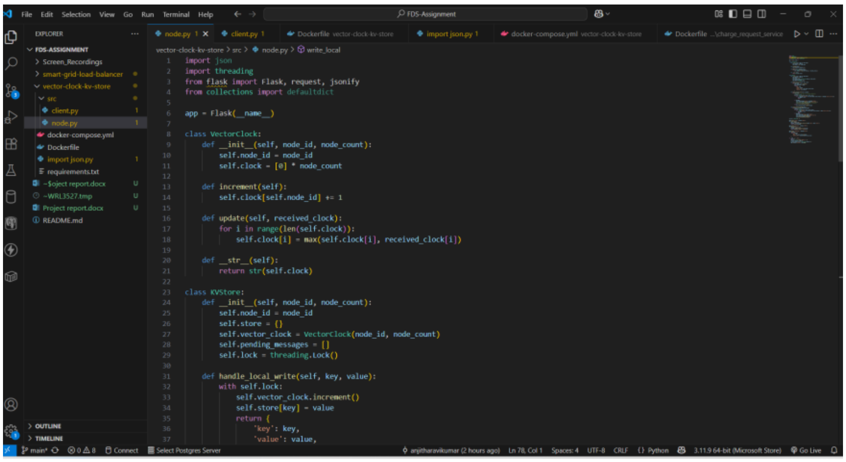
- The system has three distributed nodes, each running as a Python Flask service within a Docker container. Each node has its own local key-value store and a vector clock to track causality.
- When a node handles a local PUT request, it increments its vector clock and broadcasts the update to other nodes. Receiving nodes compare vector clocks to decide whether to apply the update immediately or buffer it for later.
- Buffered messages are periodically reviewed and applied as soon as their causal dependencies are satisfied.
- All nodes interact via REST APIs over a common Docker network.

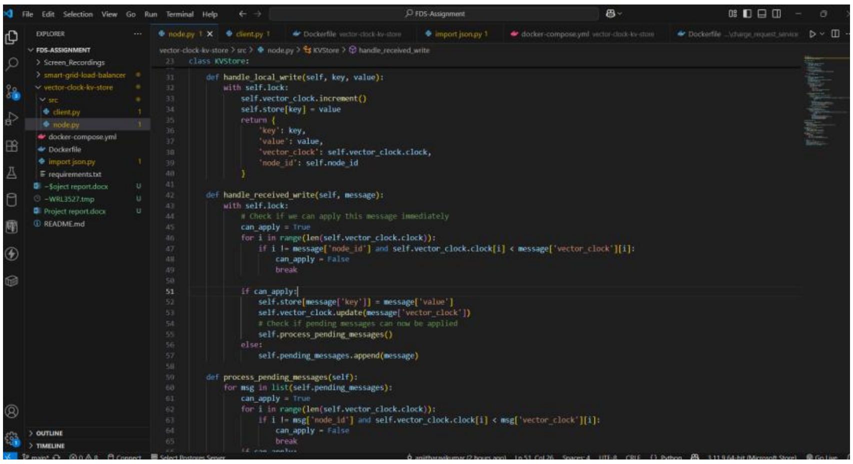
3.5 Process

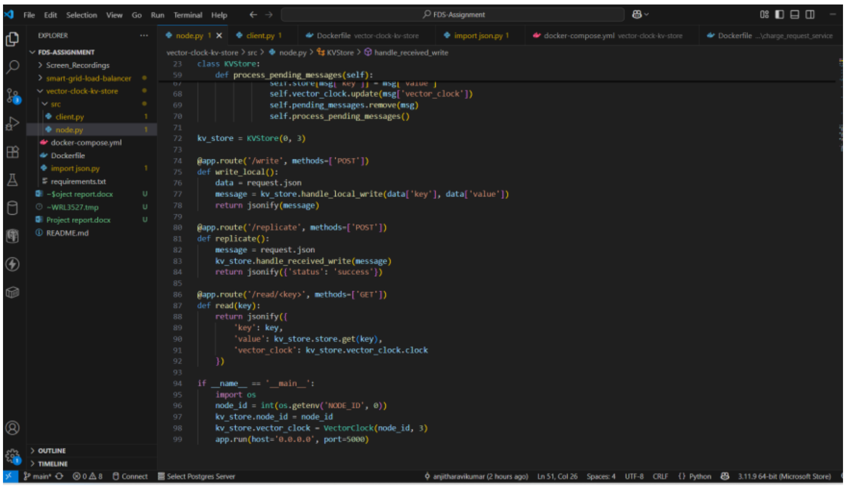
- node.py created using Flask to simulate each node in the distributed
- system.
- Each node maintains a local key-value store, a vector clock list [i, j, k] and a buffer for delayed messages.
- When a local PUT occurs, the node updates its vector clock and sends the update along with the clock to other nodes.
- Upon receiving a replicated message, a node verifies the causal delivery condition; if unmet, the message is buffered.
- Buffered messages are regularly checked to determine if they can be delivered.
- Docker Compose is used to launch three separate containers, one for each node.

