Data Streams Quest: Your Mission to Build, Deploy, and Monitor AI in the Wild

Team Name: RHYMe

Team Members:

- Hemant Sathish (hsathish)
- Meghna Nair (meghnan)
- Reuben Mathew (rgmathew)
- Yash Thakkar (ythakkar)

Executive Summary

Air quality is one of the most pressing challenges faced by urban environments. Poor air quality not only undermines public health by causing respiratory and cardiovascular diseases, but also carries significant economic and regulatory costs for governments and businesses. Timely monitoring and accurate forecasting of pollutant levels are therefore critical for enabling proactive interventions, compliance with environmental regulations, and informed long-term urban planning.

This project addresses that need by designing and implementing a real-time streaming and predictive analytics platform for environmental data. Leveraging the UCI Air Quality dataset as a proxy for sensor streams, the system ingests and processes data through a containerized microservices architecture built on Apache Kafka for message streaming, FastAPI for model inference, and Evidently AI for continuous drift detection and performance monitoring. The platform simulates production-grade machine learning operations by streaming raw sensor readings row-by-row, engineering over 100 time-dependent features on-the-fly, generating real-time predictions for carbon monoxide (CO) levels, and autonomously detecting data drift and model degradation through statistical analysis against a validated baseline.

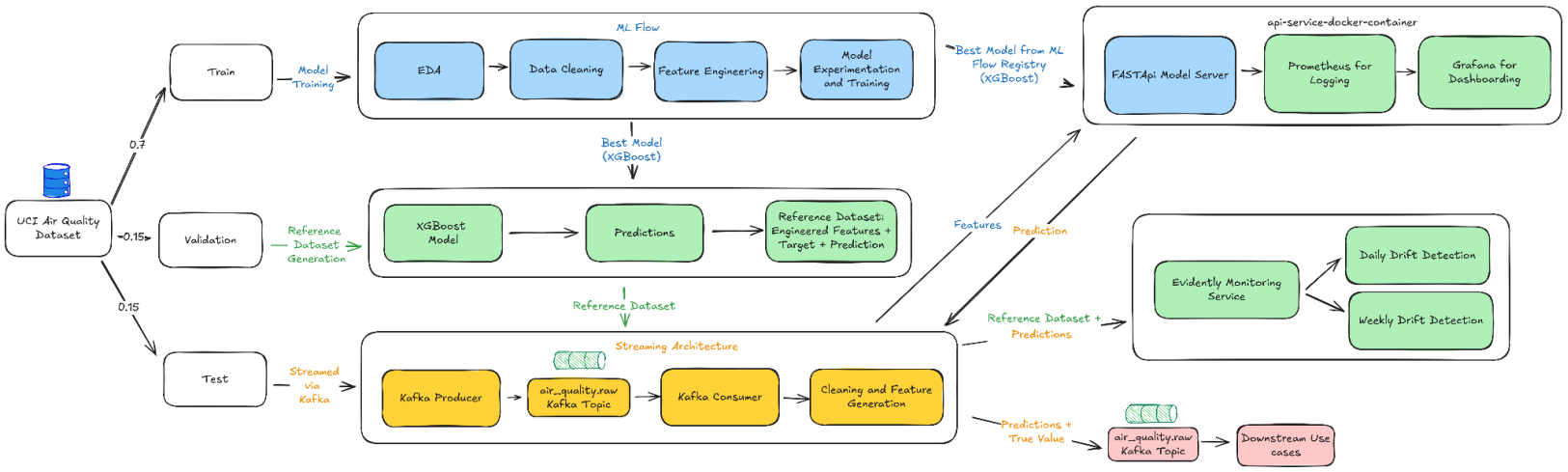
The predictive modeling phase transitioned from the simpler baselines created for Homework 1 (SARIMA and naïve persistence) to a multi-model ensemble learning framework combining classical and gradient-boosted regressors. Six supervised models were developed and tuned using Bayesian Optimization (BayesSearchCV) with TimeSeriesSplit cross-validation to prevent data leakage:

- Random Forest Regressor
- Extra Trees Regressor
- Gradient Boosting Regressor
- **XGBoost Regressor (Best)**
- LightGBM Regressor
- CatBoost Regressor

The final production model was selected based on superior performance on validation metrics (RMSE, MAE, R^2, MAPE, SMAPE, MASE) and deployed within a seven-container architecture spanning two Docker Compose deployments. The system implements comprehensive observability through multiple layers, with Prometheus collecting operational metrics every 15 seconds, Grafana providing real-time dashboards, and Evidently generating automated daily and weekly HTML reports analyzing data drift, prediction drift, and model performance degradation.

This end-to-end platform demonstrates a production-ready approach to streaming machine learning, combining robust feature engineering, rigorous model selection, real-time inference with a 168-row warmup buffer for temporal feature generation, and continuous monitoring to provide a blueprint for deploying and maintaining ML models in dynamic, time-sensitive domains such as environmental monitoring, predictive maintenance, and IoT analytics.

System Architecture & Deployment



This project implements a production-grade, microservices-based ML monitoring system for real-time air quality prediction. The architecture is designed with containerization and scalability in mind, separating concerns between model inference, data streaming, monitoring, and observability components. The system leverages Docker containers orchestrated through two independent Docker Compose configurations to achieve modularity and deployment flexibility.

Architectural Components

The system comprises seven containerized services organized into two independent deployment units:

<u>Deployment Unit 1: API Service & Observability Stack</u>

This deployment unit consists of six containers hosting the core inference service, Kafka infrastructure, and observability stack:

Container 1: FastAPI Model Server (Port 8000)

- Serves the XGBoost model for CO(GT) prediction
- Exposes /predict endpoint for inference requests
- Handles 100+ engineered features from streaming data
- Loads production model from artifacts/model.pkl
- Exports Prometheus metrics via /metrics endpoint
- Built from custom Dockerfile with Python 3.11 runtime

Container 2: Zookeeper (Port 2181)

- Kafka cluster coordination and metadata management
- Maintains broker membership and topic configurations
- Image: confluentinc/cp-zookeeper

Container 3: Kafka Broker (Port 9092)

- Message streaming backbone for the pipeline
- Topic: air_quality.raw - Raw sensor data ingestion
- Topic: air_quality.pred - Model predictions output
- Configured with single-broker setup (development)
- Image: confluentinc/cp-kafka

