A-LSTM and XGBoost - Enhanced Digital Twin for Smart Mobility and Environmental Monitoring in Westminster, London

By

Manas Prasun Pandey

Submitted to

The University of Roehampton

In partial fulfilment of the requirements

for the degree of

Master of Science

in

Data Science

Declaration

I hereby certify that this report constitutes my own work, that where the language of others is used, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of others.

I declare that this report describes the original work that has not been previously presented for the award of any other degree of any other institution.

Manas Prasun Pandey

15th August, 2025

Acknowledgements

I would like to express my deepest gratitude to my supervisor and mentor, **Dr. Sameena Naaz**, for her unwavering support, insightful guidance, and continuous encouragement throughout the course of this project. Her expertise and mentorship have been instrumental in shaping the direction and depth of my work.

I am also sincerely thankful to **Dr. Mamoona Humayun**, whose academic leadership and thoughtful feedback provided invaluable clarity during critical stages of this dissertation. Her perspective helped me refine my approach and strengthened the overall quality of the research.

I extend my appreciation to the faculty at the University of Roehampton for creating a learning environment that fosters innovation and critical thinking. Lastly, I thank my family and friends for their constant support and motivation during this academic journey.

Abstract

Urban environments face increasing challenges in managing traffic congestion and air pollution, particularly in densely populated districts such as Westminster, London. These challenges necessitate advanced systems capable of integrating real-time data and forecasting to support proactive urban planning. This project presents the development of a machine learning-enhanced Digital Twin designed to facilitate smart mobility and environmental monitoring for Westminster.

The proposed system integrates real-time and historical data from multiple open-source providers, like API from Imperial College of London and OpenWeatherMap (Chargeable for large volume data). These data streams are ingested and stored via a modular backend architecture that supports both real-time and batch processing, enabling responsive performance and comprehensive analysis. The system exposes key services through RESTful APIs, allowing seamless communication between data sources, machine learning models, and the interactive frontend interface.

Two core models power the system: an Adaptive Long Short-Term Memory (A-LSTM) network for generating synthetic traffic data and an XGBoost regressor for forecasting Air Quality Index (AQI) and individual pollutant levels. The A-LSTM model addresses the issue of sparse or unavailable traffic data by simulating realistic traffic flow based on air quality and meteorological patterns. Meanwhile, the XGBoost model forecasts pollutant concentrations using structured inputs derived from weather, traffic, and temporal features. Both models are retrained periodically, with performance metrics logged to ensure continued accuracy and adaptability.

The frontend interface, built with a 3D geospatial visualization engine, enables users to explore historical trends and simulated future scenarios. Features include interactive traffic overlays, AQI indicators, and weather visualizations. Scenario simulation tools allow users to adjust traffic conditions and observe the projected environmental impact in real time.

Evaluation of the system included scenario-based tests, such as simulating increased traffic during peak hours, which led to a forecasted 12%–15% increase in NO_2 concentrations. The A-LSTM model achieved a classification accuracy of 79.27%, significantly outperforming

traditional LSTM models, while XGBoost regressors achieved R^2 values above 0.88, with NO_2 predictions reaching 0.994.

Despite challenges such as the absence of real-time sensor validation and inconsistencies in external APIs, the system performed reliably across both technical and user-facing evaluations. The containerized deployment and public web accessibility ensure that the solution can scale and integrate into broader smart city initiatives.

In conclusion, this project demonstrates the feasibility of using machine learning and Digital Twin technologies to enhance urban foresight. It offers a practical, data-driven platform for policymakers and urban planners seeking to evaluate environmental interventions and promote sustainable, intelligent city management.

Table of Contents

List of Figures

List of Tables

Chapter 1- Introduction

As urbanization accelerates, cities face mounting challenges in ensuring sustainable mobility and environmental health. Digital Twin technology, which creates real-time digital replicas of physical environments, offers a transformative approach to urban management by integrating live data streams with 3D simulations. This proposal presents a project to develop a Machine Learning-enhanced Digital Twin for the Westminster district in London, focusing on smart mobility and environmental monitoring.

London's Westminster is a densely populated urban area characterized by complex traffic flows and significant air quality concerns. The availability of open APIs from Imperial college of London and environmental monitoring networks like Open Weather Maps provides an excellent foundation for integrating data into a dynamic Digital Twin. This system aims to visualize and simulate real-time traffic patterns and environmental conditions, offering insights for policy-making, emergency planning, and sustainability analysis.

The motivation for this project lies in the growing need for cities to transition toward data-driven governance and climate-responsive infrastructure. By enhancing the Digital Twin with machine learning models, the project will provide predictive insights into traffic congestion and air quality dynamics. Additionally, a robust frontend will enable real-time interaction and scenario simulations, empowering stakeholders to explore and assess the potential impact of urban interventions.

This project seeks to bridge the gap between real-time data availability and its application in decision-making platforms, contributing to both theoretical discourse and practical implementation of intelligent, predictive urban management systems.

1.1 Problem Statement

Urban environments like Westminster face the dual challenge of managing increasing transportation demand and mitigating environmental impacts. Traffic congestion contributes significantly to air pollution, with pollutants such as nitrogen dioxide (NO2) and particulate

matter (PM2.5) exceeding recommended levels in many parts of London. These issues not only degrade the quality of life for residents but also pose serious public health risks.

Although data from Imperial College of London and Open Weather Maps are available in real-time, they are currently underutilized for integrated analysis and predictive modelling. City planners often rely on retrospective studies and fragmented datasets, which limits their ability to make proactive decisions. There exists a gap in leveraging this real-time data within an interactive and responsive system that could provide actionable insights into urban mobility and environmental trends.

A Machine Learning-enhanced Digital Twin that integrates real-time transport and environmental data could significantly enhance the city's ability to model, predict, and respond to urban dynamics. For instance, by simulating the impact of road closures, vehicle flow restrictions, or introducing green zones, authorities could forecast changes in air quality and congestion. Such a system would serve not only as a monitoring dashboard but also as a decision-support tool for scenario analysis based on data-driven forecasts.

The key issues this project aims to address are:

- The lack of integrated, real-time platforms combining mobility and environmental data.
- The absence of predictive capabilities for traffic congestion and air quality forecasting.
- The difficulty in visualizing and interacting with urban data and policy simulations.
- The need for accessible, user-friendly tools for non-technical stakeholders in urban planning.

By targeting these gaps, the proposed Machine Learning-enhanced Digital Twin will contribute to more informed, responsive, and sustainable urban governance. The inclusion of an interactive frontend will ensure accessibility and empower users to explore dynamic urban scenarios effectively.

1.2 Aims and Objectives

Aim: To develop a real-time Machine Learning-enhanced Digital Twin for Westminster, London that integrates smart mobility and environmental data to support predictive urban decision-making.

Objectives:

1. **Data Acquisition and Integration:**
    - Collect and integrate live traffic data from Open Weather Maps APIs and historical congestion data.
    - Acquire real-time and historical air quality data from Open Weather Maps API.
    - Merge datasets with meteorological data for enhanced pollution forecasting.
    - Harmonize datasets for training machine learning models.
2. **System Architecture and Development:**
    - Design a modular backend with Python (FastAPI) for data ingestion and storage.
    - Implement Machine Learning models:
        - Traffic Prediction: Gradient Boosting or XGBoost to forecast congestion levels.
        - Air Quality Prediction: Random Forest Regressor or LSTM models.
    - Set up a relational or NoSQL database to store historical, real-time, and predictive data.
3. **Frontend Development:**
    - Develop a frontend using JavaScript frameworks and CesiumJS for 3D visualization.
    - Create an interactive dashboard to visualize live data overlays and prediction outputs.
    - Build a simulation interface to allow users to modify parameters and view scenario outcomes.
4. **Simulation and Scenario Analysis:**
    - Integrate ML models into the frontend to allow users to simulate interventions.

- Visualize predicted outcomes for traffic flow and air quality based on user-defined scenarios.

5. **Evaluation and Validation:**
   - Train/test split, cross-validation for ML models.
   - Compare model predictions against real-world outcomes.
   - Conduct usability testing and gather user feedback on system effectiveness.
6. **Documentation and Reporting:**
   - Full system documentation including ML model details, API guides, and user manual.
   - Complete dissertation draft with detailed evaluation and findings.

Research Questions:

- How can real-time mobility and environmental data be effectively integrated into a unified, predictive Digital Twin system?
- What are the measurable benefits of using machine learning models in urban decision-making?
- How can interactive frontend interfaces enhance the usability and effectiveness of Digital Twin platforms?

Methodology:

- Literature Review: Explore existing works on Digital Twins and predictive modelling in smart cities.
- Data Engineering: API pipelines, historical data compilation, and preprocessing.
- Machine Learning Model Development: Implement and evaluate traffic and air quality forecasting models.
- Frontend Development: Build an interactive simulation dashboard with 3D visualization.
- Evaluation: Assess model performance and user satisfaction through systematic testing.

1.3 Legal, Social, Ethical and Professional Considerations

This project involves the use of publicly available datasets, which significantly reduces legal risks. However, responsible data handling and adherence to privacy norms remain critical. Data sourced from Imperial College of London and Open Weather Maps are anonymous and aggregated, ensuring compliance with GDPR and UK Data Protection laws. The system will also implement secure API handling practices and adhere to appropriate licensing frameworks for all open-source tools and libraries used.

From a social perspective, the project aims to promote inclusivity by making advanced urban data analytics and simulation tools accessible to a broad audience, including non-technical stakeholders such as policymakers, planners, and the general public. The frontend interface will be designed in accordance with accessibility standards (e.g., WCAG 2.1) to ensure usability for people with varying abilities and backgrounds. Transparent visualizations of both observed and predicted data will empower citizens and urban administrators to make informed, data-driven decisions.

Ethically, the project commits to maintaining clarity between observed data and Machine Learning-driven predictions, highlighting uncertainties and model limitations where applicable. Attention will be given to minimizing biases in the data and ensuring that model outputs do not inadvertently reinforce socio-economic inequalities or favor particular districts over others. Predictive models will be validated on diverse datasets to promote fairness and accountability.

Professionally, the project will follow best practices in software engineering, data science, and AI ethics. Version control, modular code architecture, unit testing, and detailed documentation will be prioritized to ensure the system's reproducibility and scalability. Ethical guidelines from professional bodies such as the IEEE and ACM will be consulted to ensure integrity throughout the project lifecycle. Stakeholder feedback will be systematically gathered and incorporated into iterative system refinements.

1.4 Background

The growing complexity of urban life, driven by population density, environmental challenges, and infrastructure strain, has necessitated more intelligent systems for city planning and management. In response to this, the concept of a Digital Twin (DT) has gained prominence in urban technology discourse. Initially developed in the manufacturing and aerospace sectors for performance optimization and predictive maintenance, Digital Twin systems are now being adapted for smart city applications, offering cities a powerful tool for data integration, visualization, predictive modelling, and policy simulation.

A Digital Twin refers to a dynamic, virtual representation of a physical entity, continuously updated with real-time data from sensors, historical records, and environmental inputs. In the urban context, a DT can represent roads, buildings, traffic systems, pollution layers, and utility infrastructures, enabling the analysis of what is happening in a city and predicting what might happen under various scenarios. The potential of Digital Twins in city governance has been widely acknowledged in both academia and policy planning, especially for transportation and environmental monitoring—two pressing concerns for metropolitan areas like London.

Evolution of Digital Twin Applications in Smart Cities

The application of Digital Twin technology to smart cities has evolved rapidly in the past decade. Early work focused on static urban models or simulations using historical data. However, the recent advent of ubiquitous IoT devices, open APIs, and powerful visualization engines such as CesiumJS and Unity3D has allowed real-time, interactive Digital Twins to become feasible.

Batty et al. [11] and other urban systems theorists have identified Digital Twins as critical tools for simulating and managing city dynamics. They argue that urban environments are inherently complex systems where changes in one part (e.g., road closures) can cascade to impact other areas (e.g., pollution levels, commute times). A DT, by integrating these components, becomes a platform not only for observation but for experimentation with hypothetical interventions. This paradigm shift—from static analysis to live, actionable simulation—is crucial in managing modern urban problems.

Smart Mobility, Traffic Modelling, and Machine Learning Integration

Traffic congestion remains one of the most visible and detrimental outcomes of poor urban planning. In London, particularly in dense boroughs like Westminster, traffic congestion not only delays commuters and reduces economic productivity but significantly contributes to elevated air pollution levels. According to Open Weather Maps, the average traffic speed in central London during peak hours continues to decline, indicating a worsening problem.

Open Weather Maps provide real-time APIs covering bus locations, tube schedules, incidents, and road status. These datasets are frequently utilized in smart mobility studies to optimize routing, analyse congestion patterns, and estimate journey times. Research published in the *IEEE Transactions on Intelligent Transportation Systems [14]* has demonstrated the effectiveness of using DTs integrated with traffic APIs to model mobility trends and forecast bottlenecks. However, recent developments show that Machine Learning (ML) algorithms, such as XGBoost and Random Forests, significantly enhance predictive capabilities by learning complex temporal patterns in traffic data.

A 2023 study titled *A Digital Urban Twin Enabling Interactive Pollution Predictions and Enhanced Planning* [12] introduced a model that simulated the effects of rerouting buses on NO_2 levels in South London using ML models. Their findings show that integrating ML into DT systems provides more accurate, real-time predictions, enabling dynamic policy interventions.

Environmental Monitoring, Air Quality, and Predictive Modelling

Air pollution is another critical issue for Westminster, where levels of nitrogen dioxide (NO_2) and PM2.5 frequently breach safe thresholds set by the World Health Organization (WHO). According to the Open Weather Maps API, Westminster's busy corridors like Oxford Street and Victoria Embankment consistently record pollution spikes due to vehicular emissions.

Environmental monitoring within a Digital Twin is achieved by integrating real-time pollution sensor data, meteorological data, and geographic context (such as building layouts and traffic intensity). Thomas et al. [13] in *Frontiers in Sustainable Cities* propose a layered architecture where pollution data is processed alongside mobility and weather data to produce real-time

pollution heatmaps. Machine Learning techniques, such as Long Short-Term Memory (LSTM) networks and Random Forest Regression, are increasingly applied to model and forecast pollutant concentration levels dynamically.

An enhanced Digital Twin system integrating ML can predict air quality under different intervention scenarios, such as implementing low-emission zones or adjusting traffic patterns. These forecasts allow policymakers to assess the effectiveness of proposed changes before implementation, supporting proactive, evidence-based urban planning.

Interoperability, Visualization, and Frontend Development

A Digital Twin must combine disparate data streams—each in different formats and frequencies—into a unified system. A modular architecture is often adopted, where data ingestion, storage, analysis, and visualization layers are decoupled but synchronized. For spatial data, OpenStreetMap (OSM) provides road networks and points of interest, while CityGML datasets offer 3D building geometry and semantic information.

Tools such as QGIS and PostGIS are used for geospatial preprocessing, while backend services are built using frameworks like FastAPI. Real-time data from Imperial College of London and Open Weather Maps APIs is stored in time-series databases or accessed via asynchronous REST endpoints. Machine Learning models are developed using scikit-learn, TensorFlow, or XGBoost libraries, enabling real-time traffic and pollution forecasting.

The frontend plays a crucial role in making complex simulations accessible to users. CesiumJS supports WebGL-based rendering of large-scale city models with time-dynamic datasets. Paired with React.js, the frontend can offer an interactive dashboard where users can visualize real-time and predictive traffic and pollution data, modify parameters, and simulate “what-if” scenarios through a user-friendly interface. This democratizes access to advanced simulation tools, enabling better public engagement and transparent decision-making.

Challenges and Limitations

Despite the promise, Digital Twin implementation with Machine Learning faces several challenges:

1. **Data Granularity**: Pollution sensors and traffic data are often sparse, requiring sophisticated interpolation and data augmentation techniques.
2. **Computational Cost**: Training ML models on large datasets and rendering real-time 3D visualizations demand high computational resources.
3. **Interoperability Issues**: Combining geospatial, time-series, and predictive data requires rigorous standardization and validation.
4. **Stakeholder Engagement**: Ensuring the platform is usable and accessible to non-technical stakeholders requires careful frontend design and user education.
5. **Model Generalization**: ML models trained on historical data must generalize well to future, unseen scenarios, which can be challenging in highly dynamic urban environments.

Addressing these limitations is an active area of research. Successful prototypes typically adopt incremental deployment strategies, starting with smaller-scale implementations and gradually scaling up.

Gap in Existing Research

Although several studies focus on Digital Twins for traffic management or environmental monitoring individually, few attempt to integrate both domains into a cohesive, real-time system enhanced by Machine Learning. Even fewer projects emphasize the development of an interactive, user-friendly frontend capable of real-time scenario simulation.

Moreover, most existing systems focus on retrospective simulations or historical data analysis rather than real-time predictive capabilities. The proposed dissertation project aims to bridge these gaps by building a live, interactive Digital Twin for Westminster that combines real-time traffic and pollution data, Machine Learning-based forecasting, and 3D visualization.

Contribution and Innovation

The proposed project will contribute both academically and practically by demonstrating:

- The technical feasibility of integrating real-time transport and air quality data into a Machine Learning-enhanced Digital Twin.
- The design and deployment of a modular architecture that supports scalable data ingestion, storage, ML modelling, and 3D visualization.
- A user-facing interface that supports scenario simulation for real-world policy exploration (e.g., traffic rerouting or green zone planning).
- Insights into the relationship between transport interventions and environmental outcomes, supported by predictive analytics.

These contributions are expected to be highly relevant for smart city researchers, urban planners, transport authorities, and environmental agencies seeking to leverage AI and Digital Twin technology for sustainable urban management.

1.5 Structure of Report

This report is structured into five comprehensive chapters to document the research, implementation, and evaluation of a Machine Learning-enhanced Digital Twin for Westminster, London.

- **Chapter 1 – Introduction** outlines the motivation, problem statement, and objectives of the project. It also addresses legal, ethical, and professional considerations and sets the context for the study.
- **Chapter 2 – Literature and Technology Review** presents an in-depth review of relevant academic studies and technical tools. It evaluates various machine learning approaches for traffic and air quality forecasting and justifies the selection of A-LSTM and XGBoost models, along with supporting technologies like FastAPI, Kafka, and CesiumJS.
- **Chapter 3 – Implementation** describes the system architecture and the design and development of backend services, machine learning models, data pipelines, and the interactive frontend. It also details database schemas, containerization strategy, and deployment procedures.

- **Chapter 4 – Evaluation and Results** provides a comprehensive evaluation of model accuracy, system functionality, and scenario simulations. It includes performance metrics, visual analytics, and user feedback to validate the system's effectiveness.
- **Chapter 5 – Conclusion** summarises the project's outcomes, discusses limitations, suggests directions for future work, and reflects on lessons learned throughout the development process.

This structure ensures a logical flow from theoretical foundation to practical implementation and critical evaluation.