3.6 Screenshots

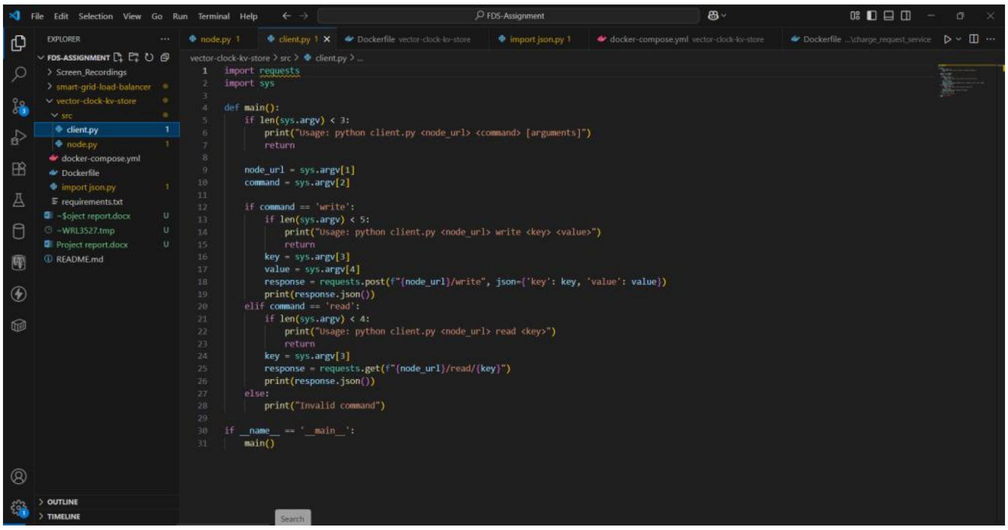
3.6.1 node.py



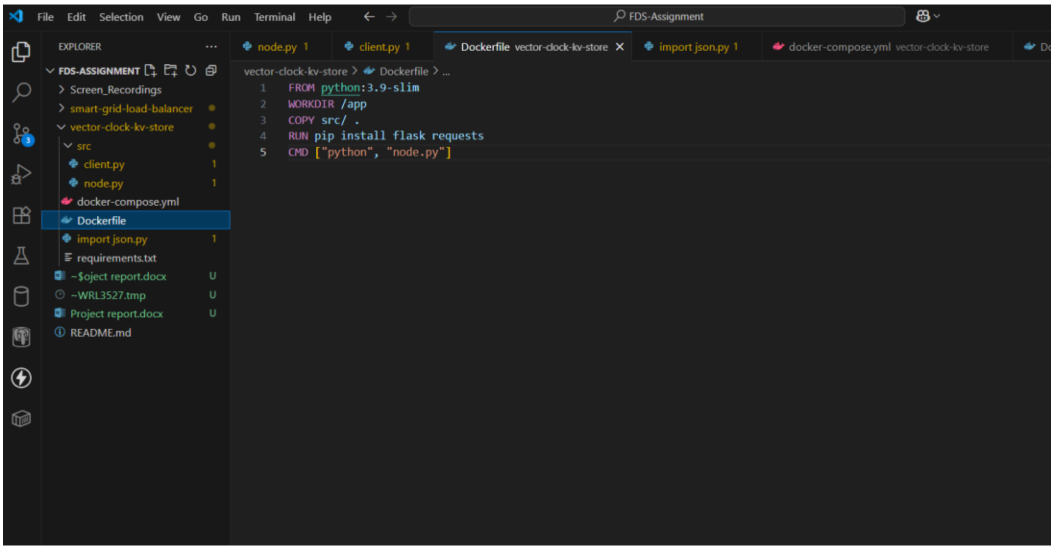




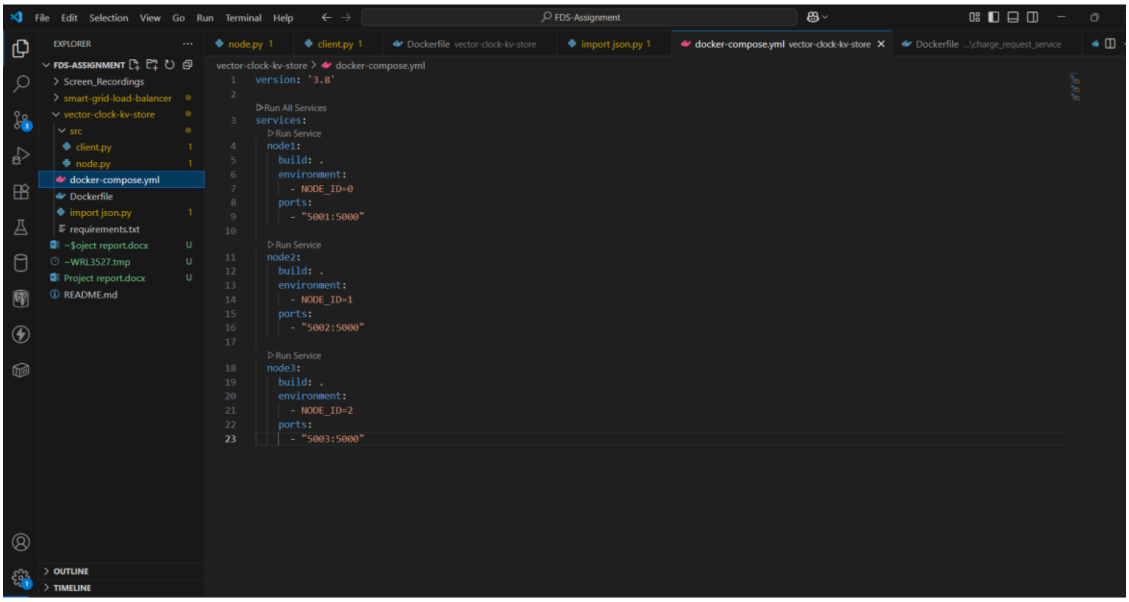
3.6.2 client.py



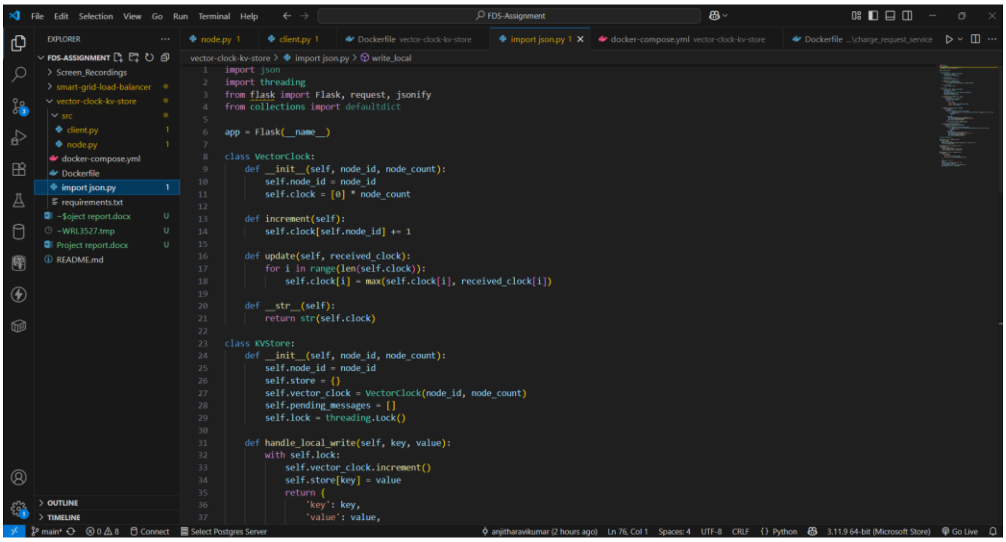
3.6.3 Dockerfile



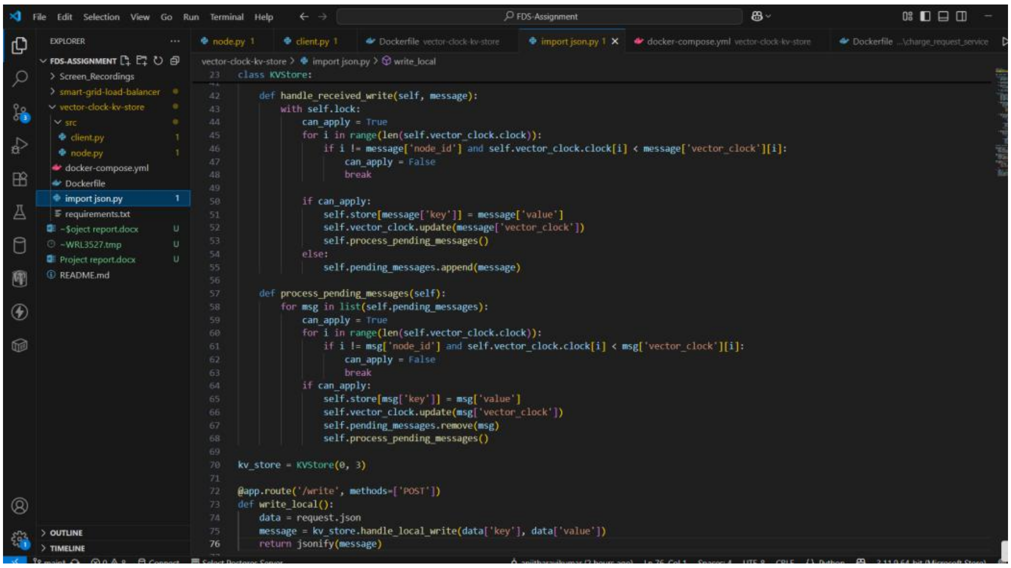
3.6.4 Docker-compose.yml



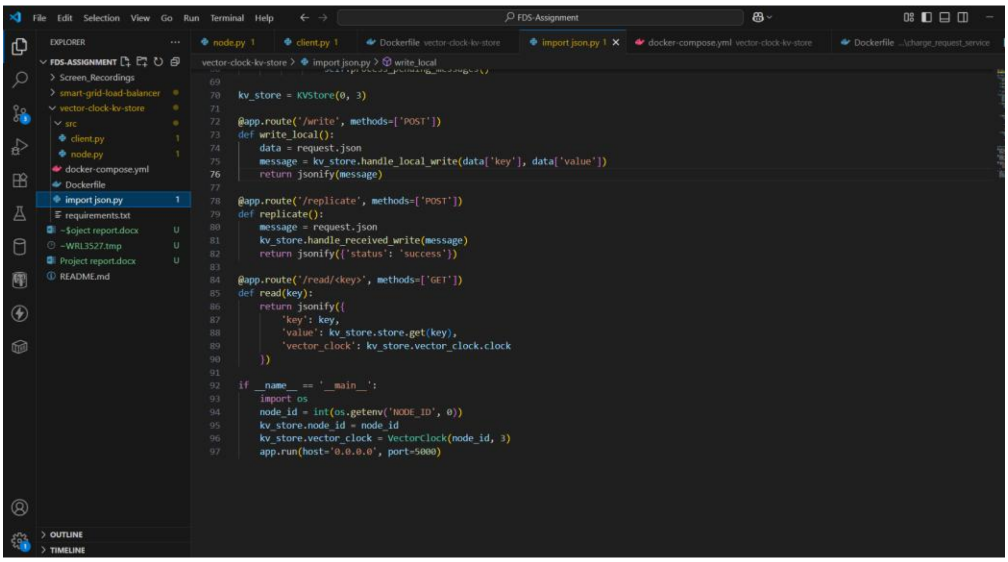
3.6.5 import json.py



```python
import json
import threading
from flask import Flask, request, jsonify
from collections import defaultdict

app = Flask(__name__)

class VectorClock:
    def __init__(self, node_id, node_count):
        self.node_id = node_id
        self.clock = [0] * node_count

    def increment(self):
        self.clock[self.node_id] += 1

    def update(self, received_clock):
        for i in range(len(self.clock)):
            self.clock[i] = max(self.clock[i], received_clock[i])

    def __str__(self):
        return str(self.clock)

class KVStore:
    def __init__(self, node_id, node_count):
        self.node_id = node_id
        self.store = {}
        self.vector_clock = VectorClock(node_id, node_count)
        self.pending_messages = []
        self.lock = threading.Lock()

    def handle_local_write(self, key, value):
        with self.lock:
            self.vector_clock.increment()
            self.store[key] = value
            return {
                'key': key,
                'value': value,
```