Container 4: Kafka UI (Port 8080)

- Web-based interface for Kafka monitoring
- Visualizes topic throughput, consumer lag, message inspection
- Accessible at http://localhost:8080
- Image: provectuslabs/kafka-ui

Container 5: Prometheus (Port 9090)

- Time-series metrics collection from FastAPI
- Scrapes /metrics endpoint every 15 seconds
- Tracks: latency, throughput, error rates, prediction values
- Configuration mounted from prometheus.yml
- Image: prom/prometheus

Container 6: Grafana (Port 3000)

- Real-time metrics visualization dashboards
- Pre-configured Prometheus datasource via grafana/datasources.yml
- Monitors: prediction latency, throughput, missing values, errors
- Default credentials: admin/admin
- Image: grafana/grafana

All six containers are managed by a single docker-compose.yml file with dependency management ensuring proper startup order: Zookeeper, Kafka, Kafka UI, API, Prometheus, Grafana.

Deployment Unit 2: Streaming & Monitoring Stack

This deployment unit consists of one container encapsulating three services for data streaming and drift monitoring:

Container 7: Streaming Services

This container runs three Python services as separate processes:

Service 1: Kafka Producer

- Reads raw test data from test_data_raw.csv
- Streams sensor readings row-by-row to Kafka topic air_quality.raw
- Simulates real-time data ingestion (1 row = 1 hour of readings)
- Configurable streaming delay for throughput control
- Script: producer.py

Service 2: Kafka Consumer

- Subscribes to Kafka topic air_quality.raw
- Implements 168-row warmup buffer for lag/rolling feature generation
- Engineers 100+ features (lag, rolling stats, interactions, time encodings)
- Calls FastAPI /predict endpoint via HTTP
- Publishes predictions to Kafka topic air_quality.pred
- Triggers Evidently monitoring service after each prediction
- Script: consumer.py

Service 3: Evidently Monitoring Service

- Standalone drift detection and model performance monitoring
- Compares streaming predictions against reference dataset baseline
- Detects data drift, prediction drift, and performance degradation
- Generates HTML reports:

    - Daily Reports: Every 24 predictions (24-hour windows)
    - Weekly Reports: Every 168 predictions (7-day windows)
- Outputs stored in mounted reports/ volume
- Script: monitoring_service.py

Data Flow Architecture

The system implements a comprehensive data pipeline from raw sensor readings to monitored predictions:

1. **Model Training Phase (Databricks ML Flow):**

   The model training phase was conducted on the Databricks platform with dedicated compute cluster resources. The process began with exploratory data analysis (EDA) and data cleaning on the UCI Air Quality dataset, followed by the development of a comprehensive feature engineering pipeline that created over 100 engineered features.

   These features included lag features at 1, 2, 3, 6, 12, 24, 48, and 72-hour intervals, rolling statistics (mean, standard deviation, minimum, and maximum) calculated over multiple time windows, rate of change metrics such as first differences and acceleration, pollutant interactions through cross-products and ratios, and time encodings using cyclical transformations (sine/cosine) for hour, day, and month.

   Model experimentation was tracked using MLflow across 19 experimental runs that tested various algorithms including tree-based models (RandomForest, ExtraTrees, GradientBoosting), gradient boosting variants (XGBoost, LightGBM, CatBoost), and ensemble methods employing weighted averaging and stacking.

   Hyperparameter optimization was performed using Bayesian optimization via scikit-optimize, with early stopping and cross-validation on the validation set to prevent overfitting. The best-performing model, XGBoost, was promoted to the production registry, served with a TransformedTargetRegressor applying log transformation. Performance was evaluated using multiple metrics including RMSE, MAE, R^2, MAPE, SMAPE, and MASE.

   Databricks was selected as the training platform due to its integrated MLflow tracking and model registry capabilities, scalable compute resources ideal for hyperparameter optimization, collaborative notebook environment enabling reproducible experiments, and seamless model versioning and deployment workflow that streamlined the transition from experimentation to production.

2. **Reference Dataset Generation**

   The reference dataset generation phase utilizes the generate_reference_dataset.py script to create a baseline for drift detection. This script loads the validation data with pre-engineered features from eval_data_engineered.csv and executes batch predictions using the trained XGBoost model.

   The output is saved as reference.csv, which contains the complete set of engineered features spanning over 100 columns, the true target values for CO(GT), and the corresponding model predictions. This reference dataset serves as the baseline for Evidently's drift detection system, enabling comparison between the known-good validation data distribution and the streaming production data to identify potential data drift, prediction drift, and model performance degradation over time.

3. Real-Time Inference Pipeline

Data Ingestion

The data ingestion process begins with the test dataset stored in test_data_raw.csv, which is read by the Kafka Producer running in Container 7. The producer streams raw sensor data row-by-row to the Kafka topic named air_quality.raw hosted on Container 3, where each row represents one hour of sensor readings. The raw data streamed at this stage contains no DateTime column or engineered features, consisting solely of the original sensor measurements.

Feature Engineering & Warmup

The Kafka Consumer in Container 7 subscribes to the air_quality.raw topic and implements a feature engineering warmup phase. During this phase, the consumer buffers the first 168 rows (equivalent to 7 days of hourly data) to build sufficient historical context for generating lag features and rolling statistics. No predictions are generated during this warmup phase, as the system requires this historical window to compute the time-dependent features that the model expects.

Inference & Publishing

Once the feature buffer contains at least 168 rows, the consumer begins the inference phase. For each new incoming row, the consumer engineers the full set of 100+ features using the buffered historical data, then sends these features via an HTTP request to the FastAPI /predict endpoint running in Container 1. The FastAPI service loads the XGBoost model and returns a prediction for CO(GT). The consumer then publishes this prediction to the Kafka topic air_quality.pred on Container 3, making the results available for downstream consumption and analysis.

Drift Monitoring

After each prediction is generated, the consumer forwards the prediction along with the engineered features to the Evidently Monitoring Service running within Container 7. The monitoring service compares the current data against the reference dataset baseline to detect three types of issues: data drift (feature distribution shifts identified through statistical tests like Kolmogorov-Smirnov and Chi-squared), prediction drift (changes in model output distribution), and performance degradation (measured through RMSE, MAE, and R^2 on streaming predictions compared to ground truth). The service generates interactive HTML reports at regular intervals- daily reports every 24 predictions and weekly reports every 168 predictions providing comprehensive visibility into model health and data quality.