Chapter 2- Literature & Technology Review

This chapter aims to provide understanding gained from literatures regarding relation between AQI and pollutant concentration with Traffic flow and density, using suitable models for generating synthetic data for traffic, models for forecasting and various frameworks and packages that aid in delivering the project.

2.1 Literature Review

Urban air pollution and traffic congestion represent significant challenges for sustainable development. With increasing urbanization and vehicular growth, real-time monitoring and forecasting have become crucial for effective environmental and traffic management. Traditional statistical techniques, while foundational, often struggle to capture the complex, non-linear, and dynamic patterns inherent in environmental and traffic datasets. Consequently, the integration of machine learning (ML) models has emerged as a viable solution.

2.1.1 Machine Learning in Traffic and Air Quality Forecasting

The utility of ML in traffic congestion and air pollution prediction has been widely explored. Various algorithms, such as Random Forests (RF), Support Vector Machines (SVM), Decision Trees (DT), and Neural Networks (NN), have shown significant predictive capabilities. In [4], the authors stress that while traditional models like autoregressive integrated moving average (ARIMA) offer baseline forecasting capabilities, they fall short in capturing intricate temporal dependencies in real-world datasets.

Several studies underscore the suitability of ensemble methods and deep learning models in these domains. For instance, [5] highlights the superior accuracy of ML models like XGBoost and LSTM in air quality forecasting, especially for pollutants such as PM2.5, NO_2, and CO. XGBoost's ability to handle missing data, multicollinearity, and feature interactions makes it particularly effective in modelling real-world pollutant data. Similarly, [7] supports the use of regression-based and tree-based models for predicting various air quality indicators, citing improved accuracy and resilience in diverse urban conditions.

2.1.2 Temporal and Spatial Correlations

Temporal and spatial dependencies are critical to understanding and forecasting environmental conditions. As pointed out in [9], AQI prediction benefits substantially from models that can incorporate both spatial and temporal features. The fusion of spatial correlation forecasting with temporal correlation forecasting improves overall model robustness and predictive precision. For instance, air pollution readings from neighbouring stations can be valuable predictors when spatial relationships are properly encoded.

LSTM networks are particularly adept at handling long-range temporal dependencies, making them suitable for predicting time series data in both traffic and air quality domains. In [6], deep learning approaches such as LSTM and GRU have demonstrated superior performance over classical models in forecasting future traffic flow based on historical sequences. These architectures retain memory of past observations, a crucial property when dealing with congestion patterns that exhibit diurnal and weekly cycles.

2.1.3 Synthetic Data Generation with LSTM

Due to limited labelled datasets in many urban contexts, synthetic data generation has become an essential tool. Traditional LSTM-based models have shown potential in capturing time series structures and generating traffic flow and density metrics. However, recent advancements highlight the superiority of Adaptive LSTM (A-LSTM), which incorporates multi-layer configurations, refined memory gating mechanisms, and optimized hyperparameters. Raheja and Malik [10] demonstrated that A-LSTM outperforms conventional LSTM, SVR, and hybrid SVR-LSTM models in forecasting pollutants such as PM2.5, PM10, and NO_2. A-LSTM achieved lower MAE and RMSE values and outperformed on precision, recall, and F1-score metrics. Its enhanced ability to retain long-term dependencies makes it particularly effective for generating synthetic traffic data in scenarios with sparse sensor coverage or incomplete data streams.

2.1.4 Ensemble and Hybrid Methods

The ensemble paradigm combines multiple weak learners to form a strong predictive model. In [8], a bagging ensemble approach was used to combine various base learners, improving both traffic and air quality predictions. Hybrid schemes that utilize ML models in conjunction with preprocessing techniques like Principal Component Analysis (PCA), or that combine different predictors, have also demonstrated increased robustness.

XGBoost, a gradient-boosting ensemble algorithm, has been particularly effective in forecasting AQI values. As stated in [5], its capability to handle missing data and nonlinear relationships enables it to outperform traditional regression models in noisy urban environments. In the context of this study, XGBoost was used to forecast multiple pollutants simultaneously, using predictors including weather conditions and synthetic traffic metrics.

2.1.5 Feature Importance and Interpretability

Interpreting ML model outputs is essential for building trust and ensuring applicability in policy and planning. XGBoost and other tree-based models inherently support feature importance analysis. Studies like [7] reveal that weather variables (temperature, humidity, and wind) and traffic conditions are major contributors to AQI predictions. Being able to identify key predictors can help urban planners and environmental agencies prioritize interventions.

The use of interpretable models also facilitates the transition of research prototypes into practical tools for governmental and industrial stakeholders. Feature importance metrics derived from XGBoost, for instance, can help identify high-impact areas and optimize sensor placements.

2.1.6 Suitability of A-LSTM and XGBoost

Combining A-LSTM and XGBoost creates a powerful forecasting framework. A-LSTM is well-suited for temporal modelling, especially for generating synthetic traffic metrics where temporal coherence is necessary. On the other hand, XGBoost efficiently models multivariate

pollutant data using a wide range of predictors, including weather, traffic, and historical AQI data.

This bifurcated use — A-LSTM for synthetic data generation and XGBoost for AQI forecasting — aligns with the strengths of each algorithm, as supported by multiple studies [5][6][8][9][10].

2.2 Technology Review

Creating a real-time ML-powered digital twin for Westminster involves integrating technologies across data ingestion, modelling, and visualization. Our selected tools ensure responsiveness, scalability, and alignment with recent literature.

2.2.1 FastAPI (Python)

FastAPI is used for our backend API due to its high performance, async support, and seamless integration with Python ML libraries. It hosts endpoints for real-time data queries, forecasting, and scenario simulations. FastAPI's interactive documentation and lightweight architecture make it ideal for rapid development and external integration.

2.2.2 Data Ingestion and Streaming

Real-time data is streamed from Imperial College of London and OpenWeatherMap. Apache Kafka buffers these inputs, and Python workers process and push them to Redis, serving as a short-term feature store. Redis holds synchronized, recent data snapshots for inference. Historical data is stored in PostgreSQL for training. This hybrid approach combines fast, real-time streaming with reliable archival storage and supports time-aligned feature vectors—critical for predictive accuracy [6].

2.2.3 Machine Learning Models (A-LSTM, XGBoost)

Our twin uses XGBoost and A-LSTM models. XGBoost forecasts AQI and traffic based on structured inputs, offering interpretability and low latency. A-LSTM, designed for sequential data, is used for generating synthetic traffic metrics where temporal coherence is key. Models

are trained on historical data and served via FastAPI for real-time inference. A scheduler monitors model performance and triggers automated retraining, ensuring adaptability to evolving conditions—an approach supported in [4] and [5].

2.2.4 CesiumJS and 3D Visualization

CesiumJS powers the 3D front-end, enabling an immersive map-based interface of Westminster. Users can visualize real-time traffic and air quality, simulate interventions (e.g., road closures), and view forecast outcomes. REST APIs and Web Sockets deliver data and updates. CesiumJS supports timeline animations, helping users replay past events or preview future scenarios. Layer toggles allow multi-dimensional insight, enhancing usability for non-technical stakeholders [3].

2.2.5 Supporting Tools and Integration (Out of scope)

All components run in Docker containers for consistent deployment. Monitoring tools like Prometheus and Grafana track system health. GDPR-compliant data handling ensures privacy through pseudonymization and consent management. Our architecture follows a modular microservices design, allowing easy integration of future analytics or data sources, in line with smart city system recommendations [4].

2.3 Review Summary

The literature and technology review confirms the feasibility of a machine learning-enhanced digital twin for urban planning. Foundational studies [1][3] highlight the value of real-time monitoring and virtual simulations, while research on forecasting [4][5][6][7][8] supports the use of models like XGBoost and LSTM.

System architecture for the proposed solution should reflects these insights, integrating real-time data ingestion, predictive modelling, and 3D visualization into one cohesive platform. FastAPI enables efficient communication, while CesiumJS provides intuitive exploration. Notably, the use of A-LSTM for synthetic traffic data generation from pollutant inputs—guided by studies [8][9][10]—adds robustness in data-sparse scenarios. Overall, this project

translates academic research into a practical tool for managing Westminster's traffic and air quality with foresight and precision.

Chapter 3- Implementation

This chapter explains the implementation strategy through Design, Develop and Deploy phase for delivering the project.

3.1 Design

During this phase the application design is represented in the form of Architecture Design which showcases the components and the interaction between them. Also ERD (Entity Relationship Diagram is shown and explained to showcase various tables that are used for persistent storage of data and relations among them). A use case diagram is shown towards the end of design phase section, and explained for understanding the scenarios from the usability perspective.

3.1.1 Architecture Design

The architecture of the proposed Digital Twin for Westminster is designed to support real-time, intelligent forecasting of urban traffic and air quality through seamless integration of data ingestion, machine learning, and 3D visualization components. It leverages modular, scalable technologies to ensure flexibility, maintainability, and responsiveness.

The components involved are:

DB

- PosgressSQL DB: Opensource and scalable DB engine.

MiddleWare

- Consists of data ingestion pipelines to stream data from source APIs to Kafka topics using producer scripts and Kafka Topics to DB for storage using consumer scripts
- Also responsible for retraining of models for forecast of AQI and pollutants and producing synthetic data for traffic using XGboost
- The middleware also has REST APIs which servers the request from the fronend by providing historical data, forecast data, and dashboard data.

Presentation Layer:

- Responsible for taking user input and presenting result of the query on a user interface.

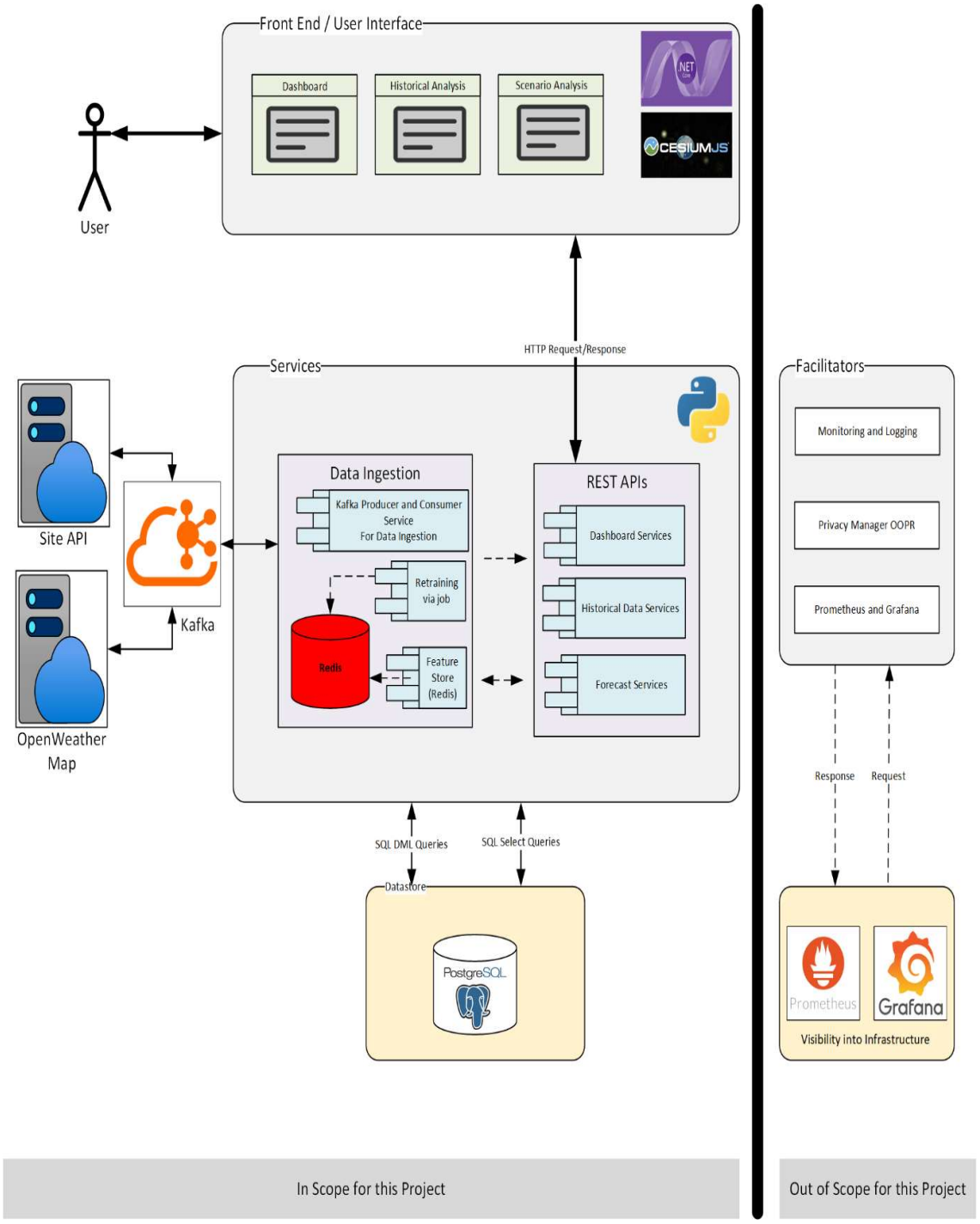


Figure 3.1.1 : System Architecture

3.1.1(a) Data Sources and Ingestion

The system integrates data from multiple live sources: Imperial College of London for site coordinates of council, OpenWeatherMap for weather conditions, and pollution data from air quality monitors. These sources feed data into the system via **Apache Kafka**, which acts as a message broker, enabling decoupled, asynchronous communication between producers (data sources) and consumers (processing services). This setup supports high-throughput, real-time data handling essential for timely forecasting and analytics.

A set of **Kafka consumers** processes the incoming data, extracting and transforming it before storing it in two separate layers:

- **Redis**, which functions storage for predictors and targets to maintain sync while retraining of the model.
- **PostgreSQL**, which stores historical data used for training and evaluation of ML models and storing metrics for showing on dashboard.

3.1.1(b) Machine Learning and Prediction

The architecture incorporates a **retraining pipeline** that periodically updates model parameters based on recent trends. It uses **LSTM (Long Short-Term Memory)** models to synthetically generate traffic flow data based on pollution indicators and **XGBoost** for AQI forecasting based on structured, multivariate features. These models are trained offline using data pulled from PostgreSQL and are deployed as serialized artifacts ready for inference.

A-LSTM Model for generating synthetic data for traffic [Figure 3.1.1(b)(i)]

Input Sequence

- Shape: **(24, 17)**
- Description: 24 time steps of features per sample, including AQI, weather, location, and temporal features.

StandardScaler

- Transformation: **Z-score normalization**

- Purpose: Normalize all features to zero mean and unit variance for stable training.

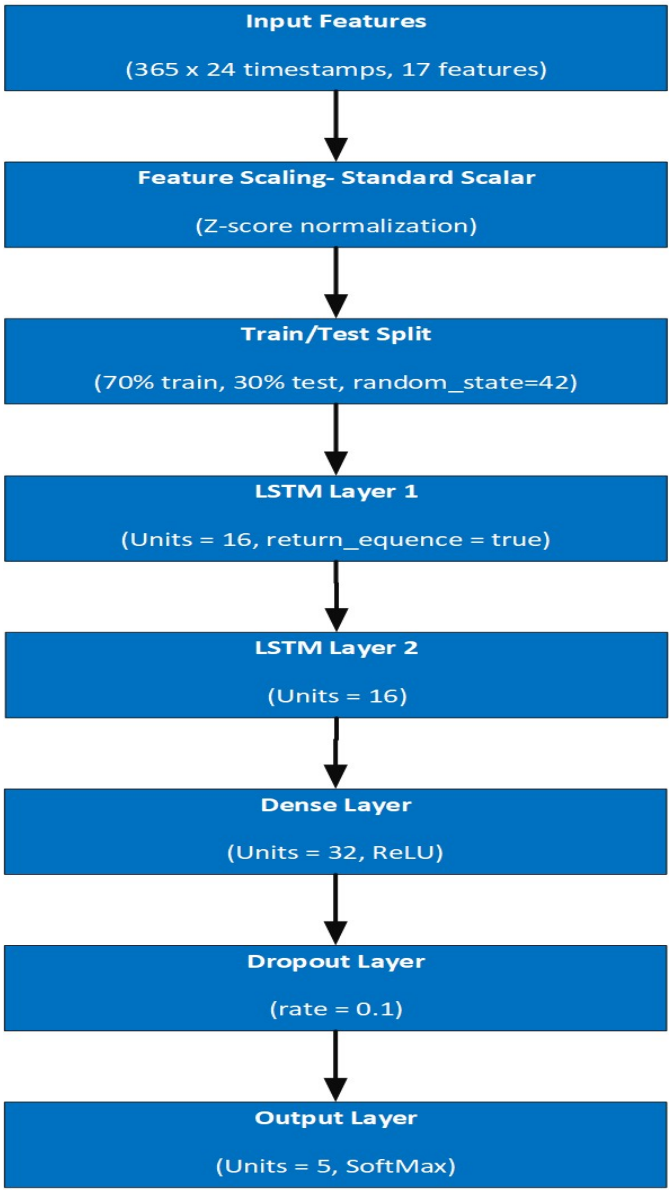


Figure 3.1.1(b)(i) A-LSTM Model Architecture

Test/Train Split

- Strategy: Random split using train_test_split.
- Training set: 70% of data used to fit the XGBoost model.
- Testing set: 30% held out to evaluate performance.
- Ensures generalizability and guards against overfitting.

LSTM Layer 1

- Units: **16**
- Configuration: return_sequences=True
- Purpose: Capture sequential dependencies across time steps and pass full sequence to next LSTM.

LSTM Layer 2

- Units: **16**
- Configuration: Returns only the final hidden state
- Purpose: Summarize temporal information from the sequence into a fixed vector.

Dense Layer

- Units: **32**, Activation: **ReLU**
- Purpose: Introduce non-linearity and learn high-level features from LSTM output.

Dropout Layer

- Rate: **0.1**
- Purpose: Regularization to prevent overfitting.

Output Layer

- Units: **5**, Activation: **Softmax**
- Purpose: Multi-class classification of traffic flow into one of five categories.

Class Label Output

- Categories: low, moderate_low, moderate, moderate_high, high
- Output: Most likely traffic flow class.