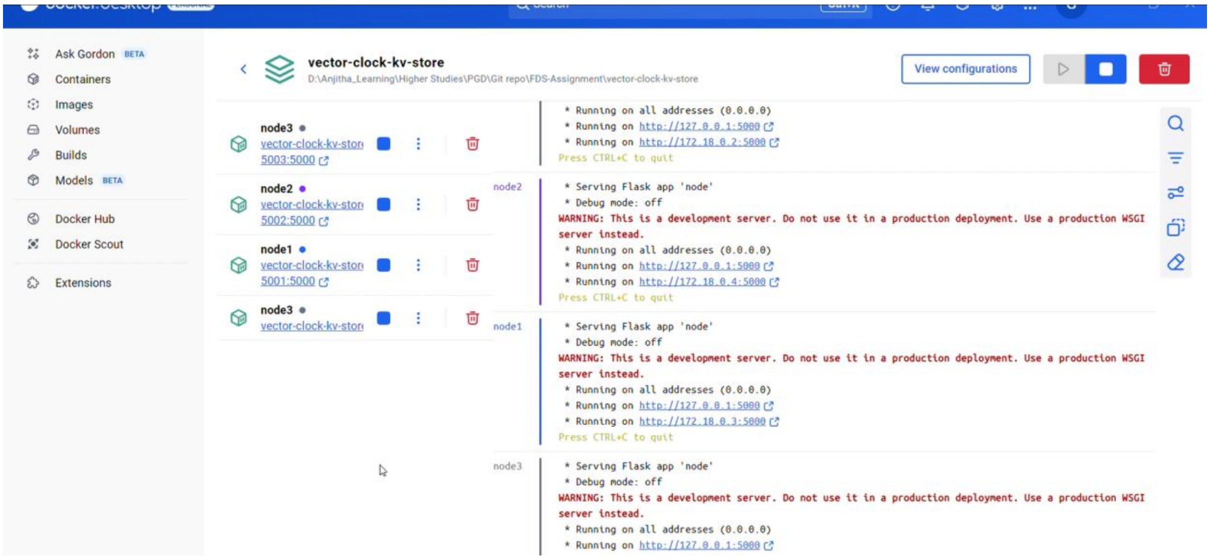


```python
class KVStore:
    def handle_received_write(self, message):
        with self.lock:
            can_apply = True
            for i in range(len(self.vector_clock.clock)):
                if i != message['node_id'] and self.vector_clock.clock[i] < message['vector_clock'][i]:
                    can_apply = False
                    break

            if can_apply:
                self.store[message['key']] = message['value']
                self.vector_clock.update(message['vector_clock'])
                self.process_pending_messages()
            else:
                self.pending_messages.append(message)

    def process_pending_messages(self):
        for msg in list(self.pending_messages):
            can_apply = True
            for i in range(len(self.vector_clock.clock)):
                if i != msg['node_id'] and self.vector_clock.clock[i] < msg['vector_clock'][i]:
                    can_apply = False
                    break
            if can_apply:
                self.store[msg['key']] = msg['value']
                self.vector_clock.update(msg['vector_clock'])
                self.pending_messages.remove(msg)
                self.process_pending_messages()

kv_store = KVStore(0, 3)

@app.route('/write', methods=['POST'])
def write_local():
    data = request.json
    message = kv_store.handle_local_write(data['key'], data['value'])
    return jsonify(message)
```



```python
kv_store = KVStore(0, 3)

@app.route('/write', methods=['POST'])
def write_local():
    data = request.json
    message = kv_store.handle_local_write(data['key'], data['value'])
    return jsonify(message)

@app.route('/replicate', methods=['POST'])
def replicate():
    message = request.json
    kv_store.handle_received_write(message)
    return jsonify({'status': 'success'})

@app.route('/read/<key>', methods=['GET'])
def read(key):
    return jsonify({
        'key': key,
        'value': kv_store.store.get(key),
        'vector_clock': kv_store.vector_clock.clock
    })

if __name__ == '__main__':
    import os
    node_id = int(os.getenv('NODE_ID', 0))
    kv_store.node_id = node_id
    kv_store.vector_clock = VectorClock(node_id, 3)
    app.run(host='0.0.0.0', port=5000)
```

3.6.6 Docker Desktop



Demo video is available in github

4. Dynamic Load Balancing for a Smart Grid

4.1 Overview

Dynamic load balancing intelligently distributes EV charging requests across substations based on real-time load. In a smart grid, where demand can spike suddenly, a centralized load balancer directs requests to the substation with the lowest load. This approach prevents overload, enhances efficiency, and improves overall system responsiveness. Tools like Prometheus are used to collect real-time data, while Grafana provides visualization of system performance.

4.2 Objective

To develop a dynamic, load-aware routing system that distributes electric vehicle charging requests across substations. The system should:

- Continuously monitor real-time load
- Route requests to the substation with the lowest load
- Visualize system metrics using Prometheus and Grafana

4.3 Components

- **substation_service/main.py**: Processes charging requests and exposes load metrics

- **load_balancer/main.py**: Contains the routing logic based on current load data
- **test.py**: Simulates EV charging requests to evaluate the load balancing system
- **prometheus.yml**: Sets up Prometheus to scrape metrics from substations
- **dashboard.json**: Configures the Grafana dashboard for real-time monitoring