Observability

The observability pipeline begins with the FastAPI service exporting operational metrics via its /metrics endpoint, including latency, throughput, error rates, and prediction values. Prometheus, running in Container 5, scrapes these metrics every 15 seconds and stores them in its time-series database. Grafana, running in Container 6, queries Prometheus and visualizes the metrics in real-time dashboards, providing immediate visibility into system health and enabling rapid detection of performance issues or anomalies in the prediction pipeline

Future Production Enhancements

To enhance scalability and resilience, the Kafka infrastructure can be expanded into a multi-broker cluster with a replication factor of three to ensure fault tolerance and high availability. Topics would be partitioned to enable parallel message consumption, while a dedicated ZooKeeper ensemble of three or more nodes would coordinate cluster metadata and leader elections. On the API side, scaling would involve deploying multiple FastAPI instances behind a load balancer such as Nginx or HAProxy, and orchestrating them using Kubernetes with horizontal pod autoscaling (HPA). A Redis cache layer would be introduced to store model

artifacts and frequently accessed data for faster inference responses. Similarly, the Kafka consumer architecture would use consumer groups with multiple consumers operating in parallel to process topic partitions, supported by a distributed feature buffer (Redis or Memcached) for intermediate data sharing.

For monitoring and reliability, a Prometheus federation setup would aggregate metrics from multiple clusters, with Thanos or Cortex providing long-term metric storage and scalability. A high-availability Grafana deployment would visualize real-time metrics and system health. The overall infrastructure would transition from a single-node Docker Compose setup to a Kubernetes-based cloud-native deployment, leveraging managed services for Kafka and Prometheus to reduce maintenance overhead. Finally, a CI/CD pipeline would automate the build, test, and deployment process, ensuring consistent, reproducible, and continuous delivery across environments.

Data Intelligence and Pattern Analysis

Data Quality and Missingness

The dataset spans from March 2004 to April 2005, and includes 9,357 hourly observations. Missingness was non-trivial for CO and NOx (18.0% and 17.5%, respectively), while Benzene had only 3.9% missing values.

A business implication of this is that sensor reliability directly impacts forecasting accuracy. For modeling, imputation or lag-based filling methods will be required, especially for CO and NOx. Benzene provides a relatively stable series and can serve as a predictive proxy when other sensors fail.

Temporal Patterns

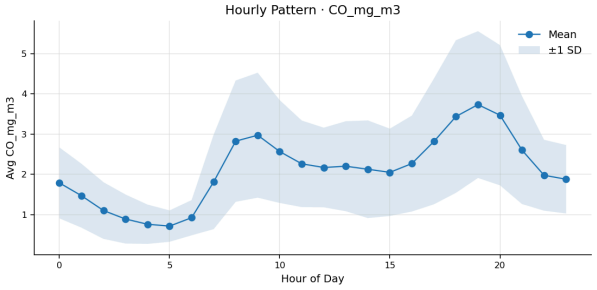
Daily Cycles

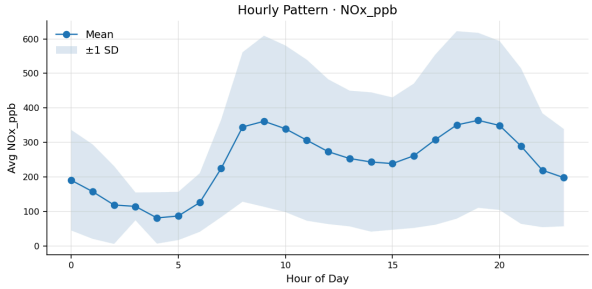
All three pollutants exhibit strong diurnal patterns with two distinct peaks:

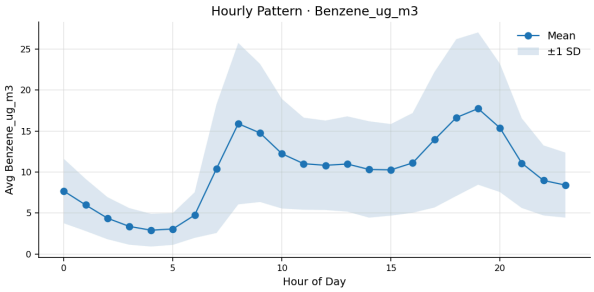
A morning peak around 08:00–09:00, coinciding with the rush-hour commute to work and school.

A larger evening peak around 18:00–20:00, as people return home.

Concentrations are at their lowest between 04:00–05:00, when traffic activity is minimal and emissions are at their daily trough. This bimodal cycle closely reflects expected urban traffic behavior and highlights the strong link between human mobility patterns and air quality.

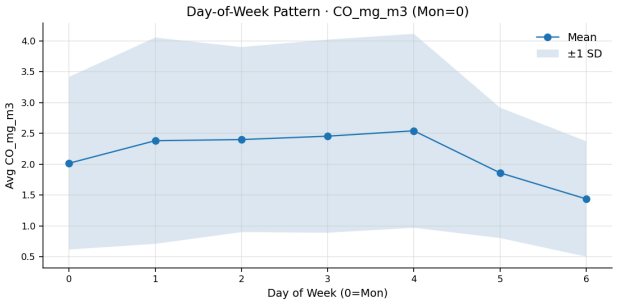


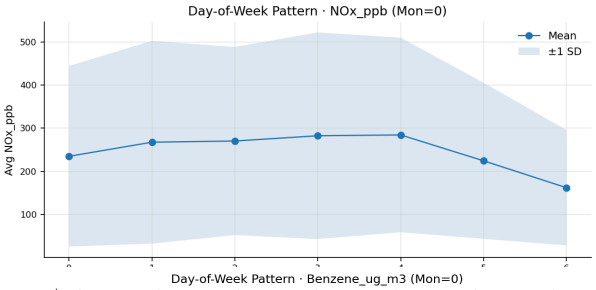


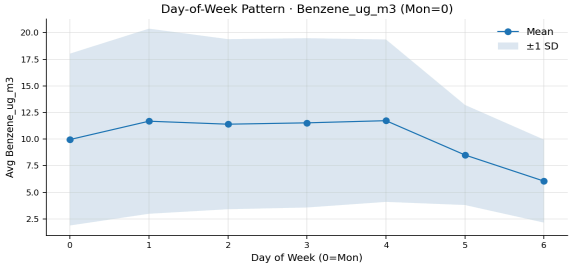


Weekly Cycles

Pollutant levels are consistently higher Monday through Friday, with reductions over the weekend. The highest concentrations occur on Thursdays, with the lowest on Sundays. This implies that human activity, particularly commuting and industrial schedules, is the dominant driver.