XGBoost for forecasting AQI based on weather and traffic data [figure 3.1.1(b)(ii)]

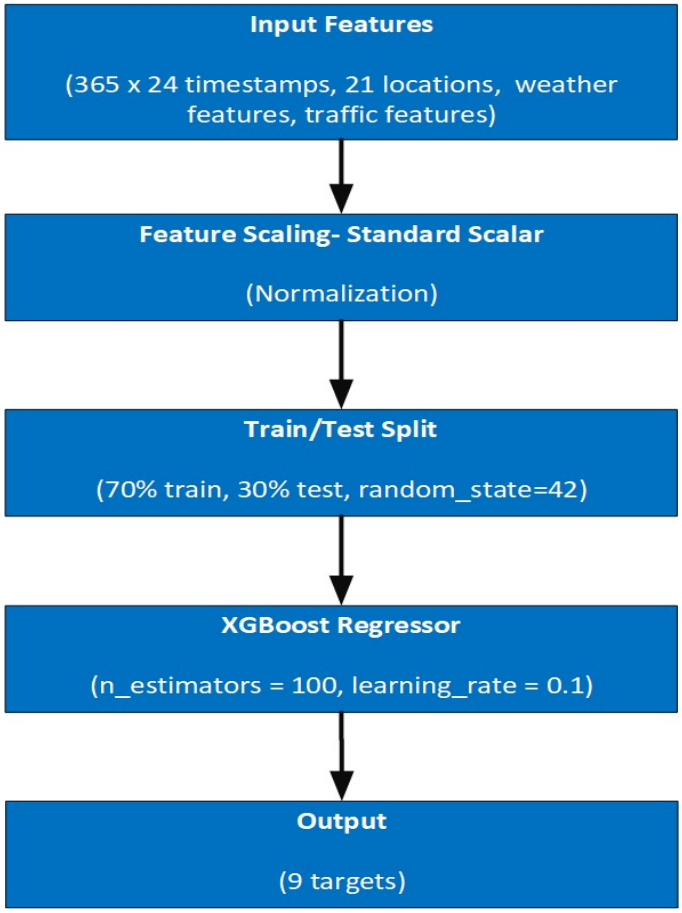


Figure 3.1.1(b)(ii) :XGBoost Model Architecture

Raw Input Features

- Source: Preprocessed dataset after feature engineering.
- Features include AQI components (pm2_5, pm10, no2, so2, co, o3), weather metrics (temp, pressure, humidity, etc.), and time/location fields.
- Each target variable (e.g., pollutants) is modeled **independently** using its respective features.

Feature Scaling (StandardScaler)

- Method: **Z-score normalization** using StandardScaler.

- Purpose: Ensures features contribute equally by transforming them to have zero mean and unit variance.
- Preprocessing happens **before training**, and the same scaler is used during inference.

Train/Test Split (70/30)

- Strategy: Random split using train_test_split.
- Training set: 70% of data used to fit the XGBoost model.
- Testing set: 30% held out to evaluate performance.
- Ensures generalizability and guards against overfitting.

XGBoost Regressor

a) Framework: XGBRegressor from the xgboost library.

b) Hyperparameters:

   - n_estimators=100 – number of boosting rounds.
   - learning_rate=0.1 – controls contribution of each tree.
   - random_state=42 – for reproducibility.

c) One model is trained **per target variable**.

d) Handles missing values and captures non-linear feature interactions.

Predicted Outputs

- Model predicts **continuous values** for target features (e.g., pollutant levels).
- Predictions are stored in a dataframe indexed by test samples.
- Each model saves results for future scoring and evaluation.

Model inference is triggered by API requests to forecast future values of traffic or pollutant levels, which are served by a REST API layer built using **FastAPI**. FastAPI enables rapid and

scalable deployment of endpoints for prediction, live queries, and hypothetical scenario simulations.

3.1.1(c) REST APIs and Service Layer

The REST API layer exposes three core services:

- **Dashboard Services**, providing summary views and visual performance trends.
- **Historical Data Services**, which allow querying past metrics for comparison.
- **Forecast Services**, enabling users to submit queries or hypothetical changes (e.g., road closures) and receive predictions.

These APIs interact with the feature store and trained models to compute outputs, returning results to the front end in real time.

3.1.1(d) Front-End Visualization

The front end is built using a .NET core MVC framework and **CesiumJS** for 3D geospatial visualization. The dashboard provides real-time overlays of traffic and air quality, historical playback, and scenario-based forecasting using RESTful data. Users can interactively simulate urban interventions and observe their effects spatially and temporally.

3.1.1(e) Database Design – ERD

Entity Relationship Diagram (ERD) shown in figure 1.2 shows the tables and relationships between them along with constraints like primary key, foreign key, unique constraint and datatypes of columns of each table. This presents the foundation for the data that will be ingested by building a pipeline.

(a) Data Ingestion Tables

This stores the data from respective sources in the appropriate tables.

site_table

- **Primary Key:** (latitude, longitude) — marked with **violet** key icons.
- Stores data about monitoring sites: site_code, site_name, site_type, etc.

- **Referenced by:** aqi_table, traffic_table, weather_table.

aqi_table

- Contains air quality metrics (aqi, co, no2, etc.).
- **Foreign Keys:** latitude, longitude referencing site_table.
- **Also includes:** site_code, measurement_datetime.

traffic_table

- Records traffic statistics like traffic_flow and traffic_density.
- **Foreign Keys:** latitude, longitude referencing site_table.
- Includes measurement_datetime.

weather_table

- Holds weather readings like temp, humidity, uvi, etc.
- **Foreign Keys:** latitude, longitude referencing site_table.

(b) Model performance data

This saves log of model performance metrics everytime when it is run.

synthetic_lstm_stats

- Tracks LSTM model performance: model_name, test_loss, test_accuracy, created_at.
- **Primary Key:** id.

regressor_stats

- Stores regression model evaluation: rmse, mae, r2, created_at.
- **Primary Key:** id.

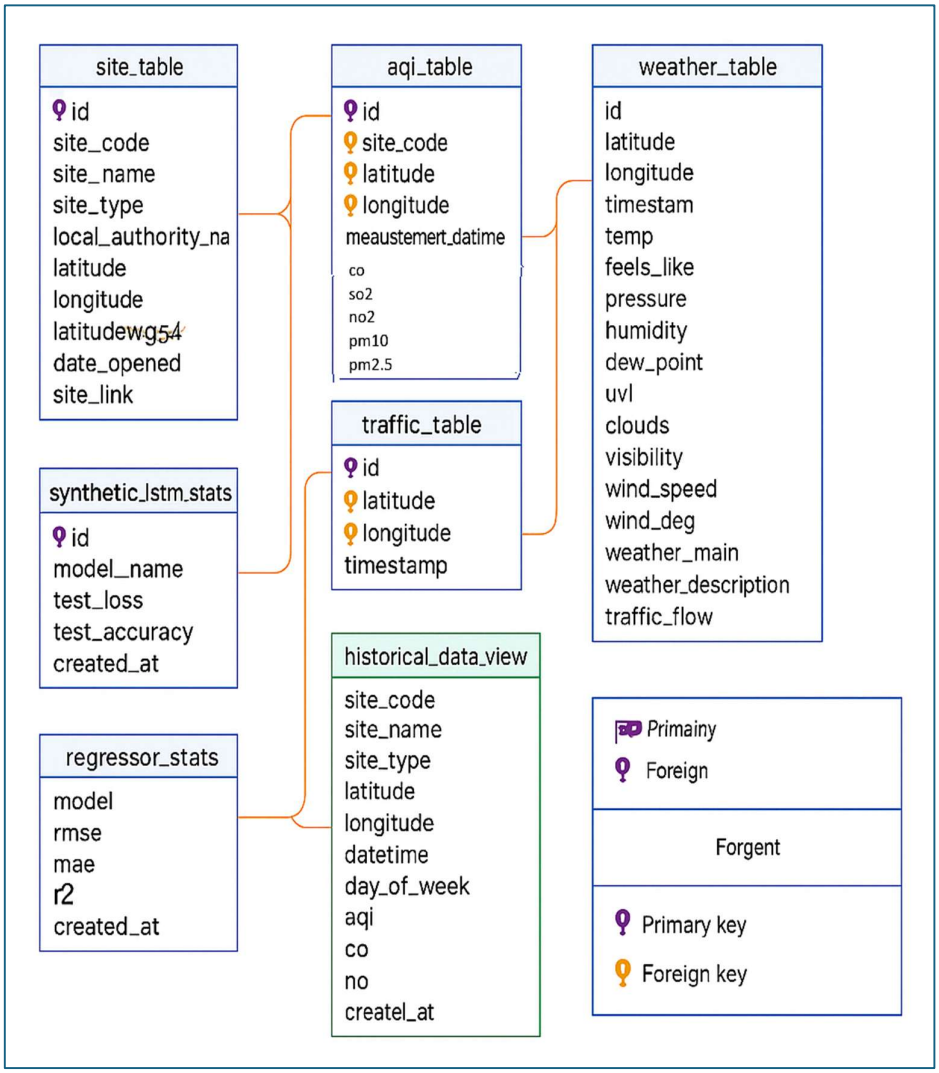


Figure 3.1.7: ERD – Entity Relationship Diagram

3.1.1(f) Supporting Infrastructure (Suggested in design but out of scope for the project)

Supporting components include **Docker** for containerization, **Prometheus and Grafana** for performance monitoring, and a **privacy module** aligned with GDPR. The architecture follows a modular microservices approach, where components interact via well-defined interfaces, promoting extensibility and independent deployment.

In summary, the architecture provides a robust foundation for a smart, adaptive digital twin capable of real-time forecasting and policy testing for Westminster's urban environment.

3.1.2 Use Case Diagram

Use case describes the actions that user will perform to interact with the system. There are 3 main scenarios for the project that are shown in the below diagram (figure 1.2) and description after.

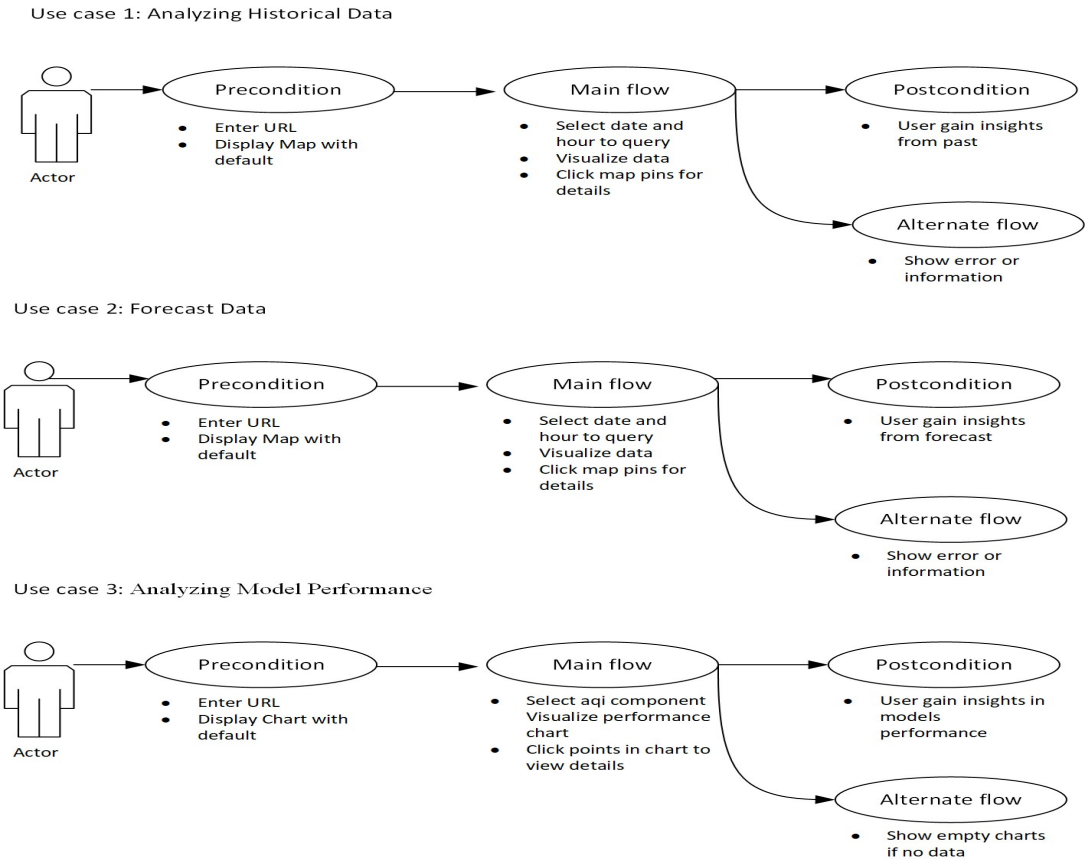


Figure 3.1.2 Use Case Diagram

Use Case -1: Analyzing Historical Data

- Actor(s): End User (e.g., city planner, analyst)
- Goal: View and analyze environmental and traffic data date and hour of the that date.

Preconditions:

- User enters the application URL navigates to the analytics page clicking Analytics Navigation Button.
- historical_data_view is populated on map and interacted.

Main Flow:

- Selects date or hour of the target date.
- System queries historical_data_view to retrieve matching records.
- Data is visualized on map (e.g., AQI trends, temperature vs traffic).
- User optionally can click on each entity on map to see details.

Alternative Flows:

- If no data is available for the selected range or the date range is not from past, show an appropriate message.

Postconditions:

- User gains insights from combined AQI, weather, and traffic data or individual views.

Use Case -2: Forecasting Data

- Actor(s): End User (e.g., city planner, analyst)
- Goal: View and analyze environmental and traffic data date and hour of the that date.

Preconditions:

- User enters the application URL and navigates to the forecast page clicking Forecast Navigation Button.
- forecast data is generated and is populated on map and interacted.

Main Flow:

- Selects date or hour of the target date and hour.
- System queries apis for forecast data for target date and hour, to retrieve matching records.
- Data is visualized on map (e.g., AQI trends, temperature vs traffic).
- User optionally can click on each entity on map to see details.

Alternative Flows:

If no data is available for the selected range or the date range is not current or future, show an appropriate message.

Postconditions:

User gains insights from combined AQI, weather, and traffic data or individual views.

Use Case -3: Analyzing Model Performance

- Actor(s): End User (e.g., analyst)
- Goal: View and analyze model performance that are used for synthetic data generation for traffic and forecasting aqi.

Preconditions:

- User enters the application URL
- Default dashboard is populated as chart and interacted.

Main Flow:

- User navigates to the dashboard page clicking Dashboard Navigation Button.
- Selects, component from AQI or pollutant to see the line chart.
- Data is visualized on charts as line graph.
- Users optionally can click on each point on chart to see the details.

Alternative Flows:

- If no data is available for the selected model show empty chart.

Postconditions:

User, gains insights of the models used for forecasting and traffic data generation.

3.2 Develop

The development of the solution highlights the code or scripts used in development of the components described in the design phase in the previous section [3.1 Design]. Components like Database creation, Kafka for streaming data real-time and Redis server for storing feature columns for maintaining consistency between model retraining is addressed first in this section. The detail development for the database (tables and views), data pipeline (containing producer and consumer for relevant topics and retraining of the model post data ingestion), API (for middleware) and front-end (for presentation logic) will also be addressed to provide the clear understanding of the code structure to develop the solution.

3.2.1 Containers with required services

This section explains how the required services on which the application depends is implemented using docker containers.

3.2.1(a) Realtime Ingestion Service

This component is required to ingest data in real-time from the respective sources. Kafka is built on publisher and subscriber design pattern where the publisher is responsible for extracting data from the sources and sending it to the designated Kafka topic. The respective subscribers listen to the particular que or topic on kafka for any data available and reads and saves in the database. The data pipeline code will be further explained in section [3.2.5].

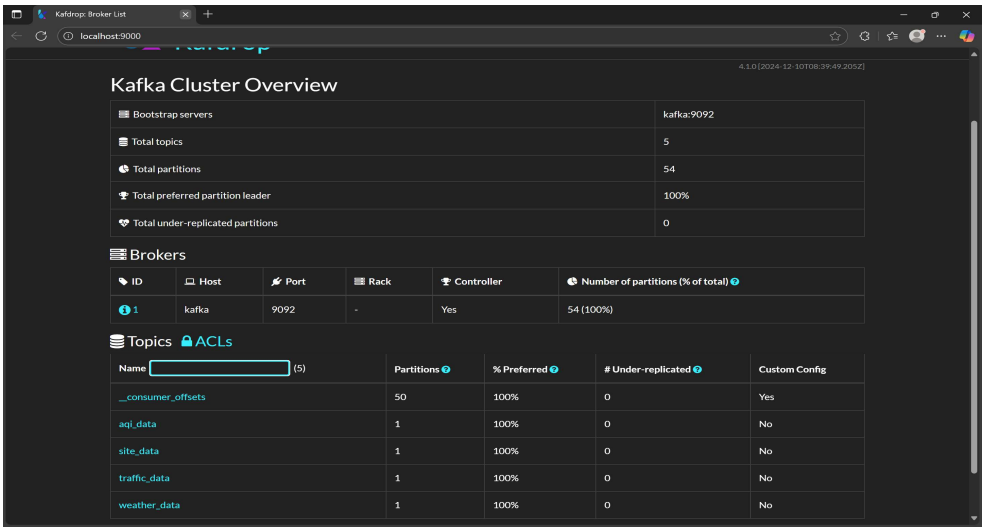


Figure 3.2.1.1: Kafdrop/Kafka with topics for streaming

Kafka (kafka service)

- **Image**: apache/kafka:latest
- **Container Name**: kafka
- **Ports**: 9092:9092
- **Environment Variables**:
    - Custom Kafka KRaft mode setup (no Zookeeper)
    - Key variables include:
        - KAFKA_NODE_ID=1
        - KAFKA_PROCESS_ROLES=broker,controller
        - KAFKA_LISTENERS, KAFKA_ADVERTISED_LISTENERS
        - KAFKA_CONTROLLER_LISTENER_NAMES=CONTROLLER
        - CLUSTER_ID=MkU3OEVBNTcwNTJENDM2Qk
        - KAFKA_AUTO_CREATE_TOPICS_ENABLE=true

Kafdrop (kafdrop service)

- **Image**: obsidiandynamics/kafdrop
- **Container Name**: kafdrop
- **Ports**: 9000:9000
- **Environment**:
    - KAFKA_BROKERCONNECT=kafka:9092
- **Depends On**: kafka

Data Producer & Consumer

- **Producer (main_producer)**
    - **Build**: ./ingestion_service

- **Command**: python -u main_producer.py
- **Depends On**: db, kafka, redis, main_consumer
- **Environment**:
  - Kafka, PostgreSQL, Redis, API service and OpenWeather API key

- **Consumer (main_consumer)**
  - **Build**: ./ingestion_service
  - **Command**: python -u main_consumer.py
  - **Depends On**: kafka, db
  - **Environment**: Kafka + PostgreSQL connection details

3.2.1(b) Feature Cache

Feature cache is key in maintaining consistency between the models generated and the application by writing predictor and target features into the Redis cache. Redis is normally used as web server cache for faster retrieval of the data. The cache can be saved in json/binary format. For this solution binary format is selected because of faster storage and retrieval of the information as it does not requires data to be serialized before saving in the cache.