4.4 Architecture

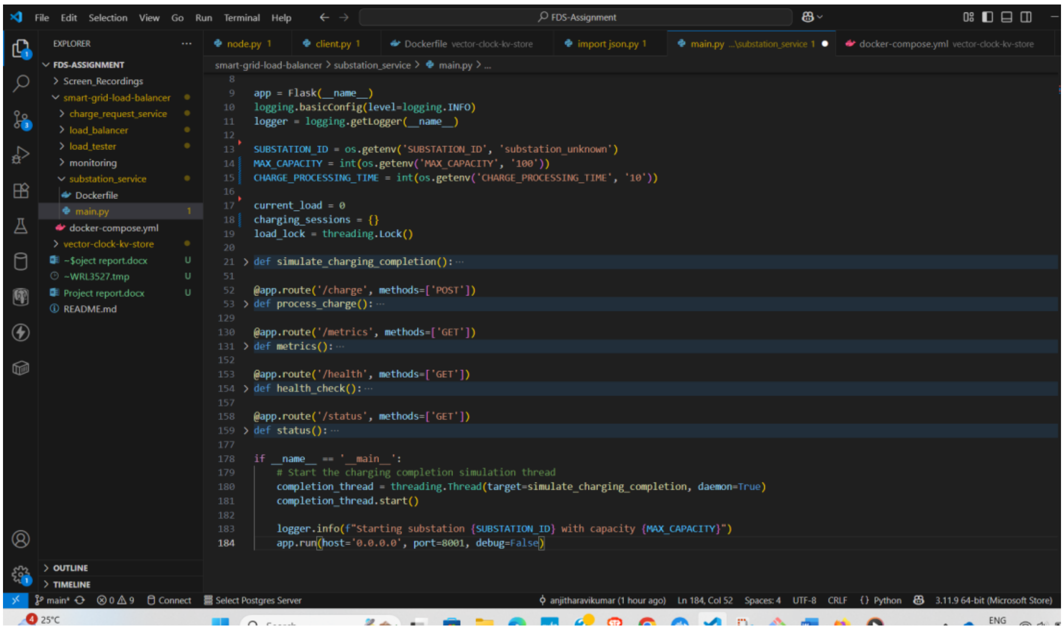
- The system includes two substation services, a centralized load balancer, and monitoring tools (Prometheus and Grafana).
- Substations handle EV charging requests and expose their current load through a /metrics endpoint.
- The load balancer regularly polls these metrics and directs new requests to the substation with the lowest load.
- Prometheus collects and stores time-series data from the substations.
- Grafana connects to Prometheus to display real-time load metrics on a custom dashboard.

4.5 Process

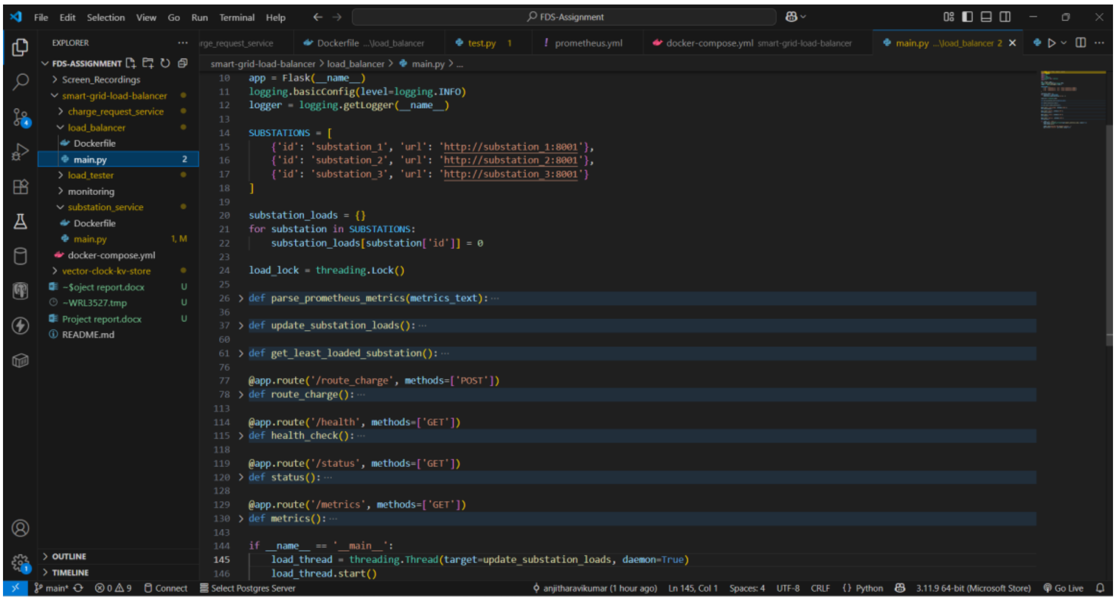
- Developed substation_service/main.py with /charge and /metrics endpoints.
- Built load_balancer/main.py to fetch substation metrics and route requests to the least-loaded node.
- Created test.py to simulate 50 EV charging requests.
- Set up Prometheus to scrape metrics from the substation services.
- Integrated Grafana with Prometheus and imported a custom dashboard for visualization.