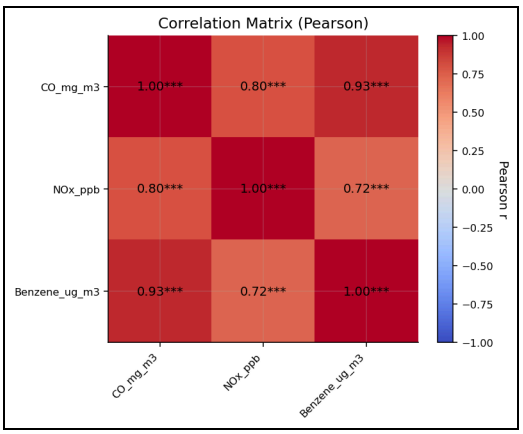






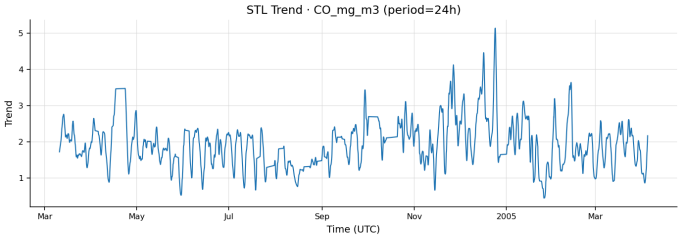
Correlation and Dependencies

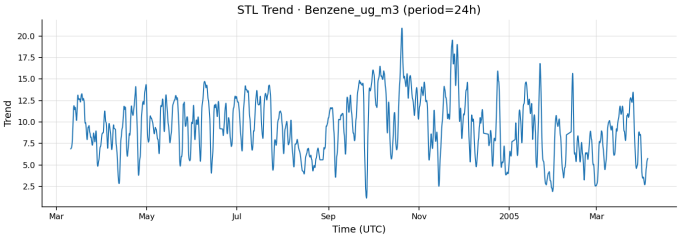
Pairwise correlations show strong interdependence:
CO vs NOx: r = 0.80
CO vs Benzene: r = 0.93
NOx vs Benzene: r = 0.72
All correlations are statistically significant ($p < 0.001$). This suggests that pollutants share common combustion-related emission sources such as vehicles and heating systems.

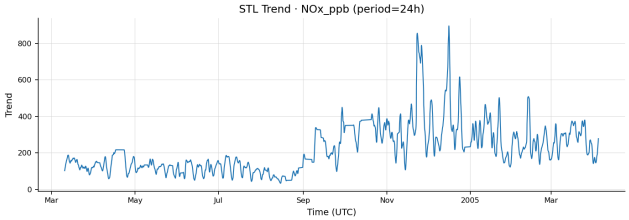


Seasonal and Trend Analysis

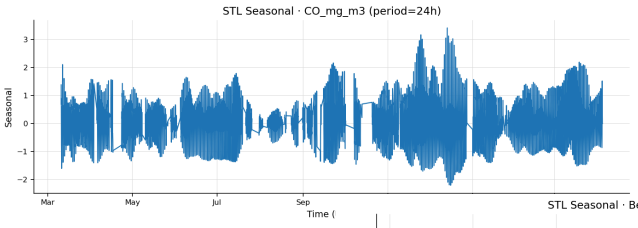
STL decomposition highlights three key elements:
Trend: The long-term trend for NOx, and to a lesser extent CO and Benzene, shows elevated concentrations during the colder months. This is consistent with increased heating demand in winter. Outside of winter, the trend is relatively

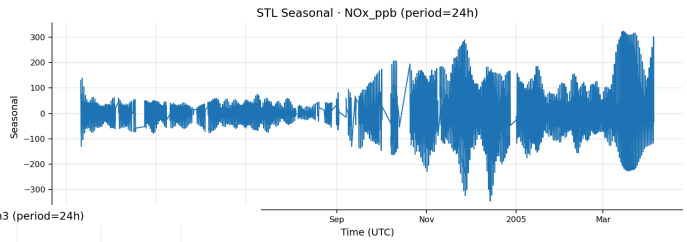


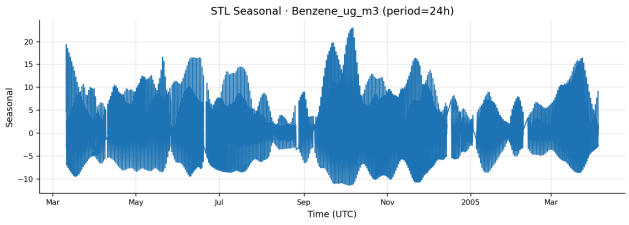




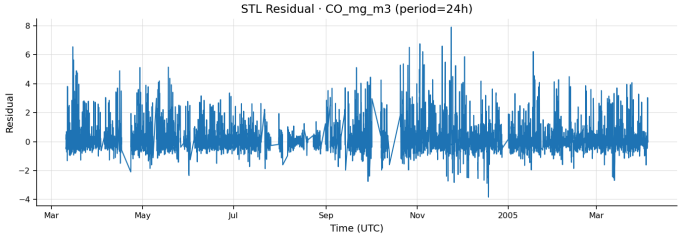
Seasonal Component: A strong daily cycle is visible across all pollutants, corresponding to traffic-related emissions. This cyclical component captures the regular morning and evening rush-hour peaks.