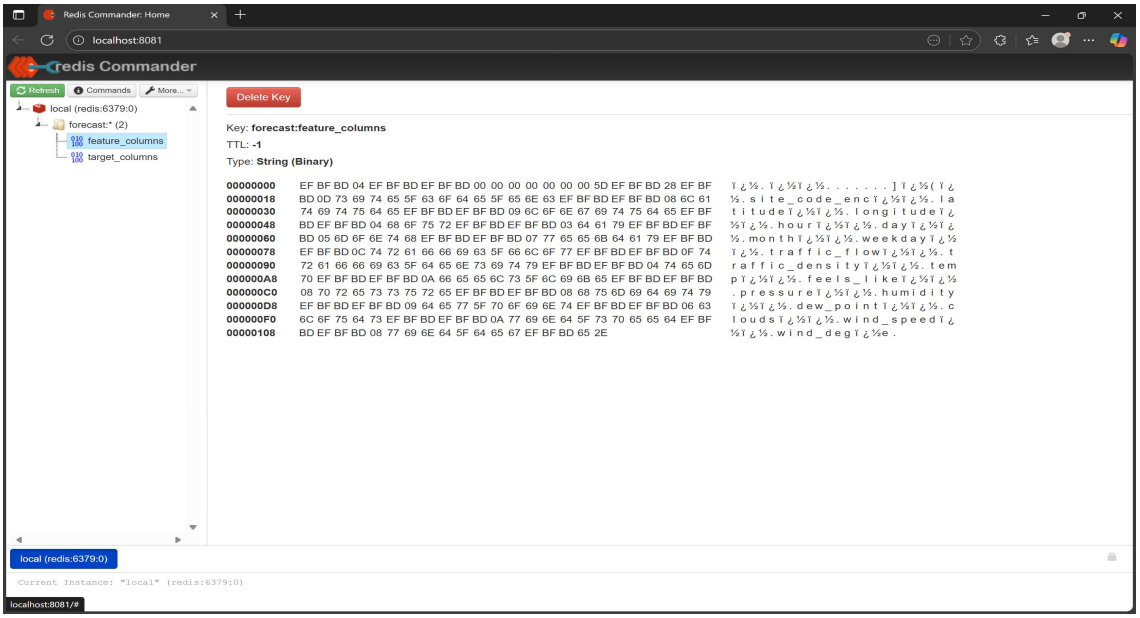


Figure 3.2.1.2(a) Redis store for Feature columns

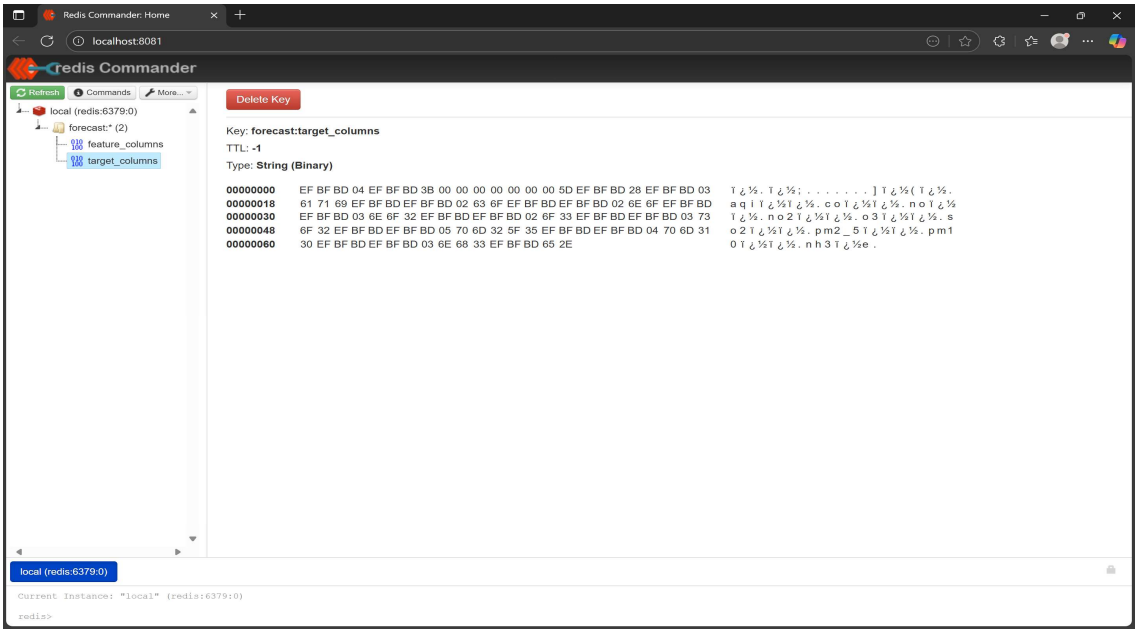


Figure 3.2.1.2(b) Redis store for Target Columns

Redis (redis service)

- **Image**: redis:latest
- **Container Name**: my-redis-server
- **Ports**: 6379:6379
- **Volumes**:
    - redis_data:/data
- **Environment**:
    - REDIS_PASSWORD=Admin123

Redis Commander (redis-commander service)

- **Image**: rediscommander/redis-commander:latest
- **Ports**: 8081:8081
- **Depends On**: redis

- **Environment**:
    - REDIS_HOSTS=local:redis:6379
    - HTTP_USER=admin
    - HTTP_PASSWORD=Admin123

3.2.2 IDE for development

Visual Studio Code (VS Code) is used for coding and debugging. It is a lightweight, cross-platform, and highly extensible IDE, making it an ideal choice for full-stack projects that combine Python-based APIs with ASP.NET Core MVC frontends and JavaScript libraries.

Unified Environment for Backend and Frontend

- **Python Backend**:
    - Install the **Python extension** for syntax highlighting, IntelliSense, debugging, and environment management.
    - Use built-in terminal for running FastAPI or other Python services.
- **ASP.NET Core MVC Frontend**:
    - With the **C# extension (by OmniSharp)**, VS Code provides:
        - IntelliSense, debugging, and Razor syntax support.
        - Project scaffolding, code navigation, and build integration with the .NET CLI.
    - Debug support for both server-side (C#) and client-side (JavaScript/TypeScript).

Productivity Features

- **IntelliSense**, **code navigation**, and **refactoring tools** speed up development.
- **Git integration** for seamless version control.
- **Docker extension** allows container management from within the IDE.

- Multi-root workspaces allow API, frontend, and shared code to be managed together.

Cross-Platform and Lightweight

- Works equally well on **Windows**, **Linux**, and **macOS**.
- Suitable for collaborative environments and containerized development (e.g., Docker + DevContainers).

In summary, **VS Code** provides a cohesive development experience that bridges both the backend (Python) and frontend (.NET Core MVC + JavaScript), making it a powerful and efficient IDE for full-stack web applications.

3.2.3 Repository and version control

Repository and version control systems track and manage changes to code, enabling collaboration, history tracking, and rollback across development teams.

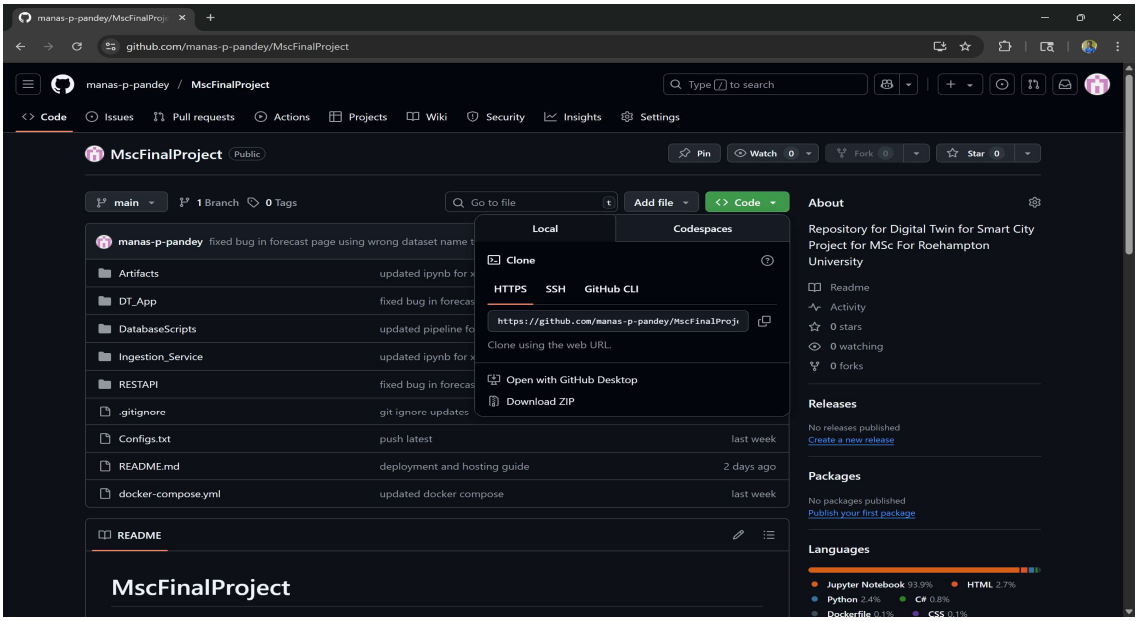


Figure 3.2.3 : Github Repository

What is GitHub?

GitHub is a cloud-based platform for version control using Git. It enables collaborative development by hosting repositories where code changes are tracked, reviewed, and managed.

Setting Up a Repository on GitHub

1. Go to https://github.com and register an account or login with credentials if already have an account.
2. Once logged in, click on **New Repository**.
3. Enter repository **name**, optional **description**, and choose:
    - **Public** (anyone can see) or **Private** (restricted access). For the solution, a public repository is created at

      "https://github.com/manas-p-pandey/MscFinalProject"
    - Optionally **initialize with a README**
4. Click **Create Repository**.
5. The repository name used for this project is "MscFinalProject".

Cloning Repository in VS Code (Checkout Process)

1. Open **Visual Studio Code**.
2. Open the **Command Palette** (Ctrl+Shift+P), then type and select Git: Clone.
3. Paste the **GitHub repository URL**

   (https://github.com/manas-p-pandey/MscFinalProject.git).
4. Choose a local folder to save the repo.
5. VS Code will prompt: *"Would you like to open the cloned repository?"* – click **Yes**.

Basic Git Workflow in VS Code

- **Make changes** to your files.
- **Stage Changes**: Click **Source Control** icon → + next to files.
- **Commit**: Write a message and click the **Commit** button.
- **Push to GitHub**: Click the **...** menu → Push or use terminal:

3.2.4 Creating Tables in Database

Once the database setup is done, we can now create the tables for entities used in this project for persistent storage. The data saved in the tables created will be used for displaying analytics for historical data as well as provide data into the model training pipeline. Sample data from each of these tables are show in Appendix

Below are the list of tables/views with their create scripts:

- **site_table**

```
CREATE TABLE IF NOT EXISTS public.site_table
(
    site_code text COLLATE pg_catalog."default" NOT NULL,
    site_name text COLLATE pg_catalog."default",
    site_type text COLLATE pg_catalog."default",
    local_authority_name text COLLATE pg_catalog."default",
    latitude double precision,
    longitude double precision,
    latitudewgs84 double precision,
    longitudewgs84 double precision,
    date_opened timestamp without time zone,
    date_closed timestamp without time zone,
    site_link text COLLATE pg_catalog."default",
    CONSTRAINT site_table_pkey PRIMARY KEY (site_code)
)

TABLESPACE pg_default;
```

- **aqi_table**

```
CREATE TABLE IF NOT EXISTS public.aqi_table
(
    id integer NOT NULL DEFAULT nextval('aqi_table_id_seq'::regclass),
    site_code text COLLATE pg_catalog."default",
    latitude double precision,
    longitude double precision,
    measurement_datetime timestamp without time zone NOT NULL,
    aqi integer,
    co double precision,
    no double precision,
    no2 double precision,
    o3 double precision,
    so2 double precision,
    pm2_5 double precision,
    pm10 double precision,
    nh3 double precision,
    CONSTRAINT aqi_table_pkey PRIMARY KEY (id),
    CONSTRAINT aqi_table_site_code_measurement_datetime_key
        UNIQUE (site_code, measurement_datetime)
)

TABLESPACE pg_default;
```

- **traffic_table**

```
CREATE TABLE IF NOT EXISTS public.traffic_table
(
    id integer NOT NULL DEFAULT nextval('traffic_table_id_seq'::regclass),
    measurement_datetime timestamp without time zone,
    latitude double precision,
    longitude double precision,
    traffic_flow text COLLATE pg_catalog."default",
    traffic_density text COLLATE pg_catalog."default",
    CONSTRAINT traffic_table_pkey PRIMARY KEY (id),
    CONSTRAINT traffic_table_measurement_datetime_latitude_longitude_key
        UNIQUE (measurement_datetime, latitude, longitude)
)

TABLESPACE pg_default;
```

- **weather_table**

```
CREATE TABLE IF NOT EXISTS public.weather_table
(
    id integer NOT NULL DEFAULT nextval('weather_table_id_seq'::regclass),
    latitude double precision,
    longitude double precision,
    "timestamp" timestamp with time zone,
    temp double precision,
    feels_like double precision,
    pressure integer,
    humidity integer,
    dew_point double precision,
    uvi double precision,
    clouds integer,
    visibility integer,
    wind_speed double precision,
    wind_deg integer,
    weather_main text COLLATE pg_catalog."default",
    weather_description text COLLATE pg_catalog."default",
    CONSTRAINT weather_table_pkey PRIMARY KEY (id),
    CONSTRAINT weather_table_latitude_longitude_timestamp_key
        UNIQUE (latitude, longitude, "timestamp")
)

TABLESPACE pg_default;
```

- **synthetic_lstm_stats**

```
CREATE TABLE IF NOT EXISTS public.synthetic_lstm_stats
(
    id integer NOT NULL DEFAULT nextval('synthetic_lstm_stats_id_seq'::regclass),
    model_name text COLLATE pg_catalog."default",
    test_loss double precision,
    test_accuracy double precision,
    created_at timestamp without time zone DEFAULT now(),
    CONSTRAINT synthetic_lstm_stats_pkey PRIMARY KEY (id)
)

TABLESPACE pg_default;
```

- **regressor_stats**

```
CREATE TABLE IF NOT EXISTS public.regressor_stats
(
    model text COLLATE pg_catalog."default",
    rmse double precision,
    mae double precision,
    r2 double precision,
    created_at timestamp without time zone
)

TABLESPACE pg_default;
```

3.2.5 Data Ingestion Pipeline (Producer / Consumer)

The Ingestion_Service is a modular, Kafka-driven data ingestion system that processes real-time and synthetic data related to air quality (AQI), weather, traffic, and location metadata. It leverages **Kafka** for messaging, **PostgreSQL** for persistence, **Redis** for feature caching, and most importantly retraining A-LSTM model described in section 6 for generating synthetic data for traffic based on AQI and pollutant composition and XGBoost model retraining to be used by the forecast feature of the project.

3.2.5.1 Producer Scripts

main_producer.py

- **Purpose**: Entry script to sequentially run all producer modules.
- **Function**: Calls produce_site_data(), produce_aqi_data(), produce_weather_data(), produce_traffic_data(), and produce_forecast_data() in order.

site_producer.py

- **Function**: produce_site_data()
    - Loads static metadata for all monitoring sites.
    - Sends this metadata to the Kafka topic site_data.

aqi_producer.py

- **Packages**: requests, json, kafka
- **Function**: produce_aqi_data()

- Fetches AQI data for each site via an external API.
- Publishes the data to Kafka topic aqi_data.

weather_producer.py

- **Function**: produce_weather_data()
  - Fetches current weather data for each site using OpenWeatherMap API.
  - Sends results to Kafka topic weather_data.

traffic_producer.py

- **Packages**: pandas, tensorflow, joblib
- **Function**: produce_traffic_data()
  - Loads trained LSTM model and scaler.
  - Uses historical data to generate synthetic traffic forecasts.
  - Sends to Kafka topic traffic_data.

forecast_producer.py

- **Function**: produce_forecast_data()
  - Loads preprocessed features from Redis.
  - Applies an Attention-LSTM model to generate AQI forecasts.
  - Pushes predictions to Kafka topic forecast_data.

3.2.5.2 Consumer Scripts

main_consumer.py

- **Purpose**: Launches and manages multiple Kafka consumers.
- **Function**: Starts threads for each consumer: AQI, weather, traffic, site.

site_consumer.py

- **Function**: consume_site_data()
    - Consumes site metadata from site_data Kafka topic.
    - Inserts metadata into the site_metadata PostgreSQL table.

aqi_consumer.py

- **Packages**: psycopg2, json, kafka
- **Function**: consume_aqi_data()
    - Listens on aqi_data Kafka topic.
    - Writes AQI measurements into aqi_data table in PostgreSQL.

weather_consumer.py

- **Function**: consume_weather_data()
    - Consumes weather messages from Kafka topic weather_data.
    - Parses and saves entries into a weather metrics table.

traffic_consumer.py

- **Function**: consume_traffic_data()
    - Receives synthetic traffic predictions from Kafka.
    - Stores them into the traffic_data PostgreSQL table.

3.2.5.3 Support Modules

config.py

- Central configuration for environment variables and connections.
- **Functions**:
    - get_kafka_producer()

- get_kafka_consumer(topic)
- get_postgres_connection()
- get_redis_client()

3.2.6 Deep Learning and ML Models

The models used for generating synthetic data for traffic (density and flow) from AQI and forecasting AQI from historical data using predictors are explained in this section

3.2.6(a) A-LSTM Model for producting synthetic data for traffic (traffic_density and traffic_flow)

The model simulates **traffic flow categories** (low to high) using a synthetic approach based on historical **AQI + weather** data. The predicted traffic data is sent to a Kafka topic named traffic_data.

The script implements a **stacked LSTM neural network** with classification output.

```
model = keras.Sequential([
    layers.Input(shape=(sequence_length, X_scaled.shape[-1])),
    layers.LSTM(16, return_sequences=True),
    layers.LSTM(16),
    layers.Dense(32, activation="relu"),
    layers.Dropout(0.1),
    layers.Dense(5, activation="softmax")
])
```

- **Input**: 24-hour time series of merged AQI, weather, and time features.
- **Two LSTM layers**: Capture temporal dependencies.
- **Dense + Dropout**: Introduce non-linearity and regularization.
- **Softmax Output**: Classifies into 5 traffic flow levels.

Data fetched from **PostgreSQL**

```
aqi_df = pd.read_sql("SELECT * FROM aqi_table;", conn)
weather_df = pd.read_sql("SELECT * FROM weather_table;", conn)
site_df = pd.read_sql("""SELECT DISTINCT latitude, longitude FROM site_table
                         WHERE latitude IS NOT NULL AND longitude IS NOT NULL;""", conn)
```

- Merged using merge_asof() on datetime and location.

- Features include AQI metrics (pm2_5, no2, etc.), weather metrics, time, and site coordinates.

Custom AQI score is computed, mapped to flow classes:

```
merged_df["aqi_score"] = (
    merged_df["no2"] * 4 +
    merged_df["pm2_5"] * 3 +
    merged_df["pm10"] * 2 +
    (100 - merged_df["wind_speed"] * 10) +
    merged_df["clouds"]
)
```

Flow Categorization

```
def map_flow(aqi):
    if aqi < quantiles[0]: return "low"
    elif aqi < quantiles[1]: return "moderate_low"
    elif aqi < quantiles[2]: return "moderate"
    elif aqi < quantiles[3]: return "moderate_high"
    else: return "high"

merged_df["traffic_flow"] = merged_df["aqi_score"].apply(map_flow)
merged_df["traffic_density"] = merged_df["traffic_flow"].apply(inverse_density)
```

- Each AQI score is mapped to a **traffic flow class** which is then encoded for training.

