EV Charging Simulation

Development, Deployment and Results Report

Your Name
your.email@example.com
Institution / Course Name

October 29, 2025

Contents

1 Executive Summary

This project implements a distributed Electric Vehicle (EV) charging simulation platform using mainly Python, Apache Kafka, and Docker. It models the interactions between a central management service, multiple charging points (CP) with engines, monitors supervising the state of engines, and driver applications. All services are containerized and orchestrated via Docker Compose, enabling reproducible deployment and communication through Kafka topics, TCP sockets, and HTTP endpoints.

2 System Architecture

2.1 Overview

Architecture used in this system is microservice-based architecture. It can be observed in the fact that all the components run independently and are event-driven. The system contains numerous crucial components which communicate with each other thanks to Kafka and TCP sockets. The main objective of this system is to simulate real-world EV chargin system. In the reality it is almost impossible to mantain manual controll over every component of such system with all plausible faults or user's errors. Our architecture tries to automate the process of communication between components and prepare the system for the unpredictability of an outside world.

2.2 Architecture Diagram

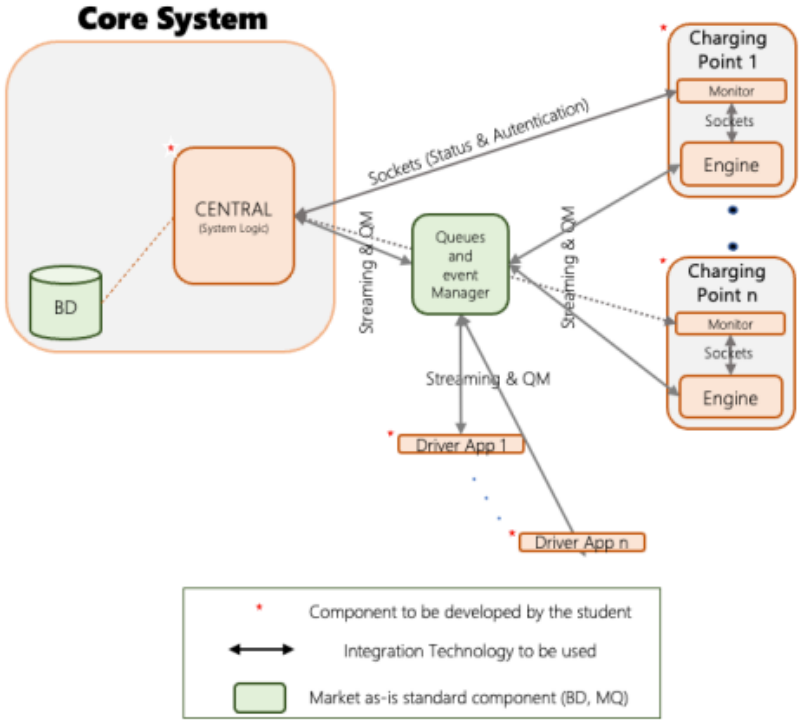

Figure 1: Architecture of the EV Charging Simulation system.

2.3 Components Description

Component	Role	Technology	Port
Central	Manages registration, commands, and telemetry	FastAPI, aiokafka	8000
CP Engine	Simulates charging session	Python, Kafka, Sockets	8001
CP Monitor	Monitors Engine and reports to Central	Python, HTTP, Sockets	–
Driver	Simulates driver requests	FastAPI, Python, Kafka	–
Kafka	Message broker	Apache Kafka	9092

Table 1: System components and their technologies.

Responsibilities of each component:

- Central:
  - Acts as the central controller and coordinator
  - Provides a FastAPI dashboard for readability
  - Receives registration requests from new CP Monitors
  - Receives heartbeats from the CP Monitors
  - Receives fault/health notifications
  - Stores state about all charging points
- CP Monitor:
  - Oversees a single CP Engine's health and connectivity
  - Sends heartbeats to the Central
  - Performs TCP healthcheck against the CP Engine
  - Notifies Central of faults
  - Registes with the Central at startup
- CP Engine:
  - Simulates the physical charging point
  - Publishes telemetry messages to Kafka topics
  - Listens for start/stop requests
  - Provides a TCP health endpoint for the Monitor
- Kafka:
  - Simulates the physical charging point
  - Publishes telemetry messages to Kafka topics
  - Listens for start/stop requests
  - Provides a TCP health endpoint for the Monitor
- Docker:

  - Simulates the physical charging point
  - Publishes telemetry messages to Kafka topics
  - Listens for start/stop requests
  - Provides a TCP health endpoint for the Monitor

- Data base:
  - Simulates the physical charging point
  - Publishes telemetry messages to Kafka topics
  - Listens for start/stop requests
  - Provides a TCP health endpoint for the Monitor

3 Development Process

3.1 Environment Setup

List tools and libraries used (Python 3.11, Kafka 3.7, Docker, etc.).

3.2 Repository Structure

```
ev-charging-simulation/
|-- docker/
|   |-- Dockerfile.central
|   |-- Dockerfile.cp_e
|   |-- Dockerfile.cp_m
|   |-- Dockerfile.driver
|-- docker-compose.yml
|-- Makefile
`-- evcharging/
    |-- apps/
    |   |-- ev_central/
    |   |-- ev_cp_e/
    |   |-- ev_cp_m/
    |   `-- ev_driver/
    `-- common/
```

3.3 Key Features

- Asynchronous Kafka-based communication.
- State management using Python enumerations.
- Health endpoints via FastAPI.
- Docker Compose-based deployment and inter-service networking.

4 Deployment Details

4.1 Docker Setup

Explain the purpose of each Dockerfile and the role of docker-compose.

Listing 1: Excerpt from docker-compose.yml

```
services:
  ev-central:
    build:
      context: .
      dockerfile: docker/Dockerfile.central
    ports:
      - "8000:8000"
    depends_on:
      kafka:
        condition: service_healthy
```

4.2 Health Checks

Describe how health checks are implemented (via curl, nc, etc.).

4.3 Kafka Topics

List topics such as:

- central.commands
- cp.telemetry
- cp.status
- driver.requests

5 Testing and Results

5.1 Testing Procedure

Describe how containers were launched, logs monitored, and communication verified.

5.2 Example Logs

```
INFO: CP cp-1 registered with Central successfully
INFO: Received telemetry update for cp-1
INFO: Driver command processed successfully
```

5.3 Results

Summarize key outcomes (successful registration, telemetry flow, driver interaction).

6 Troubleshooting and Corrections

Document encountered issues and solutions:

- Fixed missing wait-for-kafka script.
- Added curl to support health checks.
- Adjusted Kafka listener configuration.

7 Conclusion

Summarize the project outcomes, limitations, and future improvements (e.g. database persistence, dashboard UI).

A Appendices

A.1 Run Instructions

```
docker compose build
docker compose up
```

A.2 Sample API Endpoints

- `GET /health`
- `POST /cp/register`