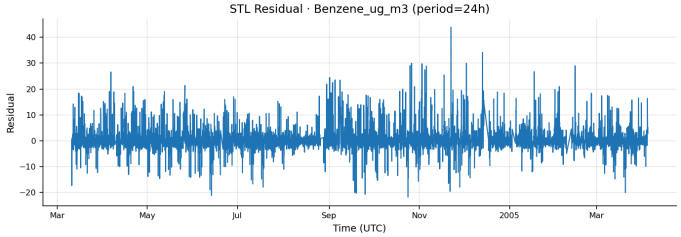






Residuals: The residual series contains short-lived spikes not explained by trend or seasonality. These may represent brief pollution episodes or sensor fluctuations.





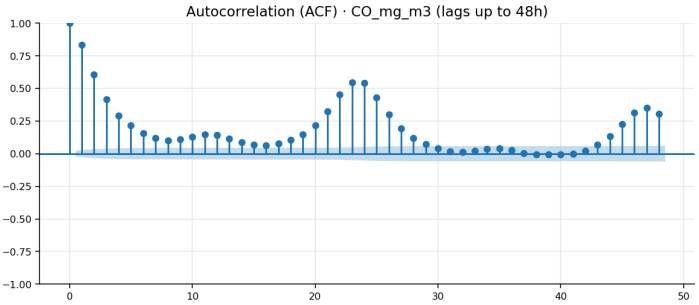
Autocorrelation and Partial Autocorrelation

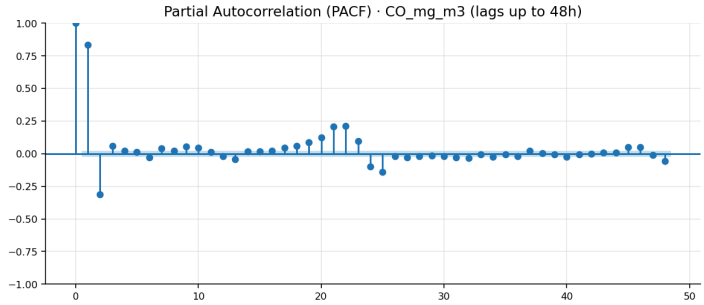
Autocorrelation (ACF) and partial autocorrelation (PACF) confirm significant persistence:

Short lags (1–3 hours): Strong positive autocorrelation, meaning pollutant levels at one hour are strongly predictive of the next few hours.

Daily cycle (24 hours): Pronounced secondary peaks reflect the repeating diurnal traffic-driven cycle.

Extended influence: Dependencies remain detectable up to 48 hours, indicating multi-day carryover effects.





Anomaly Detection

Following preprocessing, no major anomalies were identified in the cleaned dataset. This outcome suggests that the data quality procedures, particularly the handling of sentinel values such as -200, were effective in filtering out faulty sensor readings and outliers.

Even though anomalies were not present in the historical dataset, anomaly detection remains a valuable tool in a real-time deployment context. It can provide early-warning signals for extreme pollution episodes, enabling timely public health advisories, and sensor malfunctions or drift, ensuring system reliability and data integrity.

Model Experimentation Results

The objective of model experimentation was to identify a reliable and interpretable regression model capable of forecasting carbon-monoxide concentration (CO(GT)) from multivariate sensor data.

All experiments were conducted using the UCI Air Quality dataset, following time-aware validation and full lifecycle tracking in MLflow.

Data Preparation

The raw CSV contained hourly readings of gaseous pollutants and meteorological variables. Data exploration was performed and our findings are as seen in the section above. We first combined Date + Time into a single timestamp, removed invalid or missing sensor columns, and sorted chronologically. We then performed time-based interpolation and replaced −200 sentinel values, ensuring continuity in sensor readings. Numerical columns remained unscaled since all base learners were tree-based and invariant to monotone transformations. These steps yielded a continuous, stationary-ready dataset for feature generation.

Feature Engineering

To improve predictive performance, extensive feature transformations were applied to capture both short-term temporal dependencies and broader environmental effects. Lag features of the target variable, CO(GT), were generated across multiple horizons to enable the model to learn autoregressive patterns. Rolling window statistics were computed over window sizes ranging from 3 to 168 hours, capturing smooth temporal trends and variability. Rate-of-change features such as one-hour, three-hour, and twenty-four-hour differences for key pollutants were included to model short-term shifts and transients in pollution concentration.

Interaction features between pollutant sensors were constructed to capture cross-sensor dynamics indicative of complex chemical interactions in air quality systems. Environmental interactions such as temperature–humidity (T×AH) and quadratic terms (T^2, AH^2) were added to account for nonlinear relationships. Temporal context was introduced through cyclical encodings of the hour, day, and month to capture daily and seasonal periodicity, while categorical flags such as weekend, rush hour, night-time, and seasonal indicators were included to encode contextual behavioral effects. After removing rows with missing lag-derived values, the resulting dataset contained 127 engineered features.

Category	Examples	Purpose
Lag Features	CO(GT)_lag_{1,2,3,6,12,24,48,72}	Model recent history and autoregressive effects
Rolling Statistics	Mean / Std / Min / Max over {3, 6, 12, 24, 48, 168 h}	Capture short and long-term trends
Rate-of-changes	1 h, 3 h, 24 h differences of pollutant sensors	Detect sudden spikes or decays
Cross-Pollutant interactions	Pairwise products & ratios	Encode chemical/physical coupling
Environmental effects	T×AH, T^2, AH^2, T×RH	Capture nonlinear weather influences
Temporal context	Hour, Day, Month, Week, sin/cos encodings	Preserve cyclical daily / seasonal patterns
Binary Flags	Weekend, Rush hour, Night, Winter, Summer	Represent categorical time patterns

Experimental Setup

The target variable for prediction was CO(GT), representing hourly average CO concentration. The dataset was chronologically split into 70% training, 15% validation, and 15% test sets to ensure no temporal leakage. All models were trained and validated using a 7-fold TimeSeriesSplit cross-validation scheme. A TransformedTargetRegressor wrapper applied a log transform (log1p to expm1) to stabilize variance where the target values were non-negative. Evaluation metrics included Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R^2, Symmetric Mean Absolute Percentage Error (SMAPE), and Mean Absolute Scaled Error (MASE), all chosen to balance interpretability and robustness across scales. Databricks was used to train the models, and MLflow tracked every experiment, storing configurations, parameters, and artifacts for each model, ensuring experiment lineage could be traced across the pipeline.

Model Portfolio and Bayesian Optimization

A diverse portfolio of regression models was trained using Bayesian Optimization (BayesSearchCV) for hyperparameter tuning. Each model was evaluated using time-aware cross-validation to avoid overfitting and to reflect real-world sequential forecasting conditions.