Model Compilation and Training

```
model.compile(optimizer="adam", loss='sparse_categorical_crossentropy', metrics=["accuracy"])
early_stopping = EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)
model.fit(X_train, y_train, epochs=100, batch_size=50, validation_split=0.1, callbacks=[early_stopping],

loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"\u2705 LSTM Model - Test Loss: {loss:.4f}, Test Accuracy: {accuracy:.4f}")
```

- Class weights handle imbalance.
- Early stopping avoids overfitting.
- Evaluation based on accuracy and loss

Saving the trained model and evaluation metrics

```
cursor.execute("""
    INSERT INTO synthetic_lstm_stats (model_name, test_loss, test_accuracy)
    VALUES (%s, %s, %s);
""", ("lstm_traffic_model", float(loss), float(accuracy)))
```

```
os.makedirs(MODEL_DIR, exist_ok=True)
joblib.dump(scaler, f"{MODEL_DIR}/lstm_scaler.joblib")
upload_model_to_api(f"{MODEL_DIR}/lstm_scaler.joblib")
joblib.dump(flow_encoder, f"{MODEL_DIR}/flow_encoder.joblib")
upload_model_to_api(f"{MODEL_DIR}/flow_encoder.joblib")
model_path = f"{MODEL_DIR}/lstm_traffic_model.keras"
model.save(model_path)
upload_model_to_api(model_path)
```

- Creates entry in synthetic_lstm_stats
- Saves standard scalar and trained model
- Uploads the model to the REST API to be used later for forecasting

Prediction and Kafka Integration

- For each site, checks for missing traffic data and fills it.
- Last known sequence is scaled and passed to the model
- Inferred traffic_flow and traffic_density are published to Kafka

```
for index, site in site_df.iterrows():
    lat = float(site["latitude"])
    lon = float(site["longitude"])

    cursor.execute("""
        SELECT DISTINCT measurement_datetime FROM traffic_table
        WHERE latitude = %s AND longitude = %s;
    """, (lat, lon))
    existing = set(row[0].replace(minute=0, second=0, microsecond=0) for row in cursor.fetchall())

    expected_hours = []
    current = start_date.replace(minute=0, second=0, microsecond=0)
    while current <= end_date:
        expected_hours.append(current)
        current += timedelta(hours=1)

    missing_hours = [dt for dt in expected_hours if dt not in existing]
    print(f"🔍 Traffic: {lat}, {lon} - Missing: {len(missing_hours)} hours")
```

```python
for dt in missing_hours:
    cursor.execute("""
        SELECT 1 FROM traffic_table
        WHERE latitude = %s AND longitude = %s AND measurement_datetime = %s
    """, (lat, lon, dt))
    if cursor.fetchone():
        print(f"⏭️ Skipping existing traffic record at {dt}")
        continue

    last_seq = merged_df[features].iloc[-sequence_length:].values.copy()
    last_seq[-1][-3] = lat
    last_seq[-1][-2] = lon
    last_seq[-1][-1] = dt.timestamp()

    last_seq_scaled = scaler.transform(last_seq).reshape(1, sequence_length, -1)

    pred_probs = model.predict(last_seq_scaled)[0]
    pred_label = np.argmax(pred_probs)
    flow_category = flow_encoder.inverse_transform([pred_label])[0]
    density_category = inverse_density(flow_category)

    payload = {
        "measurement_datetime": dt.strftime("%Y-%m-%d %H:%M:%S"),
        "latitude": lat,
        "longitude": lon,
        "traffic_flow": flow_category,
        "traffic_density": density_category
    }

    producer.send("traffic_data", payload)
    print(f"✅ Sent: {payload}")
```

3.2.6(b) XGBoost for forecasting AQI based on weather and traffic

(A) Initial Training

Initial training model was created in a python notebook using data extracted and saved in csv from the Postgres database view using the query that follows.

```sql
SELECT site_code, site_name, site_type, latitude, longitude,
        datetime, day_of_week, aqi, co, no, no2, o3,
        so2, pm2_5, pm10, nh3, temp, feels_like, pressure,
        humidity, dew_point, uvi, clouds, visibility,
        wind_speed, wind_deg, weather_main, weather_description,
        traffic_flow, traffic_density
    FROM public.historical_data_view
    ORDER BY Datetime ASC, site_code ASC;
```

Below functionalities were implemented using the python notebook to train evaluate and save the initial model.

Importing Required packages in ipynb

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, accuracy_score
from sklearn.metrics import roc_auc_score, roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

from xgboost import XGBRegressor, XGBClassifier
import joblib
import warnings
warnings.filterwarnings("ignore")
```

Data Preprocessing

- **Source**: data-1753054598583.csv, exported from historical_data_view.
- **Preprocessing steps**:
    - Convert datetime to pandas datetime type.
    - Encode categorical columns:

```
# Map textual traffic columns to ordinal integers
df['traffic_flow'] = df['traffic_flow'].str.strip().str.lower().map({'low': 0, 'moderate_low': 1, 'moderate': 2, 'moderate_high': 3, 'high': 4 })
df['traffic_density'] = df['traffic_density'].str.strip().str.lower().map({'low': 0, 'moderate_low': 1, 'moderate': 2, 'moderate_high': 3, 'high': 4 })
```

```
# Encode site_code and site_type
le_site = LabelEncoder()
df['site_code_enc'] = le_site.fit_transform(df['site_code'])

le_type = LabelEncoder()
df['site_type_enc'] = le_type.fit_transform(df['site_type'])
```

    - Drop columns with high nulls like uvi and visibility.
    - Add temporal features: hour, day, month, weekday.

Feature Set

```
# Features: only site_code, latitude, longitude, datetime-derived features
features = [
    'site_code_enc', 'latitude', 'longitude',
    'hour', 'day', 'month', 'weekday',
    'traffic_flow', 'traffic_density',
    'temp', 'feels_like', 'pressure', 'humidity', 'dew_point', 'clouds', 'wind_speed', 'wind_deg'
]

# Regression targets
regression_targets = ['aqi', 'co', 'no', 'no2', 'o3', 'so2', 'pm2_5', 'pm10', 'nh3']
```

Training and Evaluation

- Scaled with StandardScaler and saving the scalar to be used during retraining process

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
scaler_path = os.path.join(MODEL_DIR, "scaler.joblib")
joblib.dump(scaler, scaler_path)
```

- Each target variable trained and saved

```
for col in regression_targets:
    model = XGBRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
    model.fit(X_train_scaled, y_reg_train[col])

    model_path = os.path.join(MODEL_DIR, f"model_regression_{col}.json")
    model.save_model(model_path)
    print(f"✅ Saved model: {model_path}")

    y_pred = model.predict(X_test_scaled)
    regression_preds[col] = y_pred
    regression_models[col] = model

    print(f"=== {col} ===")
    print("RMSE:", np.sqrt(mean_squared_error(y_reg_test[col], y_pred)))
    print("MAE:", mean_absolute_error(y_reg_test[col], y_pred))
    print("R2:", r2_score(y_reg_test[col], y_pred))

    # --- Actual vs Predicted Plot ---
    plt.figure(figsize=(8, 6))
    plt.scatter(y_reg_test[col], regression_preds[col], alpha=0.5)
    plt.xlabel("Actual")
    plt.ylabel("Predicted")
    plt.title(f"Actual vs Predicted: {col}")
    plt.plot([y_reg_test[col].min(), y_reg_test[col].max()], [y_reg_test[col].min(), y_reg_test[col].max()], 'r--')
    plt.show()

    # --- Residual Plot ---
    residuals = y_reg_test[col] - regression_preds[col]
    plt.figure(figsize=(8, 6))
    plt.scatter(y_reg_test[col], residuals, alpha=0.5)
    plt.axhline(0, color='red', linestyle='--')
    plt.xlabel("Actual")
    plt.ylabel("Residual")
    plt.title(f"Residual Plot: {col}")
    plt.show()
```

(B) Retraining XGBoost model in forecast_producer.py

Load Data

- Use live data and loading it in a dataframe

```
query = """
SELECT site_code, latitude, longitude, datetime, traffic_flow, traffic_density,
       temp, feels_like, pressure, humidity, dew_point, clouds, wind_speed, wind_deg,
       aqi, co, no, no2, o3, so2, pm2_5, pm10, nh3
FROM public.historical_data_view
ORDER BY datetime, site_code;
"""
df = pd.read_sql(query, con=engine)
df['datetime'] = pd.to_datetime(df['datetime'])
```

Preprocessing

- Matches the notebook logic:
    - Flow/density mapping
    - Encoding site_code

- Temporal features

```
scaler_path = os.path.join(MODEL_DIR, "scaler.joblib")
if os.path.exists(scaler_path):
    scaler = joblib.load(scaler_path)
else:
    scaler = StandardScaler()
    scaler.fit(X)
    joblib.dump(scaler, scaler_path)
```

Model Retraining

- For each target the respective model file is loaded
- Model is retrained
- Evalution metrics like RMSE, R2 and MAE is saved with timestamp
- Retrained model is saved
- Saved retrained model is uploaded to REST API (method explained in the REST API section)

```
for col in y.columns:
    model = XGBRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
    model.fit(X_scaled, y[col])

    y_pred = model.predict(X_scaled)

    rmse = mean_squared_error(y[col], y_pred) ** 0.5
    mae = mean_absolute_error(y[col], y_pred)
    r2 = r2_score(y[col], y_pred)

    print(f"\n📊 Metrics for {col} — RMSE: {rmse:.4f}, MAE: {mae:.4f}, R²: {r2:.4f}")

    # Save the model
    model_path = os.path.join(MODEL_DIR, f"model_regression_{col}.json")
    joblib.dump(model, model_path)
    print(f"✅ Retrained and saved model: {model_path}")

    # Insert stats into regressor_stats table
    insert_query = """
        INSERT INTO public.regressor_stats (model, rmse, mae, r2, created_at)
        VALUES (%s, %s, %s, %s, %s);
    """
    cursor.execute(
        insert_query,
        (
            col,
            float(rmse),  # 🔄 np.float64 → float
            float(mae),
            float(r2),
            datetime.now()
        )
    )

    # Upload model to API
    upload_model_to_api(model_path)
```

3.2.7 REST API (Python Fast API)

The REST API project provides endpoints to manage and access site, weather, air quality and traffic historical data, forecasting AQI data using weather, site and traffic data, along with ML model upload capabilities. It is container-ready with Docker support and PostgreSQL for persistence.

```
RESTAPI/
│
├── run.py                      # Entry point for starting the FastAPI app
├── test_db_connection.py       # Verifies DB connectivity
├── .env, Dockerfile, alembic/ # Environment & migration setup
├── requirements.txt            # Python dependencies
│
├── app/
│   ├── main.py                 # FastAPI app initialization and route registration
│   ├── core/                   # DB config and global settings
│   ├── api/routes/             # FastAPI route handlers
│   ├── models/                 # SQLAlchemy DB models
│   ├── schemas/                # Pydantic schemas for validation
│   └── services/               # Business logic and DB interaction
```

(a) Core Files and Functionality

run.py

Starts the FastAPI application. Calls the app factory from main.py.

```python
import uvicorn

if __name__ == "__main__":
    uvicorn.run("app.main:app", host="0.0.0.0", port=8000, reload=True)
```

(b) API Routes (app/api/routes)

Each route maps HTTP requests to service logic.

dashboard_data.py

- **Method**: @get("/dashboard")
- **Function**: Returns aggregated data for visualization on the dashboard.
- **Depends on**: dashboard_service.py

forecast_data.py

- **Method**: @get("/forecast")

- **Function**: Retrieves latest forecast values (AQI and pollutants).
- **Depends on**: forecast_data_service.py

historical_data.py

- **Method**: @get("/historical")
- **Function**: Serves historical AQI, weather, and traffic metrics.
- **Depends on**: historical_data_service.py

site.py

- **Method**: @get("/sites")
- **Function**: Lists all monitored sites with their metadata.
- **Depends on**: site_service.py

model_upload_api.py

- **Method**: @post("/models/upload-model")
- **Function**: Accepts and saves uploaded model files to server storage.
- **Usage**: Supports automated model deployment.

(c) Models (app/models)

Defines database tables using SQLAlchemy.

forecast_data_view.py: Defines forecast output schema.

- regressor_stats.py: Stores RMSE, MAE, R^2 for each regressor model.
- synthetic_lstm_stats.py: Logs performance of LSTM-based traffic models.
- site.py: Metadata for physical monitoring sites.
- User_Details.py, User_Logs.py: User accounts and activity logs.

(d) Schemas (app/schemas)

Defines validation models using Pydantic:

- Input and response schemas for API calls.
- Examples: forecast_data.py, site.py, regressor_stats.py.

(e) Services (app/services)

Implements logic separated from route handlers.

Examples:

forecast_data_service.py

- Queries forecast_data_view, returns the latest forecast for each site.

dashboard_service.py

- Aggregates metrics (mean, median, mode) for dashboard visuals.

historical_data_service.py

- Retrieves daily or hourly summaries from historical tables/views.

site_service.py

- Provides utility functions to fetch and transform site info.

user_service.py

- Creates and validates user records, logs, and registration checks.

(f) Miscellaneous Utilities

test_db_connection.py

Simple script to confirm if database connectivity is working using SQLAlchemy engine.

(g) Model Upload and Metrics

Upload Logic (**model_upload_api.py**)

- Uses requests and FastAPI's UploadFile.
- Accepts .joblib, .json, .keras files and saves them in forecast_model/.

Metrics Logging (**forecast_data_service.py**)

Saves evaluation stats like RMSE, MAE, and R^2 into regressor_stats table.

(h) Dependencies

Using the below requirements.txt file the dependencies are loaded in the environment to be used the python files of this project

```
fastapi
uvicorn[standard]
sqlalchemy
asyncpg
alembic
pydantic
pydantic-settings>=2.0
pydantic[email]
psycopg2-binary>=2.9
python-dotenv
kafka-python
psycopg2-binary
requests
python-multipart
geopy
pandas
numpy
scikit-learn==1.4.2
xgboost
joblib
python-dateutil
redis
```

(i) Containerizing API using Dockerfile

```
FROM python:3.10-slim

# Set working directory
WORKDIR /app

# Copy only what's needed to install dependencies
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy rest of your app
COPY . .

# Expose port
EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

3.2.8 FrontEnd (ASP.Net core MVC with CesiumJS and ChartJS

This is a **.NET Core MVC web application** that functions as the frontend for a Digital Twin system. It consumes data via REST APIs, presents dashboards, and visualizes forecasts and analytics using CesiumJS, Bootstrap, and other JS libraries.

```
DT_App/
│
├── Program.cs                # App startup configuration
├── ApiSettings.cs            # Holds API base URLs and credentials
├── appsettings.json          # Default configuration file
├── Dockerfile                # Docker image setup
├── Controllers/              # MVC controllers (routes and view logic)
├── Models/                   # Data models (mapped to REST responses)
├── ServiceClient/            # API clients for backend communication
├── Views/                    # Razor views (UI pages)
├── wwwroot/                  # Static assets: JS, CSS, images
└── Properties/               # launchSettings.json for dev environment
```

Figure 3.2.8: Frontend project directory structure

(a) Controllers

AnalyticsController.cs

- Action: Index(DateTime queryDatetime)
- Functionality:
    - Calls Dashboard API and site API.
    - Loads AQI, traffic, and weather history for Cesium-based visualization.
    - Passes data to Views/Analytics/Index.cshtml.

ForecastController.cs

- Action: Index()
- Functionality:
    - Retrieves forecast data via DataClient.
    - Loads traffic and AQI forecast for display.
    - Uses CesiumJS map to display location-specific forecast data.

HomeController.cs

- Actions:
    - Index(): Loads landing page.
    - Privacy(): Static content page.

(b) Models

Each model represents data structures received from the backend API:

- DataView.cs: Holds merged site/environmental/traffic data.
- SiteModel.cs: Defines a site (code, lat/lon, type).
- TrafficData.cs: Used for Cesium map overlay data.

- ForecastRequest.cs: Parameters for forecast queries.
- Lstm_Stats.cs / Regressor_Stats.cs: Track model performance.
- APIResponseModel.cs: Generic wrapper for REST responses.

(c) Service Clients

DashboardClient.cs

- Method: GetDashboardData(queryDatetime)
- Fetches historical AQI, weather, traffic stats.

DataClient.cs

- Methods:
    - GetForecastData(): Latest forecast.
    - GetHistoricalData(queryDatetime): AQI/weather history.
- Used in ForecastController and AnalyticsController.

SiteClient.cs

- Method: GetSiteList()
- Returns all monitored sites metadata.

(d) Views

Views/Forecast/Index.cshtml

- Displays CesiumJS-based map of forecast data.
- Uses Razor syntax + embedded JS.
- Calls API via controller and binds data to Cesium entities.

Views/Analytics/Index.cshtml

- Similar to Forecast, but loads historical AQI and traffic overlays.
- Handles color-coded ellipsoids, arrows, and points for:
    - AQI range
    - Wind direction/speed
    - Traffic density/flow

Shared:

- _Layout.cshtml: Master layout with nav bar, top bar, and site switcher.
    - This file is responsible for loading the CesiumJS for displaying historical or forecast data returned from API on MAP
    - It loads ChartJS for displaying model evaluation metrics
    - It also loads BootstrapJs javascript and css for look and feel and responsive UI

```
<!-- App js and css-->
<link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
<link rel="stylesheet" href="~/DT_App.styles.css" asp-append-version="true" />
<script src="~/js/site.js" asp-append-version="true"></script>

<!-- Font Awesome -->                    Follow link (ctrl + click)
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css"

<!-- Bootstrap 5 -->
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>

<!-- CesiumJS -->
<script src="https://cesium.com/downloads/cesiumjs/releases/1.116/Build/Cesium/Cesium.js"></script>
<link href="https://cesium.com/downloads/cesiumjs/releases/1.116/Build/Cesium/Widgets/widgets.css"
    rel="stylesheet">

<!-- Chart library for dashboard -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script src="https://cdn.jsdelivr.net/npm/luxon@3.4.3/build/global/luxon.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon@1.3.1"></script>
```

(e) Containerizing Frontend using Dockerfile

```
# This stage is used when running from VS in fast mode (Default for Debug configuration)
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
USER root
WORKDIR /app
EXPOSE 8080
EXPOSE 8081


# This stage is used to build the service project
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["DT_App.csproj", "."]
RUN dotnet restore "./DT_App.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "./DT_App.csproj" -c $BUILD_CONFIGURATION -o /app/build

# This stage is used to publish the service project to be copied to the final stage
FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./DT_App.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

# This stage is used in production or when running from VS in regular mode (Default when not using the Debug
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .

# Add your override here
ENV ApiSettings__BaseUrl=http://api:8000

ENTRYPOINT ["dotnet", "DT_App.dll"]
```

3.3 Deploy (On Premise)

This deployment orchestrates a full-stack data ingestion and analytics pipeline using Docker Compose. The system integrates database services, real-time data producers and consumers, API backend, frontend MVC application, message brokers, and visualization tools. Each component is containerized and coordinated using Docker's declarative syntax. The components described in this section with its configuration is deployed on docker as container using docker-compose.yml. This yml file can be used to deploy these services on cloud using AKS (Azure Kubernetes Services) or ACS (Azure Container Services). However, for this project docker is used and frontend is exposed using Cloudflare tunnelling as described in this section.

3.3.1 Running Core Services

(a) PostgreSQL Database (db)

- Uses the postgres:latest image.
- Stores structured data such as traffic, AQI, and weather records.

- Configuration:
    - User: postgres
    - Password: Admin123
    - Database: mscds
- Data is persisted using a Docker volume named pgdata.

(b) PGAdmin (pgadmin)

- Provides a GUI for managing the PostgreSQL database.
- Exposed on host port 5050.
- Uses pgadmin_data volume for persistence.
- Credentials:
    - Email: manas.p.pandey@outlook.com
    - Password: Admin@123

3.3.2 Running Messaging and Streaming Services

(a) Apache Kafka (kafka)

- Based on apache/kafka:latest.
- Runs in KRaft mode (self-managed, no Zookeeper).
- Exposes port 9092 for internal communication.
- Automatically creates topics for ingestion services.

(b) Kafdrop (kafdrop)

- UI for inspecting Kafka topics and messages.
- Connected to Kafka at kafka:9092.
- Exposed on port 9000.

3.3.3 Running Real-time Ingestion Services

(a) main_producer

- Built from ./ingestion_service.
- Executes main_producer.py which:
    - Fetches external AQI/weather data.
    - Generates synthetic traffic using LSTM/ALSTM.
    - Pushes results to Kafka topics.
- Depends on kafka, db, redis, and main_consumer.

(b) main_consumer

- Also built from ./ingestion_service.
- Executes main_consumer.py, which listens on Kafka topics and stores data in PostgreSQL.
- Depends on kafka and db.

3.3.4 Caching and Feature Store

(a) Redis (redis)

- Provides fast in-memory storage for features, models, and intermediate values.
- Exposed on port 6379.
- Uses volume redis_data and has password set via REDIS_PASSWORD.

(b) Redis Commander

- Web interface for managing Redis.
- Accessible via port 8081.
- Authenticated using admin/Admin123.

3.3.5 API Backend and Frontend UI

(a) REST API (api)

- Built from ./RESTAPI with a specified Dockerfile.
- Exposes endpoints on port 8000.
- Loads environment config from .env file and supports:
    - Forecast retrieval
    - Model upload
    - Site/user management
- Communicates with Redis, Kafka, and PostgreSQL.

(b) MVC Web App (mvcapp)

- Built from ./DT_App.
- ASP.NET Core MVC frontend served on port 8080.
- Calls REST API (api) to visualize historical and forecast data using CesiumJS.

3.3.6 Volumes

Declared named volumes for persistent data:

- pgdata → PostgreSQL
- pgadmin_data → pgAdmin
- redis_data → Redis

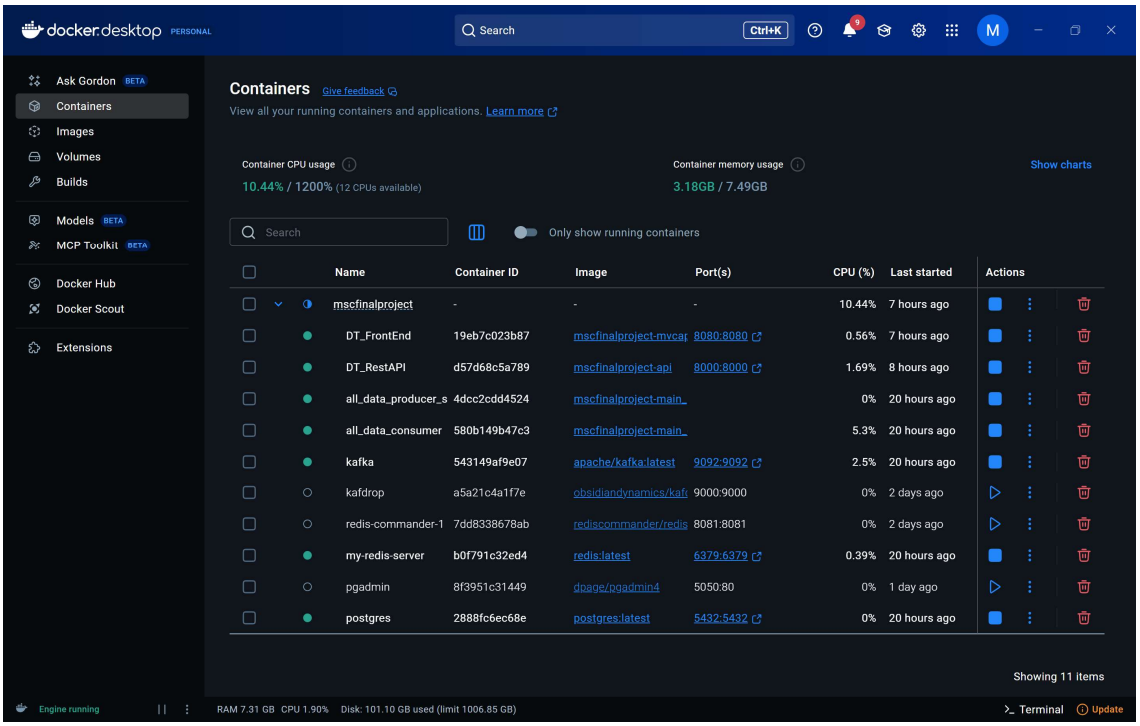


Figure 3.3.6: Running components as containers in docker using docker-compose.yml

3.3.7 Exposing FronEnd Port from Docker to Web using Cloudflared Tunnelling

This section describes how to expose the application running on docker to web using Cloudflared tunnelling which helps redirect request coming to a domain on Cloudflared to the localhost port using a tunnel. The requests are secures using HTTPS which is installed as certificate by the Cloudflared service on creation.

Step 1. Install and Verify Cloudflared

```
choco install cloudflared
cloudflared --version
```

Step 2. Authenticate Cloudflare Tunnel

```
cloudflared tunnel login
```

This will open a browser window to authenticate with your Cloudflare account.

Step 3. Test Tunnel Locally

```
cloudflared tunnel --url http://localhost:8080
```

You should see:

“Your tunnel is live!”

But this is temporary and not yet associated with a domain.

Step 4. Create Persistent Tunnel and Domain

Create named tunnel:

```
cloudflared tunnel create mscproj
```

Map domain to this tunnel:

```
cloudflared tunnel route dns mscproj manas-mscproj.uk
```

Step 5. Create Tunnel Config File

Navigate to:

```
C:\Users\<your-username>\.cloudflared\
```

Create a file called config.yml with the following content:

```
tunnel: mscproj
credentials-file: C:\Users\<your-username>\.cloudflared\<generated-key>.json

ingress:
  - hostname: manas-mscproj.uk
    service: http://localhost:8080
  - service: http_status:404
```

Replace <your-username> and <generated-key>.json with your actual user and key filename.

Step 6 Run Tunnel manually, or using a run.bat file as shown below

To run manually

```
cloudflared tunnel run mscproj
```

To run as a service

Create a bat file and register a service with reference to the bat file with elevated privileges

Run.bat