4.6 Screenshots

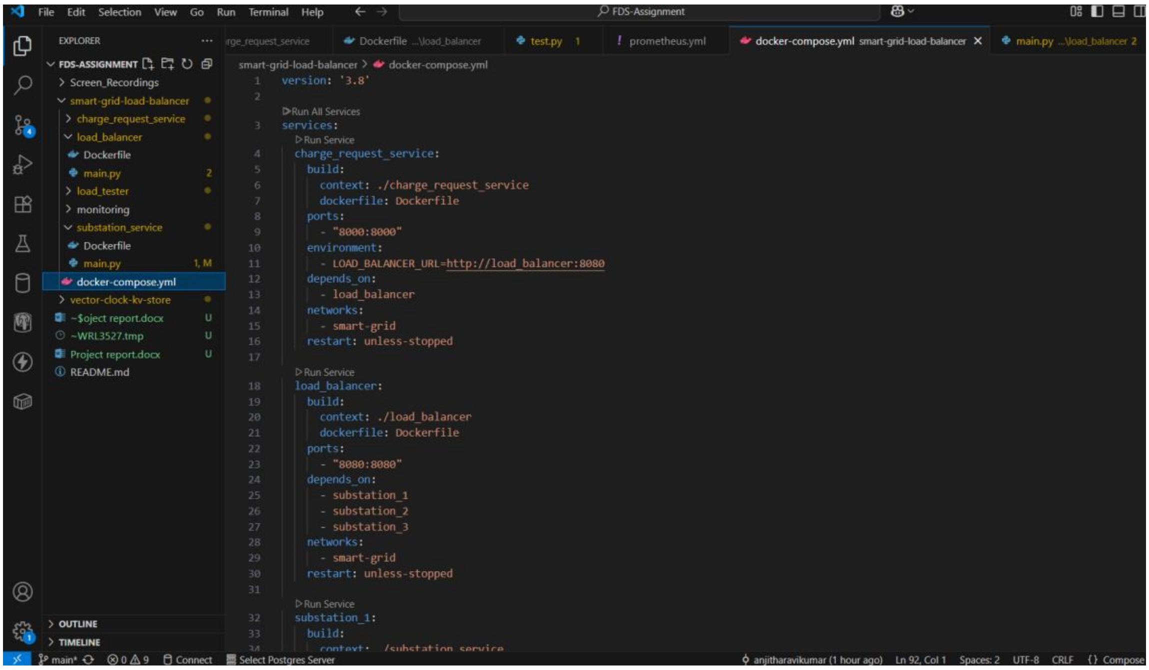
4.6.1 substation_service/main.py



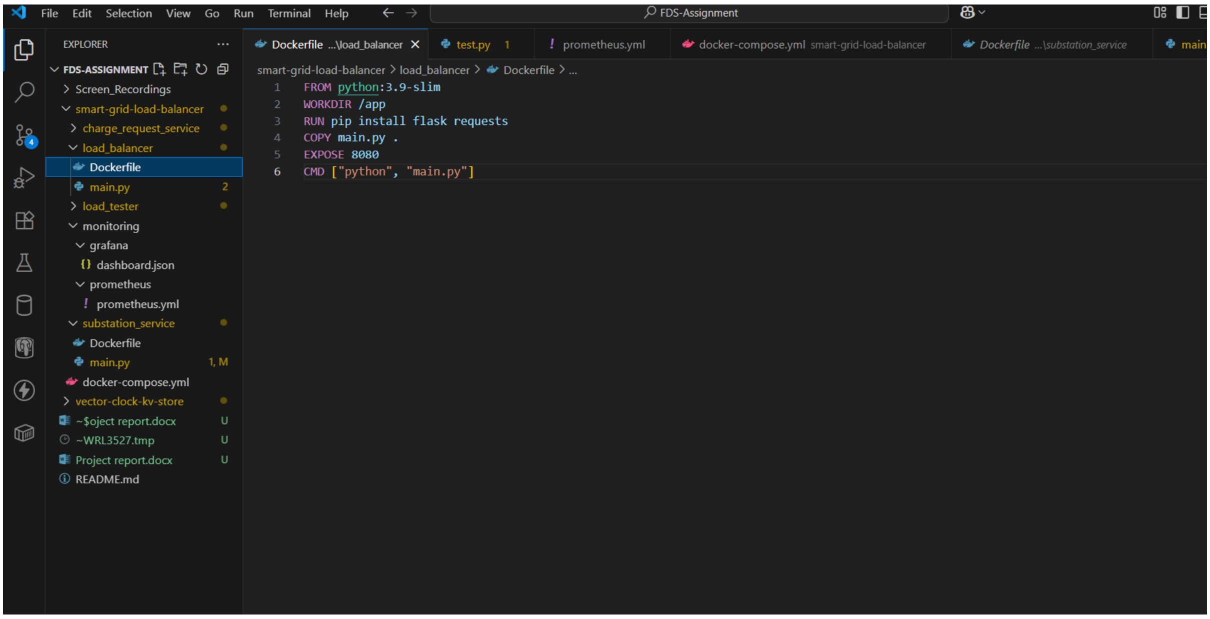
4.6.2 load_balancer/main.py



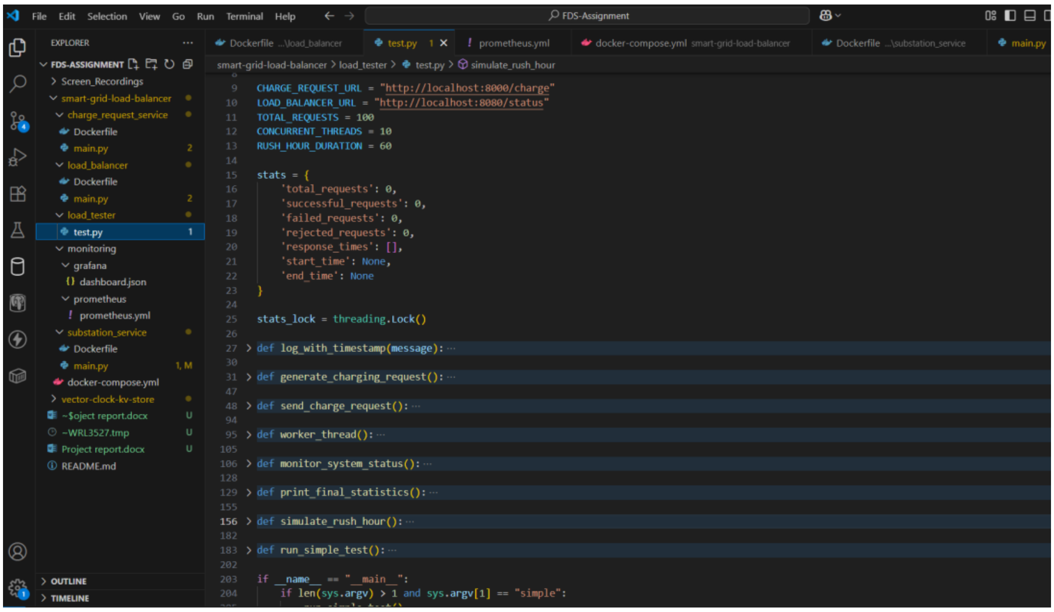