Model	Description	Key Tuned Parameters
Random Forests	Bagging of decorrelated trees for robust baselines	n_estimators, max_depth, max_features, min_samples_split
Extra Trees	Randomized split selection to reduce variance	n_estimators, max_depth, min_samples_leaf
Gradient Boosting	Sequential boosting with shrinkage and subsampling	learning_rate, max_depth, subsample, n_estimators
LightGBM (LGB)	Leaf-wise boosting optimized for large feature sets	num_leaves, max_depth, learning_rate, subsample
CatBoost (CAT)	Ordered boosting with robust handling of imbalance	depth, iterations, learning_rate, l2_leaf_reg
XGBoost (XGB)	Optimized gradient boosting with regularization	max_depth, learning_rate, subsample, colsample_bytree, min_child_weight

Each model underwent 40–50 Bayesian search iterations, automatically exploring parameter spaces with skopt and identifying optimal configurations that minimized cross-validated RMSE. All model configurations, metrics, and validation predictions were logged to MLflow.

XGBoost Refinement

Given its consistently superior validation performance, XGBoost was selected for an additional fine-tuning stage. After the Bayesian search identified the best hyperparameters (best_p), the model was retrained using early stopping to determine the optimal number of trees (best_n). This was implemented with XGBoost's DMatrix API, monitoring validation RMSE and halting after 50 non-improving rounds to prevent overfitting. Finally, the model was refitted on the combined training and validation data using both best_p and best_n, and logged to MLflow with its inferred input–output signature for seamless deployment. This two-phase optimization approach allowed the model end up performing the best among all our trained models.

Ensemble Learning

To further improve predictive robustness, two ensemble strategies were implemented.

- Weighted Ensemble: Predictions from all six base models (RF, ET, GB, LGB, CAT, XGB) were combined linearly, with optimal weights determined by minimizing validation RMSE using constrained optimization (SLSQP), enforcing $\sum w_i = 1$.

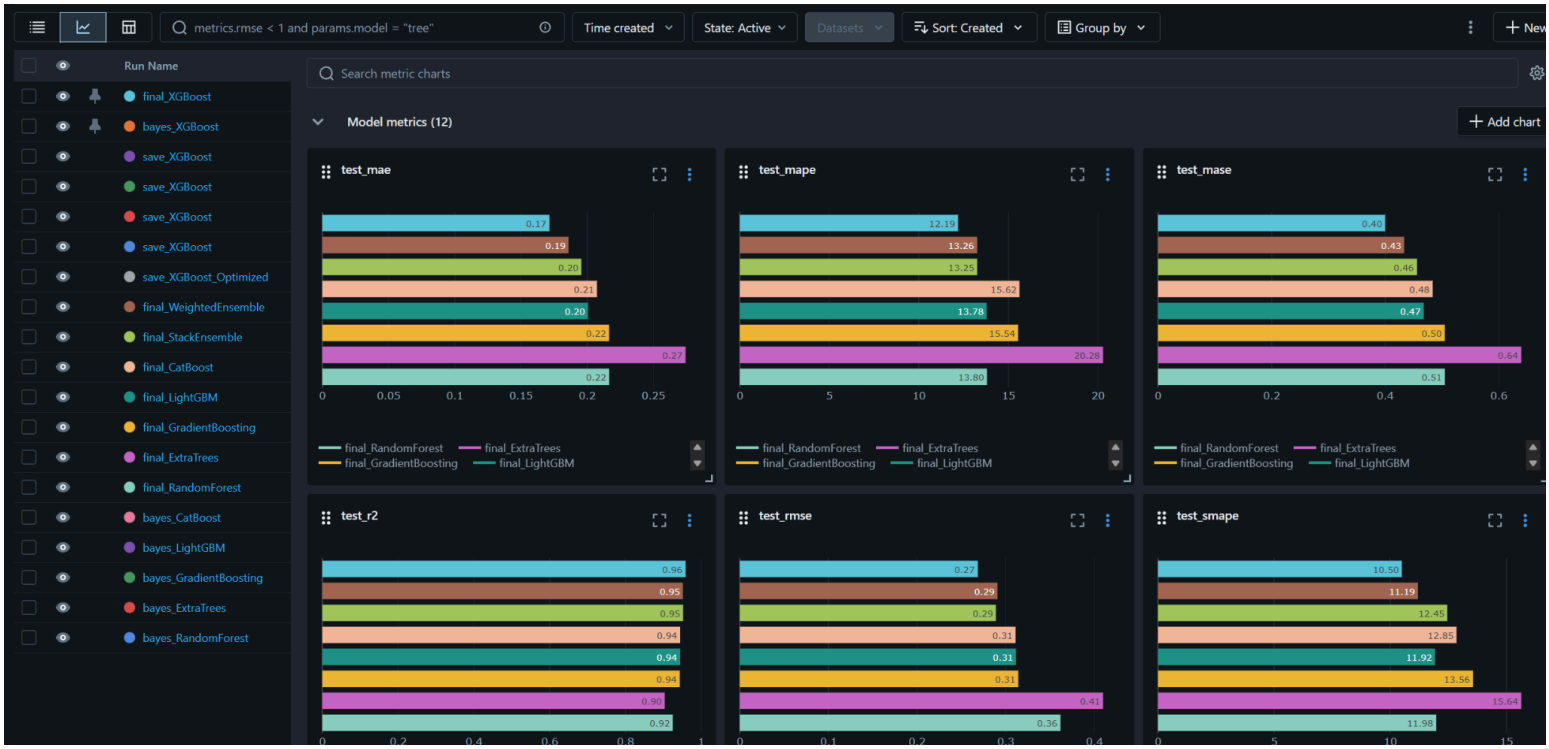
- Stacking Ensemble: The second approach used a Ridge Regression meta-learner to combine base model outputs dynamically. Each ensemble configuration was treated as a separate MLflow experiment, with metrics and artifacts logged for later analysis.