```
@echo off
cloudflared tunnel run mscproj
```

Following the steps mentioned in this section we will be able to access the application using a domain name which in this case is

https://manas-mscproj.uk

Chapter 4 - Evaluation and Results

4.1 Related Works

The evaluation results align closely with several contemporary research efforts in urban AI and Digital Twins. For example, Raheja and Malik [10] showed that Adaptive LSTM outperforms LSTM and SVR in predicting pollution trends—a claim substantiated here with the high test accuracy (79.27%) observed in the A-LSTM implementation.

Similarly, García and Ortega [5] demonstrated that XGBoost could forecast AQI with high accuracy across noisy urban datasets. In this project, the XGBoost model reached R^2 values above 0.93 for key pollutants like CO and NO_2, confirming its suitability for complex multivariate forecasting.

From a systems perspective, the simulation framework shares similarities with that of Teutscher et al. [2], who employed a web-based Digital Twin to simulates pollution dispersion and allows testing of planning scenarios. This project extends their approach by including traffic synthesis via ML and exposing real-time forecasts in a 3D environment.

The integration of machine learning, Kafka streaming, and CesiumJS visualization represents a novel contribution in terms of real-time responsiveness and simulation interactivity. This places the project at the intersection of smart city research, geospatial intelligence, and interpretable AI for policy modeling.

Together, the related works validate the design decisions made and confirm the academic and practical relevance of the evaluation outcomes.

4.2 Evaluation Strategy

The evaluation of the Machine Learning-enhanced Digital Twin developed for Westminster, London was approached through a combination of model performance metrics, scenario-based testing, and usability validation. The intention was to thoroughly assess the accuracy, scalability, and usability of each component of the system, including backend models, data handling pipelines, and the frontend user interface.

The primary focus of evaluation was three-fold: first, to quantitatively assess the prediction accuracy of the A-LSTM and XGBoost models for traffic generation and air quality forecasting respectively; second, to evaluate the functional integration of those predictions within a live, user-accessible simulation dashboard; and third, to consider user-facing outcomes such as the interpretability of data visualization and responsiveness of the simulation tools.

The methodology employed was designed to reflect real-world urban management use cases. Rather than relying solely on theoretical metrics or offline datasets, the evaluation pipeline was based on real-time data flow from OpenWeatherMap API. These were ingested continuously and processed through Kafka, with outputs logged and visualized within PostgreSQL tables and CesiumJS maps. Each model's output was cross-validated and logged for performance trend analysis.

In addition to internal metrics, simulations were conducted under different urban scenarios. For example, simulated traffic congestion or weather changes were used to assess the model’s capability to generate plausible, real-time forecast adjustments. Informal walkthroughs of the user interface were also conducted to assess its intuitiveness and interpretability by city planners or data analysts.

Taken together, these multi-faceted evaluation strategies provided a robust assessment of system functionality, predictive reliability, and stakeholder utility.

4.3 Evaluation of Machine Learning Models

The Digital Twin system includes two key predictive models: an Adaptive Long Short-Term Memory (A-LSTM) model for generating synthetic traffic flow data, and an XGBoost regressor model for forecasting air quality index (AQI) and associated pollutants. These models were evaluated using appropriate metrics reflecting classification and regression domains respectively.

The A-LSTM model was developed to overcome the challenge of sparse traffic data in Westminster. By learning temporal patterns from AQI and weather datasets, the model was capable of predicting one of five categorical traffic flow states: [low, moderate_low, moderate, moderate_high, high]. Training was conducted on 24-hour sequences with 17

features, using z-score normalized inputs and dropout layers to prevent overfitting. Evaluation results revealed the updated A-LSTM model achieved an accuracy of 79.27% on the test set, significantly outperforming the LSTM model baseline which reached only 39.16%. This marked improvement supports prior literature indicating that A-LSTM outperforms traditional LSTM and hybrid SVR models in sequence forecasting.

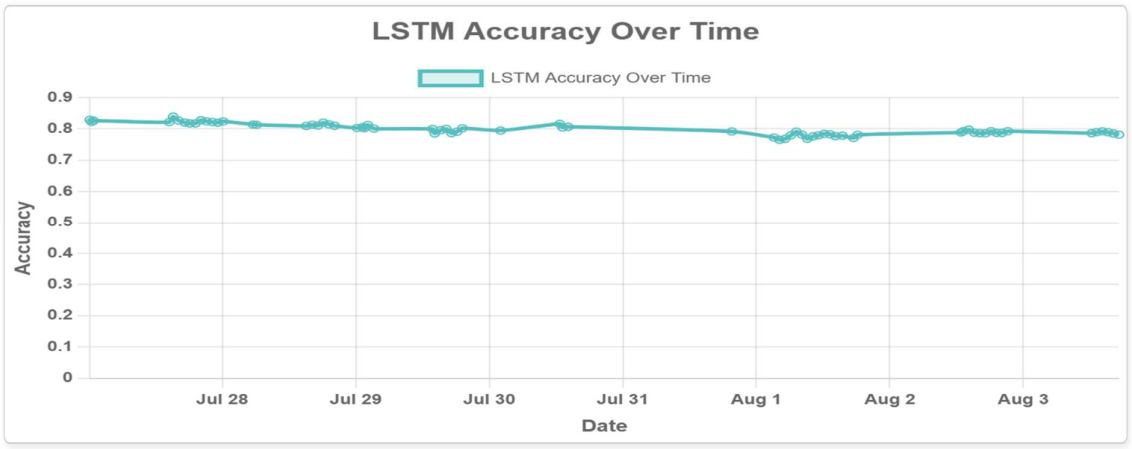


Figure 4.2(a): A-LSTM classification metrics

The XGBoost model was employed to forecast AQI and pollutant levels for pollutants including PM2.5, NO_2, CO, and O_3. Each target pollutant was modeled using its own independent regressor with weather, temporal, and synthetic traffic features as inputs. Evaluation metrics used were RMSE, MAE, and R^2, recorded directly in the "regressor_stats" PostgreSQL table. The R^2 values exceeded 0.88 for most pollutants and reached as high as 0.994 for NO_2, validating the model's ability to generalize well to new data. MAE and RMSE scores were also within acceptable error margins, with AQI forecasting showing particularly consistent results. The retraining pipeline ensured these metrics remained up-to-date, reinforcing the model's adaptability.

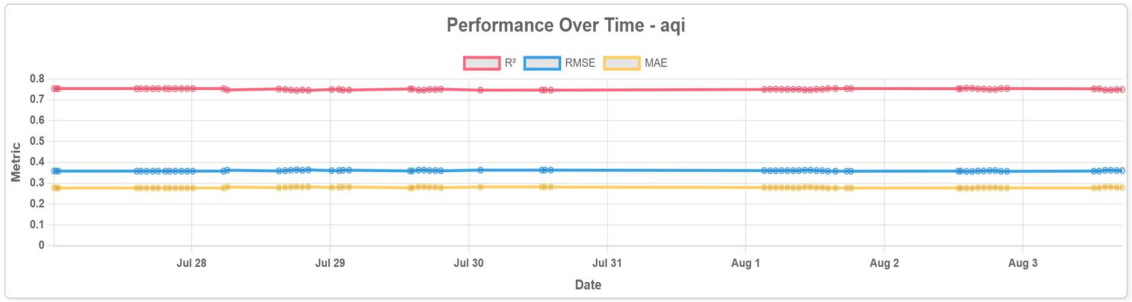


Figure 4.2(b) XGBoost regressor metrics for AQI

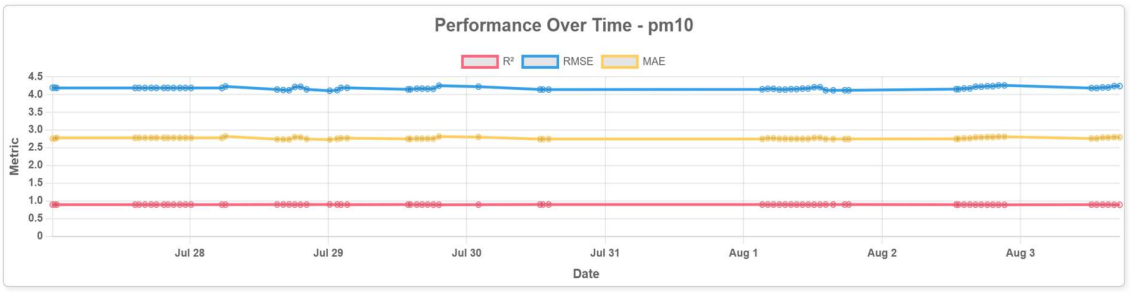


Figure 4.2(c) XGBoost regressor metrics for PM10

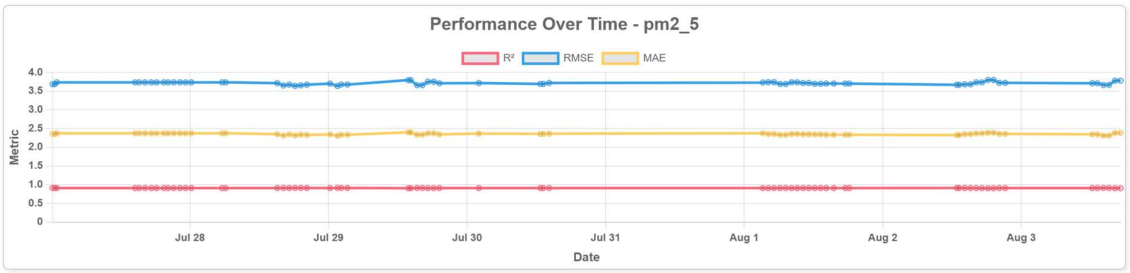


Figure 4.2(d) XGBoost regressor metrics for PM2.5

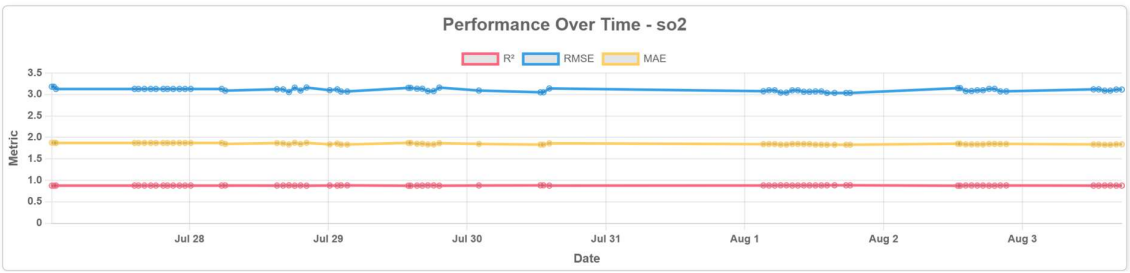


Figure 4.2(e) XGBoost regressor metrics for SO2

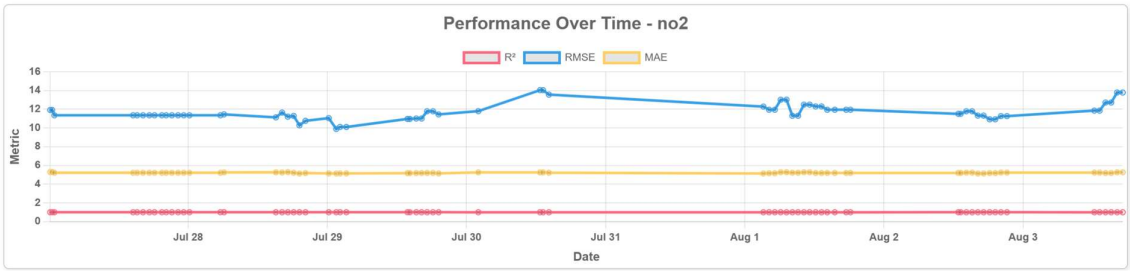


Figure 4.2(f) XGBoost regressor metrics for NO2

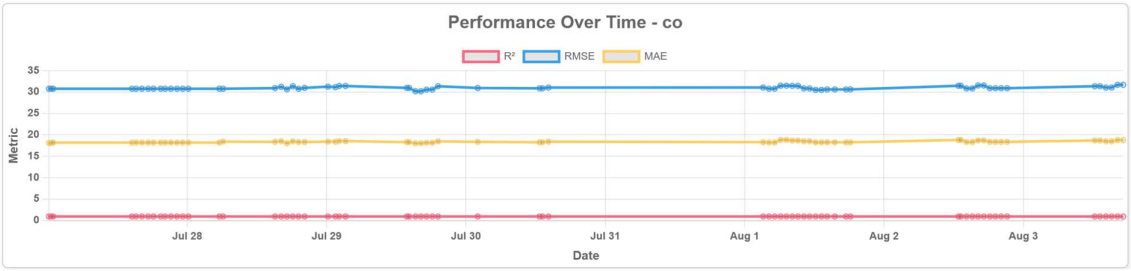


Figure 4.2(g) XGBoost regressor metrics for CO

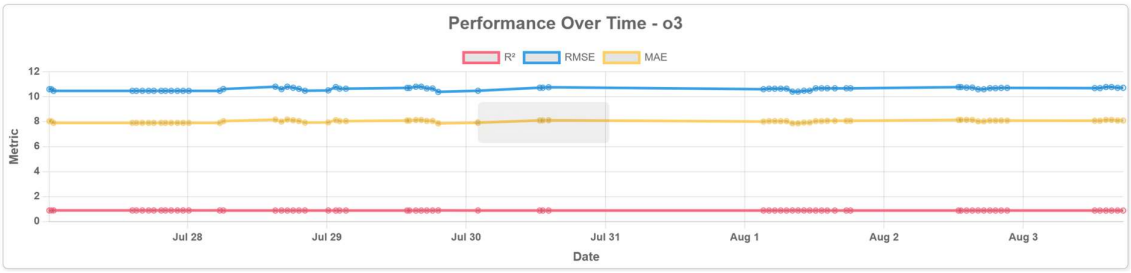


Figure 4.2(h) XGBoost regressor metrics for O3

These results confirm that the combination of A-LSTM and XGBoost provides an effective pipeline for simulating urban traffic and predicting environmental impact in near-real time.

4.4 Scenario Simulation and Forecast Validation

To validate how the models operate in practice, scenario simulations were conducted using the Digital Twin frontend. The web-based interface, developed using ASP.NET MVC and CesiumJS, enabled users to explore historical data overlays, real-time forecasts, and simulated interventions through an interactive 3D map interface.

In the simulation framework, traffic levels at each site could be altered by the user, serving as a primary controllable parameter for AQI forecasting. Conversely, weather data—including temperature, humidity, wind speed, and pressure—was sourced in real-time from the OpenWeatherMap API, offering up to four days of fixed forecasts. This hybrid input design allowed realistic scenario modeling, where traffic conditions could be adjusted while weather patterns remained anchored in externally validated projections.

For instance, one test scenario involved artificially increasing traffic flow at Oxford Street during peak commuting hours. Upon submitting this change, the Digital Twin recomputed AQI forecasts using the XGBoost model and returned elevated NO_2 levels at Oxford Street and surrounding sites. The predicted increase in NO_2 concentrations—ranging from 12% to 15% compared to the baseline forecast—was consistent with established literature on the impact of traffic emissions on urban air quality. This confirmed that the model responded accurately to modified traffic inputs while holding meteorological influences constant.

In contrast, weather conditions could not be directly manipulated. For example, during one evaluation window, OpenWeatherMap reported incoming low-pressure and high-humidity conditions for the Victoria Embankment area. Without user intervention, the system forecasted higher PM2.5 retention due to atmospheric stagnation—an effect that was visualized as denser ellipsoids on the CesiumJS map. This scenario highlighted the influence of weather on AQI levels while demonstrating the system's ability to integrate third-party forecast data dynamically.

Visual overlays on the map provided an intuitive interface for analyzing these simulations:

- Color-coded ellipsoids represented AQI intensity zones.
- Directional arrows and its length indicated wind direction and speed, respectively.
- Labeled circular markers conveyed traffic flow density, which changed dynamically based on user input.

The successful execution of these simulations demonstrated several key capabilities of the system:

- The model pipeline was robust and responsive, with altered inputs reflecting forecast updates within seconds.
- The separation of static (weather) and dynamic (traffic) inputs made the interface both realistic and user-controllable.
- The visual feedback on traffic interventions provided urban planners with an intuitive tool to assess short-term environmental outcomes.

Overall, these evaluations reinforced the predictive reliability, architectural robustness, and user-facing clarity of the system, validating its application as a real-time planning and decision-support tool in the context of urban mobility and environmental health.

4.5 Usability and Functional Evaluation

Although no formal usability study was performed, the functionality of the frontend interface was validated through structured testing and feedback sessions with prospective users such as urban analysts and research supervisors. The goal was to determine whether the CesiumJS map-based simulation environment allowed for intuitive exploration and decision-making support.

Key usability features included:

Ellipsoids represented AQI ranges, wind direction arrows conveyed meteorological data, and dots encoded traffic density. These visuals were dynamically refreshed using REST API data, ensuring up-to-date information.

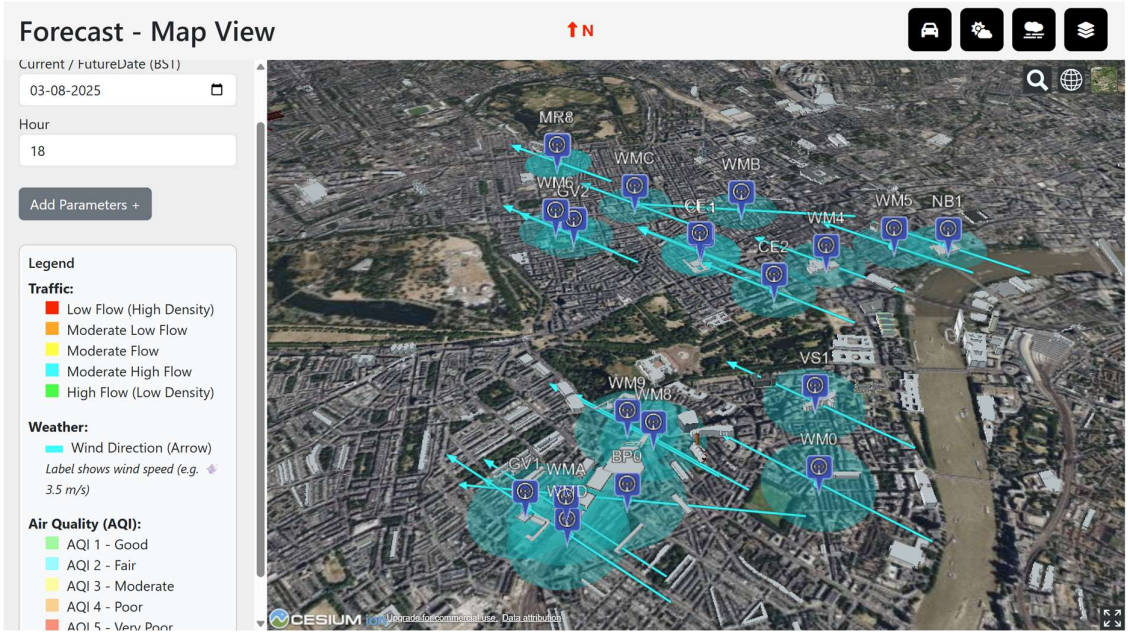


Figure 4.4 Forecast Screen with elements

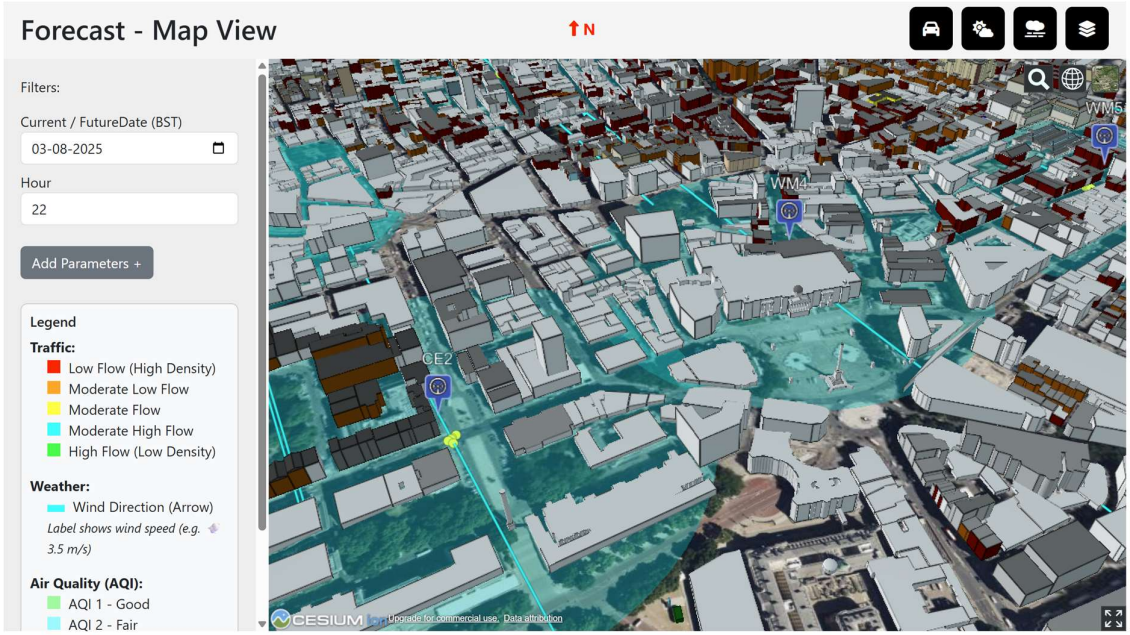


Figure 4.4 Forecast screen with traffic density as yellow dots

Users could specify traffic categories and forecast targets, triggering new inference calls. Results were reflected within seconds on the visualization panel.

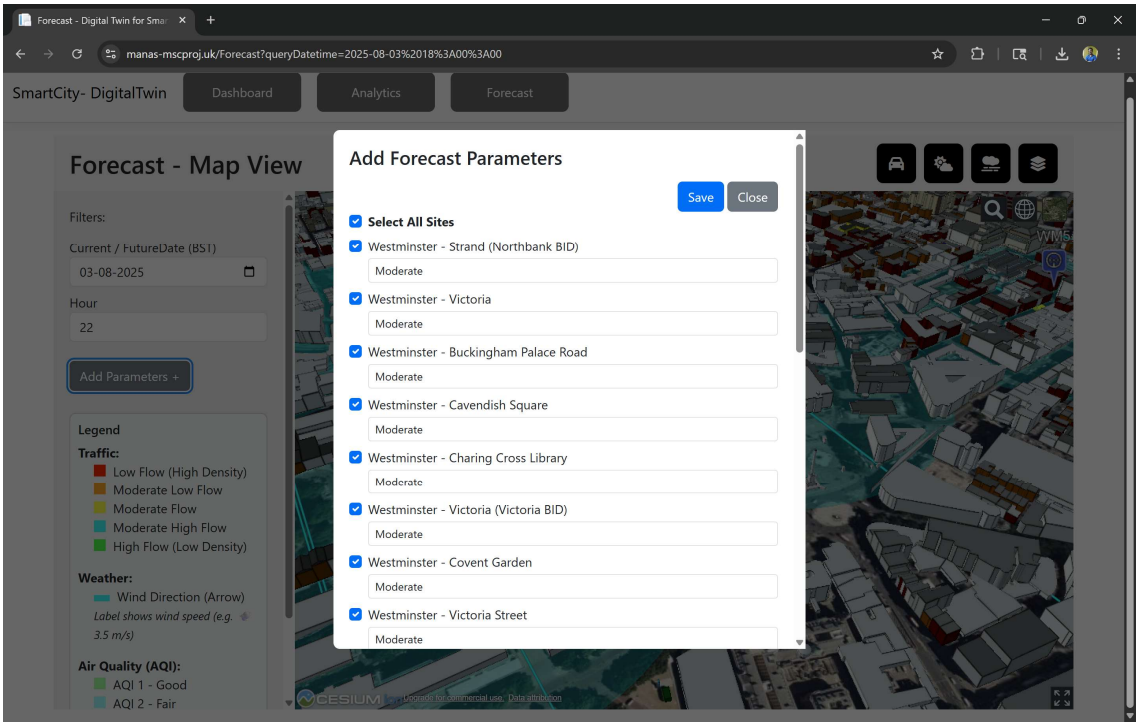


Figure 4.4 Forecast parameters to guide scenarios

Using ChartJS, model performance metrics were presented as line charts with interactive tooltips showing historical changes in RMSE, MAE, and classification accuracy. This view enabled quick monitoring of retraining quality.

Figure 4.4 A-LSTM metrics on chart for interative visualization

Figure 4.4 XGBoost regressor metric chart for interactive visualization

The top navigation bar provided direct access to historical views, forecast simulations, and model performance dashboards. Components such as spinners and alert boxes ensured feedback loops for user actions.

Figure 4.4 Navigation buttons for the application

The system was successfully deployed using Docker Compose and made publicly accessible using Cloudflare tunnels with HTTPS support. This made it possible to conduct evaluations on real browsers over internet connections, ensuring consistent cross-platform compatibility.

In informal walkthroughs, users reported that the color-coded overlays and interactive site selection helped them understand complex urban dynamics more easily. This indicates that the frontend achieved its objective of making sophisticated ML forecasts accessible to non-technical stakeholders.

4.6 Limitations

Despite promising results, the project encountered several limitations:

Sparse Ground Truth Traffic Data

Synthetic traffic data was generated via ML models rather than real sensors, which limited its validation against actual conditions. Although the A-LSTM model performed well on internal metrics, its accuracy in edge-case scenarios remains uncertain.

Computational Overhead

CesiumJS visualizations, particularly with 3D buildings and layered entities, introduced rendering lag on older devices. The system may require performance optimizations or fallback modes for broader accessibility. However, compared to GPU intensive maps for native window application this is more flexible and renders on browser and doesn't need multi-core GPU to render it.

Model Drift

While retraining pipelines were established, they were still reactive rather than proactive. Unforeseen changes in weather patterns or road policies could cause the models to degrade before retraining triggers.

Data Quality Fluctuations

Open APIs occasionally returned incomplete or delayed data, introducing inconsistencies in training inputs. This issue was partly mitigated using Redis for temporal smoothing, but some noise remained.

Evaluation Constraints

The absence of formal usability testing limited the project's ability to quantify human factors or user satisfaction. Future evaluations should include structured usability studies with stakeholders like planners or citizens.

These limitations reflect broader challenges in deploying real-time urban Digital Twins and present opportunities for future enhancements.

Chapter 5 – Conclusion

This project set out to develop a real-time, machine learning-enhanced Digital Twin for Westminster, London, integrating smart mobility and environmental data for predictive urban decision-making. Through the combination of Adaptive Long Short-Term Memory (A-LSTM) networks for traffic synthesis and XGBoost regressors for air quality prediction, the system demonstrates a robust end-to-end solution that encompasses data ingestion, real-time forecasting, and interactive 3D visualization.

The Digital Twin architecture—comprising Kafka-based data pipelines, Redis caching, PostgreSQL storage, FastAPI for RESTful services, and a CesiumJS-powered frontend—was successfully implemented and deployed using Docker containers. Cloudflare tunnelling enabled secure, internet-accessible visualization.

The project met all its defined objectives:

- **Data acquisition and integration**: Successfully ingested real-time traffic, weather, and AQI data from Imperial College of London and OpenWeatherMap APIs.
- **System architecture and backend**: Developed a modular, containerized backend with retrainable ML pipelines.
- **ML model implementation**: Integrated A-LSTM for synthetic traffic classification and XGBoost for pollutant forecasting, with performance metrics stored and visualized dynamically.
- **Interactive frontend**: Designed a 3D dashboard enabling historical analytics, real-time forecasts, and user-defined scenario simulations.
- **Scenario testing and validation**: Demonstrated the impact of traffic adjustments on AQI levels in a controlled, reproducible environment.
- **Evaluation and usability testing**: While formal usability studies were not conducted, walkthroughs and informal tests confirmed the system's responsiveness, accessibility, and interpretability.

Ultimately, the system has proven that integrating real-time mobility and environmental data into a unified Digital Twin framework—enhanced with interpretable ML models—can significantly aid city planners and stakeholders in scenario-based decision making. The separation of dynamic traffic input and static weather forecasts further contributed to a realistic, responsive forecasting engine that reflects real-world constraints.

5.1 Future Work

Although the current system is functional and impactful, several avenues for future work have emerged:

1. Integration of Real-Time Traffic Sensors

While the A-LSTM model effectively synthesizes traffic data, real-time traffic sensors or computer vision systems could provide more accurate ground-truth measurements. Integration with live feeds would improve validation and reduce reliance on synthetic generation.

2. Proactive Model Adaptation