4.6.3 Docker-compose.yml



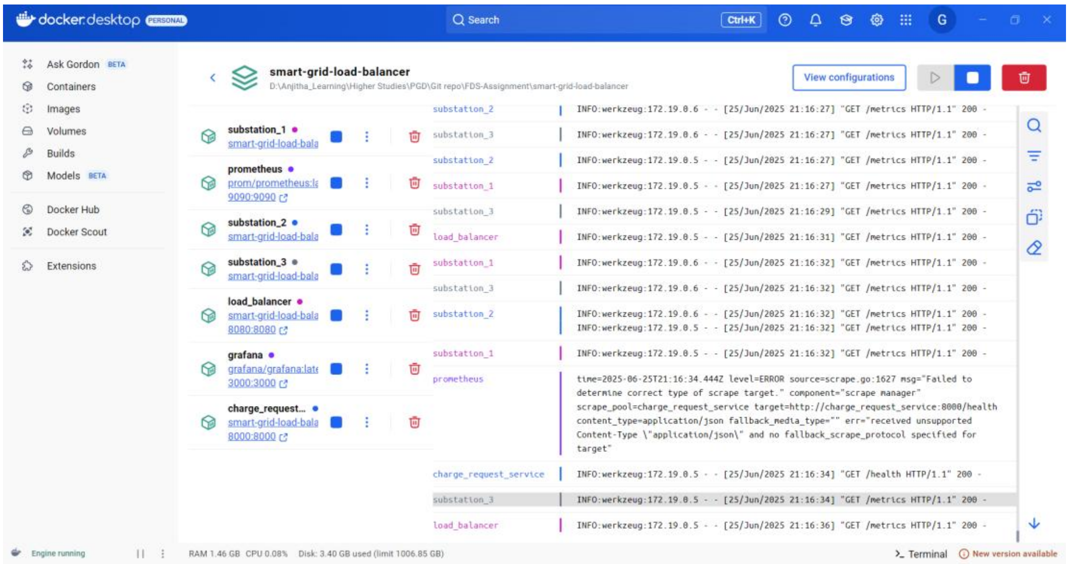
4.6.4 load_balancer/Dockerfile



4.6.5 load_tester/test.py



4.6.6 Docker Desktop



Demo video is available in github