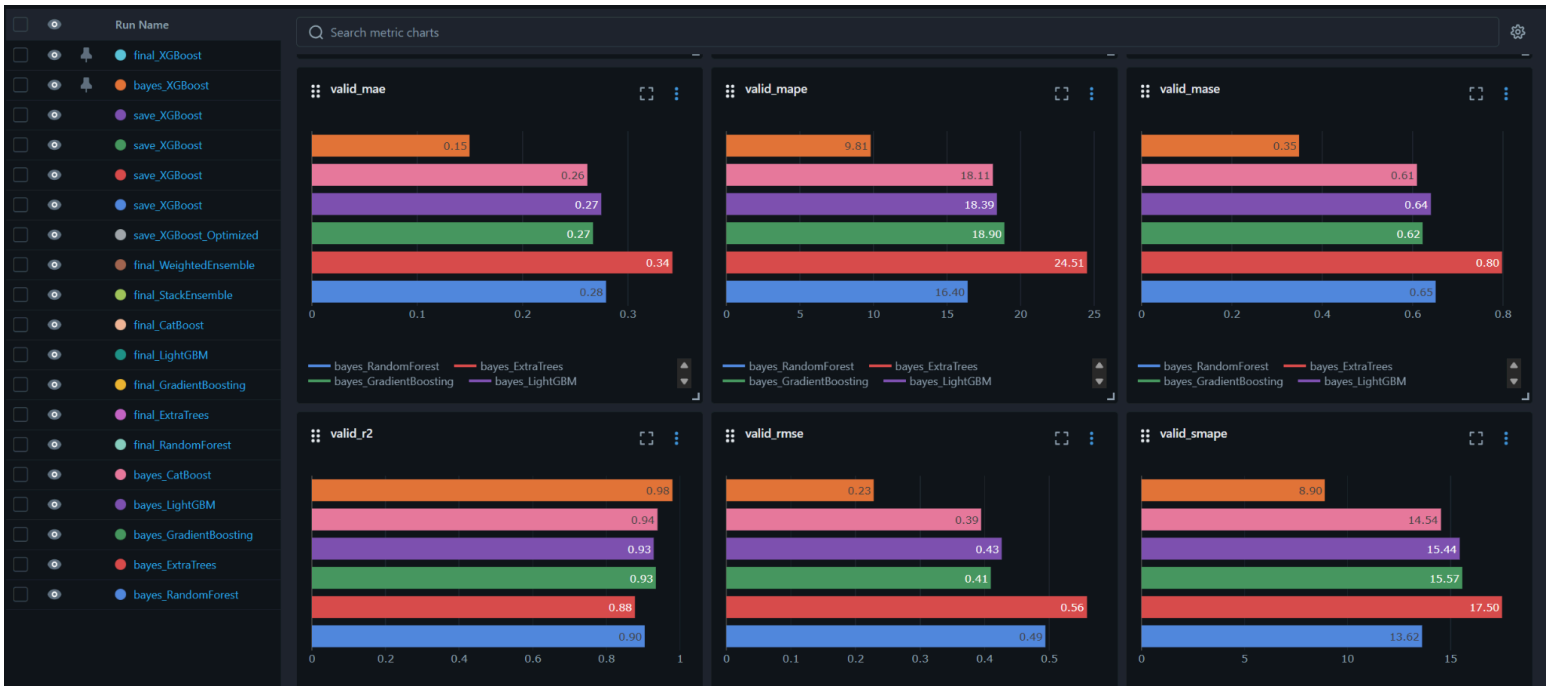
Results

Model	RMSE	MAE	R^2	SMAPE
Random Forests	0.3618	0.2166	0.9249	11.978%
Extra Trees	0.4099	0.2743	0.9036	15.636%
Gradient Boosting	0.3142	0.2165	0.9434	13.561%
XGBoost (XGB)	0.2686	0.1715	0.9586	10.498%
LightGBM (LGB)	0.3115	0.2006	0.9443	11.923%
CatBoost (CAT)	0.3111	0.2074	0.9445	12.848%
StackEnsemble	0.2889	0.1956	0.9521	12.454%
WeightedEnsemble	0.2907	0.1859	0.9515	11.185%

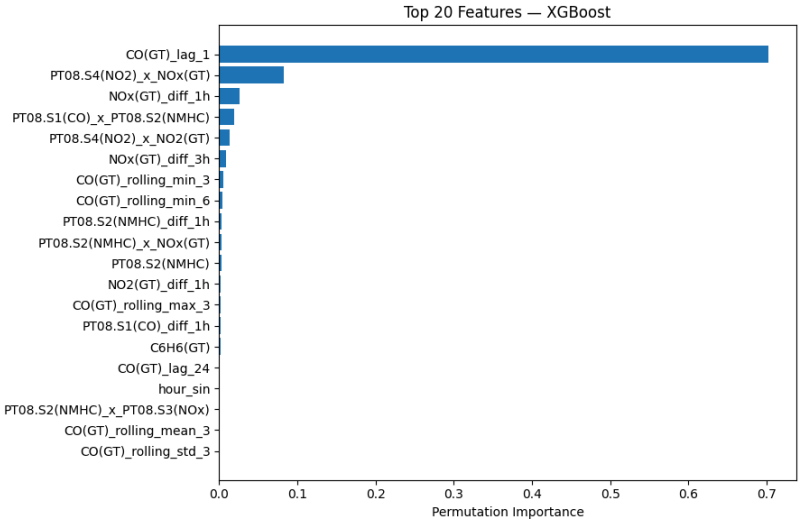
The XGBoost model achieved the best single-model performance, demonstrating excellent fit and generalization. Both ensemble approaches yielded near-identical performance, confirming the strong agreement among tuned base models. Based on the results, the XGBoost model with early stopping was promoted to Production within the MLflow Model Registry.