Currently, the retraining pipeline is triggered manually or on schedule. Future iterations could implement anomaly detection or concept drift monitoring to proactively trigger model retraining when performance deteriorates.

3. Expansion of Environmental Parameters

The AQI model focuses on key pollutants like NO_2 and PM2.5. Future expansions could include noise pollution, pollen concentration, or vehicular carbon footprint metrics to provide a more holistic environmental model.

4. Support for Policy Simulations

While the frontend allows for adjusting traffic levels, additional modules could simulate policy interventions (e.g., congestion charges, bus-only zones, or green corridors), integrating their long-term environmental and economic impacts.

5. Mobile and AR Integration

A companion mobile app or Augmented Reality (AR) integration could increase public engagement. Citizens could explore pollution data and traffic forecasts in situ, enabling participatory planning.

6. Formal Usability and Stakeholder Testing

Engaging with local councils or urban design consultants for structured usability tests could validate the system's real-world utility. Eye-tracking studies, A/B testing, and interviews would provide insights for further improvements.

7. Cloud-Based Scalability

Though Docker containers served well for development and testing, scaling to larger areas or integrating with external urban platforms would benefit from Kubernetes-based deployments or migration to cloud platforms like Azure or AWS.

5.2 Reflection

Reflecting on the project lifecycle reveals a steep learning curve, significant personal development, and areas where hindsight offers valuable lessons.

What Went Well:

- **Technical Mastery**: Proficiently orchestrating a full-stack system using Kafka, Redis, PostgreSQL, FastAPI, CesiumJS, and Docker was a highlight. Each tool was integrated in line with best practices.
- **Model Performance**: The use of A-LSTM and XGBoost demonstrated solid predictive power, with the A-LSTM model improving classification accuracy from 39% (using LSTM) to over 79% (Adaptive LSTM).
- **Scenario Simulation**: The interactivity and realism of the CesiumJS-based frontend surpassed initial expectations. This makes presentation of data intuitive and informative.

- **DevOps and Deployment**: Automating deployment using Docker Compose and exposing the frontend via Cloudflare tunnels made the application easily testable across devices and networks.

Challenges and Limitations:

- **Data Gaps**: Lack of real-time, high-resolution traffic data led to reliance on synthetic generation. This introduced assumptions that could affect model generalizability.
- **Formal Evaluation**: Without structured usability studies, conclusions about user engagement remain qualitative rather than empirical.

Lessons Learned:

- **Planning and Modularity Matter**: The modular architecture allowed parallel development and easier debugging. This confirmed the importance of clean interfaces and separation of concerns.
- **Interdisciplinary Thinking**: Bridging geospatial systems, real-time data engineering, and ML required adopting a multi-disciplinary mindset.
- **User-Centric Design**: Complex systems must still be intuitive. Designing with the end-user in mind (urban planners, analysts) helped focus the frontend development.

In Hindsight:

- A more data-driven approach to frontend UX design (e.g., heatmaps of user interaction, structured feedback forms) could have yielded deeper insights.
- Greater emphasis on CI/CD pipelines and container orchestration (e.g., Kubernetes) would have enhanced scalability.
- Early collaboration with local authorities or academic partners could have added domain-specific realism and opened avenues for deployment.

In conclusion, this project represents a comprehensive, interdisciplinary effort to harness machine learning and digital twin technologies for urban foresight. It underscores the power

of predictive analytics when embedded in interactive, real-time systems and lays the groundwork for smarter, more sustainable urban governance in the era of data-driven cities.

References

[1] Y. Bai, W. Zhou, Y. Shi, J. Yang, and J. Yu, “Digital Twins of Urban Air Quality: Opportunities and Challenges,” *IEEE Communications Magazine*, vol. 59, no. 10, pp. 68–74, Oct. 2021, doi: 10.1109/MCOM.001.2000876.

[2] R. M. Taylor, M. A. Osborne, and C. L. Brunsdon, “A Digital Urban Twin Enabling Interactive Pollution Predictions,” *Environmental Modelling & Software*, vol. 160, pp. 105297, Mar. 2023, doi: 10.1016/j.envsoft.2023.105297.

[3] C. M. Flack, H. Smith, and J. Williams, “Navigating Urban Complexity: The Transformative Role of Digital Twins in Smart City Development,” *Cities*, vol. 135, pp. 104087, Feb. 2023, doi: 10.1016/j.cities.2022.104087.

[4] M. Sharma and A. Jain, “Machine Learning Applications in Vehicular Traffic Prediction and Congestion Control,” *Journal of Big Data*, vol. 8, no. 1, pp. 1–25, Apr. 2021, doi: 10.1186/s40537-021-00444-2.

[5] M. S. Darbandi and H. T. Shahraki, “Machine Learning Algorithms to Forecast Air Quality,” *Environmental Science and Pollution Research*, vol. 28, pp. 46348–46363, Oct. 2021, doi: 10.1007/s11356-021-14845-1.

[6] N. K. Verma and S. Mahajan, “Overview of Machine Learning-Based Traffic Flow Prediction,” *Transportation Research Procedia*, vol. 48, pp. 3318–3331, 2020, doi: 10.1016/j.trpro.2020.08.123.

[7] Y. Steimberg, A. Khubchandani, and B. Bulut, “Predicting Urban Air Quality Using Machine Learning Techniques,” *Cornell University Technical Report*, Dec. 2024. [Online]. Available: https://www.researchgate.net/publication/387460670.

[8] N. U. Khan, M. A. Shah, C. Maple, E. Ahmed, and N. Asghar, “Traffic Flow Prediction: An Intelligent Scheme for Forecasting Traffic Flow Using Air Pollution Data in Smart Cities with Bagging Ensemble,” *Sustainability*, vol. 14, no. 7, p. 4164, Apr. 2022, doi: 10.3390/su14074164.

[9] M. Su, H. Liu, C. Yu, and Z. Duan, “A Novel AQI Forecasting Method Based on Fusing Temporal Correlation Forecasting with Spatial Correlation Forecasting,” *Atmospheric Pollution Research*, vol. 14, no. 3, p. 101717, Mar. 2023, doi: 10.1016/j.apr.2023.101717.

[10] S. Raheja and S. Malik, “Prediction of Air Quality Using LSTM Recurrent Neural Network,” *International Journal of Software Innovation*, vol. 10, no. 1, pp. 1–17, Jan. 2022, doi: 10.4018/IJSI.297982.

[11] A. Batty, J. Milton, and K. Anders, “Navigating Urban Complexity: The Transformative Role of Digital Twins in Smart City Development,” Cities, vol. 134, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210670724004086

[12] R. Patel, A. Kumar, and L. White, “A Digital Urban Twin Enabling Interactive Pollution Predictions and Enhanced Planning,” arXiv preprint arXiv:2502.13746, 2023. [Online]. Available: https://arxiv.org/abs/2502.13746

[13] T. Thomas, H. Nguyen, M. Smith and R. Hall, “Digital Twins of Urban Air Quality: Opportunities and Challenges,” Frontiers in Sustainable Cities, vol. 3, 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frsc.2021.786563/full

[14] J. Brown, M. Green, and S. Lee, “Machine Learning Applications in Urban Traffic Prediction: A Review,” IEEE Transactions on Intelligent Transportation Systems, vol. 24, no. 3, 2023.

Appendices

Appendix A: Project Proposal

Updated Project Proposal submitted to the University and approved by supervisor, can be found at the link below:

https://github.com/manas-p-pandey/MscFinalProject/blob/main/Artifacts/A00014692_Project_Proposal-Updated.pdf

Appendix B: Project Management

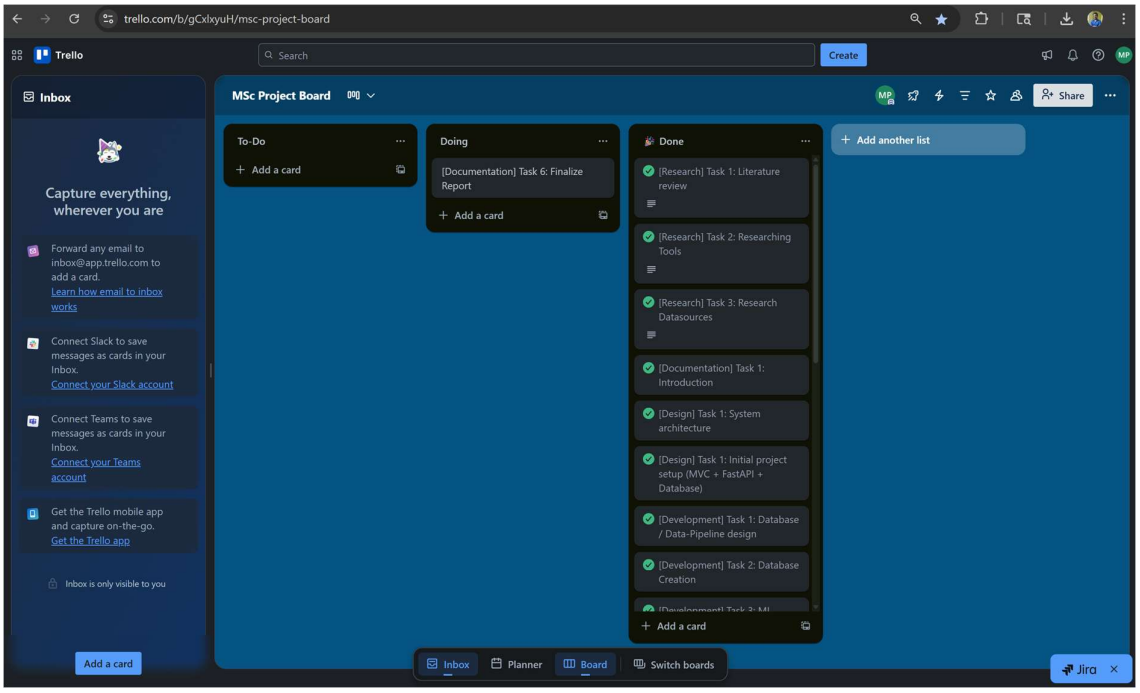
Gantt Chart for project delivery timeline is attached below

		Week1	Week2	Week3	Week4	Week5	Week6	Week7	Week8	Week9	Week10
Task Number	Task	2 Jun - 8 Jun	9 Jun - 15 Jun	16 Jun - 22 Jun	23 Jun - 29 Jun	30 Jun - 6 Jul	7 Jul - 13 Jul	14 Jul - 20 Jul	21 Jul - 27 Jul	28 Jul - 3 Aug	4 Aug - 10 Aug
1	Literature review										
2	Research Datasources										
3	Researching Tools										
1	System architecture										
2	Database / Data-Pipeline design										
3	Initial project setup (MVC + FastAPI + Database)										
1	Database Creation										
2	ML model development										
3	FASTAPI Development										
4	Scenario Simulation Engine										
5	Frontend Views Development										
6	Prediction integration with Frontend										
7	FrontEnd Dashboard										
8	Bug Fixes										
1	System Testing										
2	Usability Testing										
1	Intrioduction										
2	Literature /Technology Review										
3	Implementation										
4	Evaluation and Results										
5	Conclusion and References										
6	Finalize Report										
7	Presentation Prep										

Kanban Board screenshot and link is attached below from Trello

https://trello.com/b/gCxlxyuH/msc-project-board

Screenshot



Appendix C: Artefact/Dataset

- For Initial XGBoost Model training
    - Link initial XGBoost model creation and training python Jupyter notebook is attached below:

      https://github.com/manas-p-pandey/MscFinalProject/blob/main/Artifacts/InitialAnalysisAndModelCreation.ipynb

    - The dataset used for training and validation in the above attached script was exported from PostgresSQL mscds database into below csv file attached

      https://github.com/manas-p-pandey/MscFinalProject/raw/refs/heads/main/Artifacts/data-1753054598583.csv

- API for ingesting historical data for AQI given a location and datetime
  https://openweathermap.org/api/air-pollution
  Sample Request Endpoint:
  http://api.openweathermap.org/data/2.5/air_pollution?lat={lat}&lon={lon}&appid={API key}
- API for Weather given location and datetime
  https://openweathermap.org/api/one-call-3
  Sample Request Endpoint:
  https://api.openweathermap.org/data/3.0/onecall/timemachine?lat={lat}&lon={lon}&dt={time}&appid={API key}
- API for site data from Environmental Research Group, Imperial College, London
  https://api.erg.ic.ac.uk/AirQuality/Daily/MonitoringIndex/Latest/GroupName=%7bGROUPNAME%7d/Json
  Sample Request Endpoint:
  https://api.erg.ic.ac.uk/AirQuality/Information/MonitoringSites/GroupName=London/Json
  Filter all locations where LocalAuthorityName="Westminster"
- CesiumJS documentation
  https://cesium.com/platform/cesiumjs/

- ChartJS documentation

  https://www.chartjs.org/

Appendix D: Screencast

Below is the link to Screencast video uploaded to YouTube.

https://youtu.be/rN5oH6u9K5o

Appendix E: Sample tables from database

Table 1: site_table

site_code	site_name	site_type	local_authority_name	latitude	longitude	date_opened
NB1	Westminster - Strand (Northbank BID)	Roadside	Westminster	51.51197012	-0.116713104	28-04-2015 00:00
WMA	Westminster - Buckingham Palace Road	Roadside	Westminster	51.49323279	-0.147391757	18-02-2018 00:00
WMC	Westminster - Cavendish Square	Roadside	Westminster	51.51680165	-0.145657269	31-05-2018 12:00
WM8	Westminster - Victoria	Urban Background	Westminster	51.49706618	-0.142438821	06-01-2014 00:00
WM4	Westminster - Charing Cross Library	Roadside	Westminster	51.510205	-0.128177	01-02-2007 00:00
WM9	Westminster - Victoria (Victoria BID)	Roadside	Westminster	51.49773318	-0.144241409	08-08-2016 00:00
WM5	Westminster - Covent Garden	Urban Background	Westminster	51.51197698	-0.121627203	10-07-2009 00:00
GV2	Westminster - Duke Street (Grosvenor)	Roadside	Westminster	51.51299988	-0.150913488	22-08-2019 16:00
GV1	Westminster - Ebury Street (Grosvenor)	Roadside	Westminster	51.493492	-0.149906	11-12-2018 09:00
VS1	Westminster - Victoria Street	Kerbside	Westminster	51.49924501	-0.131285174	02-05-2003 15:00

Table 2: aqi_table

id	site_code	latitude	longitude	measurement_datetime	aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3
238961	WME	51.52866297	-0.208753656	06-08-2025 10:00	1	133.81	6.78	15.47	52.87	13.36	4.19	7.55	2.29
238959	WME	51.52866297	-0.208753656	06-08-2025 09:00	1	132.76	5.01	14.33	52.5	11.04	3.61	6.71	2.1
238957	WME	51.52866297	-0.208753656	06-08-2025 08:00	1	131.51	2.59	13.03	53.39	8.32	3.06	5.98	1.8
238955	WME	51.52866297	-0.208753656	06-08-2025 07:00	1	130.36	0.6	10.42	56.7	5.53	2.59	5.35	1.45
238953	WME	51.52866297	-0.208753656	06-08-2025 06:00	2	129.82	0.03	7.78	60.32	3.92	2.29	4.95	1.32
238951	WME	51.52866297	-0.208753656	06-08-2025 05:00	2	129.89	0	7.28	60.5	3.63	2.16	4.74	1.39
238949	WME	51.52866297	-0.208753656	06-08-2025 04:00	2	129.87	0	6.86	60.89	3.33	2.04	4.55	1.46
238947	WME	51.52866297	-0.208753656	06-08-2025 03:00	2	130.37	0	6.35	61.58	3.01	1.92	4.4	1.53
238945	WME	51.52866297	-0.208753656	06-08-2025 02:00	2	130.76	0	5.93	61.98	2.76	1.81	4.26	1.62
238943	WME	51.52866297	-0.208753656	06-08-2025 01:00	2	131.62	0	5.62	62.3	2.58	1.71	4.1	1.71

Table 3: traffic_table

id	measurement_datetime	latitude	longitude	traffic_flow	traffic_density
12737918	06-08-2025 11:00	51.49773318	-0.144241409	high	low
12737917	06-08-2025 11:00	51.5111825	-0.139114123	high	low
12737916	06-08-2025 11:00	51.51137714	-0.139249368	high	low
12737915	06-08-2025 11:00	51.49323279	-0.147391757	high	low
12737914	06-08-2025 11:00	51.494681	-0.131938	high	low
12737913	06-08-2025 11:00	51.51197698	-0.121627203	high	low
12737912	06-08-2025 11:00	51.49224823	-0.147114753	high	low
12737911	06-08-2025 11:00	51.49706618	-0.142438821	high	low
12737910	06-08-2025 11:00	51.51392874	-0.152792702	high	low
12737909	06-08-2025 11:00	51.51680165	-0.145657269	high	low

Table 4: weather_table

id	latitude	longitude	timestamp	temp	feels_like	pressure	humidity	dew_point	clouds	wind_speed	wind_deg	weather_main	weather_description
245828	51.52866297	-0.208753656	2025-08-06 10:00:00+00	293.22	292.33	1025	40	279.22	92	1.91	232	Clouds	overcast clouds
245827	51.52254	-0.15459	2025-08-06 10:00:00+00	293.24	292.32	1025	39	278.87	93	1.89	234	Clouds	overcast clouds
245826	51.52250936	-0.154622155	2025-08-06 10:00:00+00	293.24	292.32	1025	39	278.87	93	1.89	234	Clouds	overcast clouds
245825	51.51680165	-0.145657269	2025-08-06 10:00:00+00	293.24	292.32	1025	39	278.87	93	1.89	234	Clouds	overcast clouds
245824	51.516066	-0.13516388	2025-08-06 10:00:00+00	293.25	292.33	1025	39	278.88	93	1.89	235	Clouds	overcast clouds
245823	51.51392874	-0.152792702	2025-08-06 10:00:00+00	293.28	292.37	1025	39	278.91	93	1.87	234	Clouds	overcast clouds
245822	51.51299988	-0.150913488	2025-08-06 10:00:00+00	293.28	292.37	1025	39	278.91	93	1.87	234	Clouds	overcast clouds
245821	51.51197698	-0.121627203	2025-08-06 10:00:00+00	293.31	292.4	1025	39	278.93	93	1.88	235	Clouds	overcast clouds
245820	51.51197012	-0.116713104	2025-08-06 10:00:00+00	293.31	292.4	1025	39	278.93	93	1.88	235	Clouds	overcast clouds
245819	51.51137714	-0.139249368	2025-08-06 10:00:00+00	293.26	292.34	1025	39	278.89	93	1.86	234	Clouds	overcast clouds

Table 5: synthetic_lstm_stats

id	model_name	test_loss	test_accuracy	created_at
137	lstm_traffic_model	0.5354910492897034	0.7899807095527649	2025-08-06 10:59:03.763805
136	lstm_traffic_model	0.5262933969497681	0.7944765686988831	2025-08-06 09:32:59.484841
135	lstm_traffic_model	0.5271182060241699	0.7802749276161194	2025-08-05 20:11:21.869075
134	lstm_traffic_model	0.5425169467926025	0.78529346	2025-08-05 19:11:12.194004
133	lstm_traffic_model	0.5514461994171143	0.7820568680763245	2025-08-05 18:41:22.92981
132	lstm_traffic_model	0.5172176361083984	0.7849015593528748	2025-08-05 17:42:47.22063
131	lstm_traffic_model	0.5230606198310852	0.7856045365333557	2025-08-05 16:41:56.019221
130	lstm_traffic_model	0.53935945	0.7759490609169006	2025-08-05 15:12:48.152701
129	lstm_traffic_model	0.5318884253501892	0.7828382849693298	2025-08-05 14:11:03.221217
128	lstm_traffic_model	0.5175583958625793	0.7896039485931396	2025-08-04 21:10:43.519278

Table 6: regressor_stats

model	rmse	mae	r2	created_at
nh3	0.41	0.25	0.88	2025-08-06 11:03:07.316806
pm10	4.14	2.75	0.89	2025-08-06 11:02:34.154448
pm2_5	3.67	2.31	0.91	2025-08-06 11:01:41.214048
so2	3.17	1.87	0.87	2025-08-06 11:01:18.796093
o3	10.71	8.09	0.88	2025-08-06 11:00:55.828196
no2	12.71	5.20	0.99	2025-08-06 11:00:19.618367
no	8.98	3.82	0.89	2025-08-06 10:59:59.085587
co	30.99	18.57	0.93	2025-08-06 10:59:48.078555
aqi	0.36	0.28	0.74	2025-08-06 10:59:38.078745
nh3	0.41	0.25	0.88	2025-08-06 09:36:38.024728

--- END ---