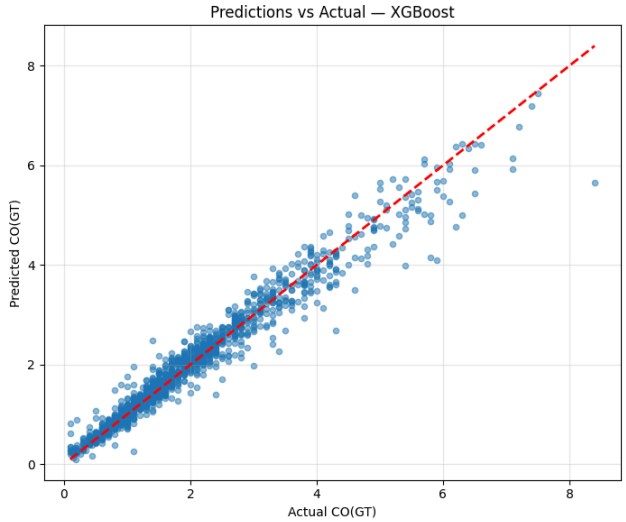
MLFlow Training Plots





Interpretability and Diagnostics





Model interpretability was examined using permutation importance and predicted vs. actual analysis.
The top predictors for the final XGBoost model are shown above. The most influential feature, CO(GT)_lag_1, confirms that recent CO levels dominate short-term forecasts. Other important features included cross-pollutant interactions (PT08.S4(NO2)_x_NOx(GT), PT08.S1(CO)_x_PT08.S2(NMHC)), pollutant deltas (NOx(GT)_diff_1h, NO2(GT)_diff_3h), and rolling minima and means over 3–6 hours, all capturing local temporal and chemical dynamics. Cyclical encodings such as hour_sin also appeared in the top 20, reflecting the model's sensitivity to daily traffic and meteorological cycles.

The Predicted vs Actual plot shows a strong linear relationship along the 45° reference line, with minimal spread, indicating well-calibrated predictions. Only minor underestimations were observed at very high CO values (>6 mg/m³), likely due to data sparsity at extreme pollution levels. Overall, the XGBoost model exhibited strong calibration, low bias, and consistent predictive behavior across the full concentration range.

Monitoring: Evidently

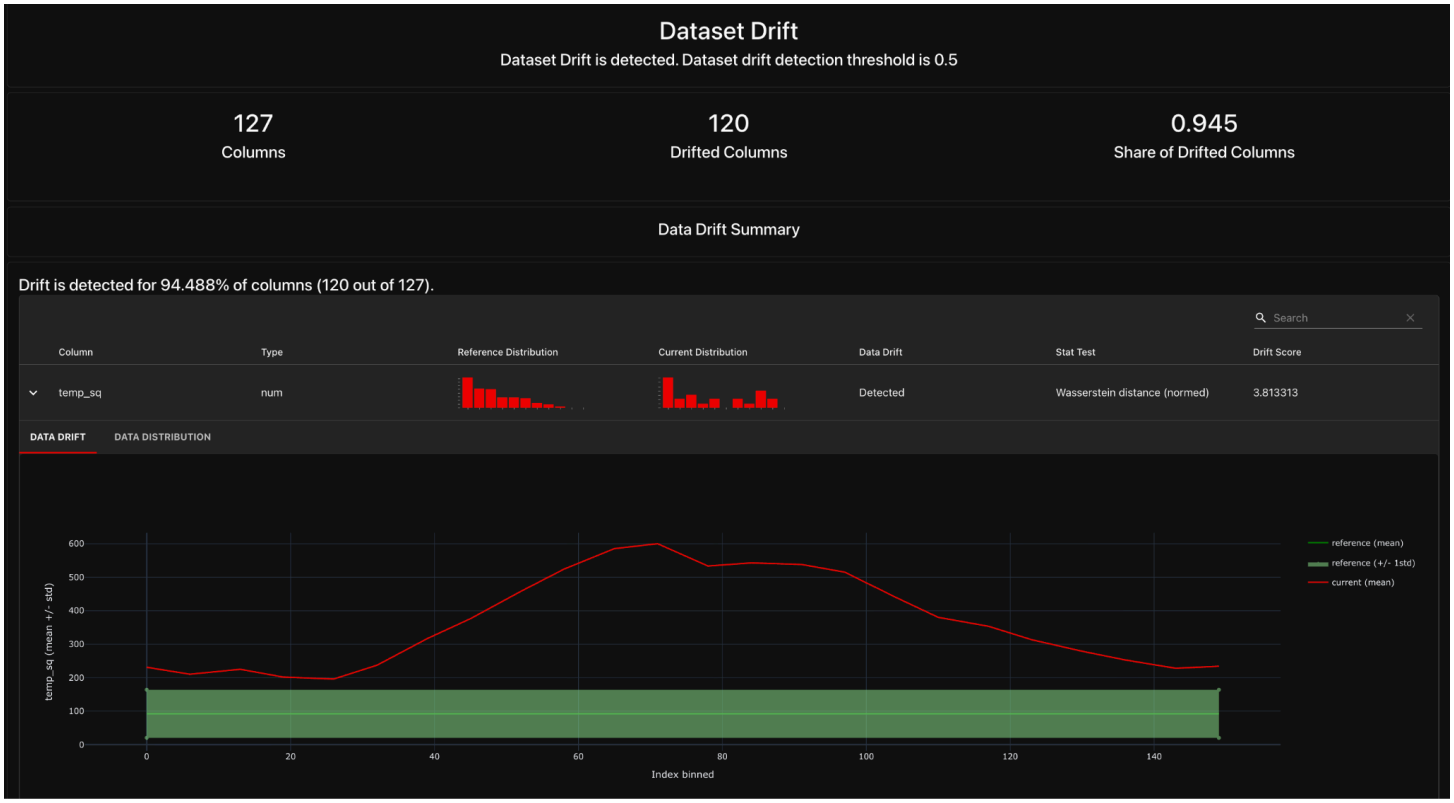
To ensure data and model reliability in a production-like streaming setup, the system integrates Evidently, an open-source library for continuous data and model monitoring.

Evidently compares live Kafka-streamed air-quality data against a reference baseline dataset generated from stable historical readings. The monitoring service runs daily and weekly drift analyses to assess both data quality and model performance stability over time.

Daily Report

Generated every 24 hours, these reports capture short-term changes in feature distributions. Evidently calculates statistical drift tests (e.g., Kolmogorov–Smirnov, Chi-square) and summarizes the share of features that have drifted.

The daily report highlights issues such as missing values, out-of-range measurements, and sensor anomalies, allowing rapid detection of pipeline problems.



Weekly Report:

Aggregated over seven-day windows, these reports track long-term trends in both input data and model accuracy. Evidently computes regression metrics - Mean Error (ME), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE)
The Predicted vs Actual in Time plot visualizes how model predictions deviate from true sensor values across the week, helping identify gradual model drift.

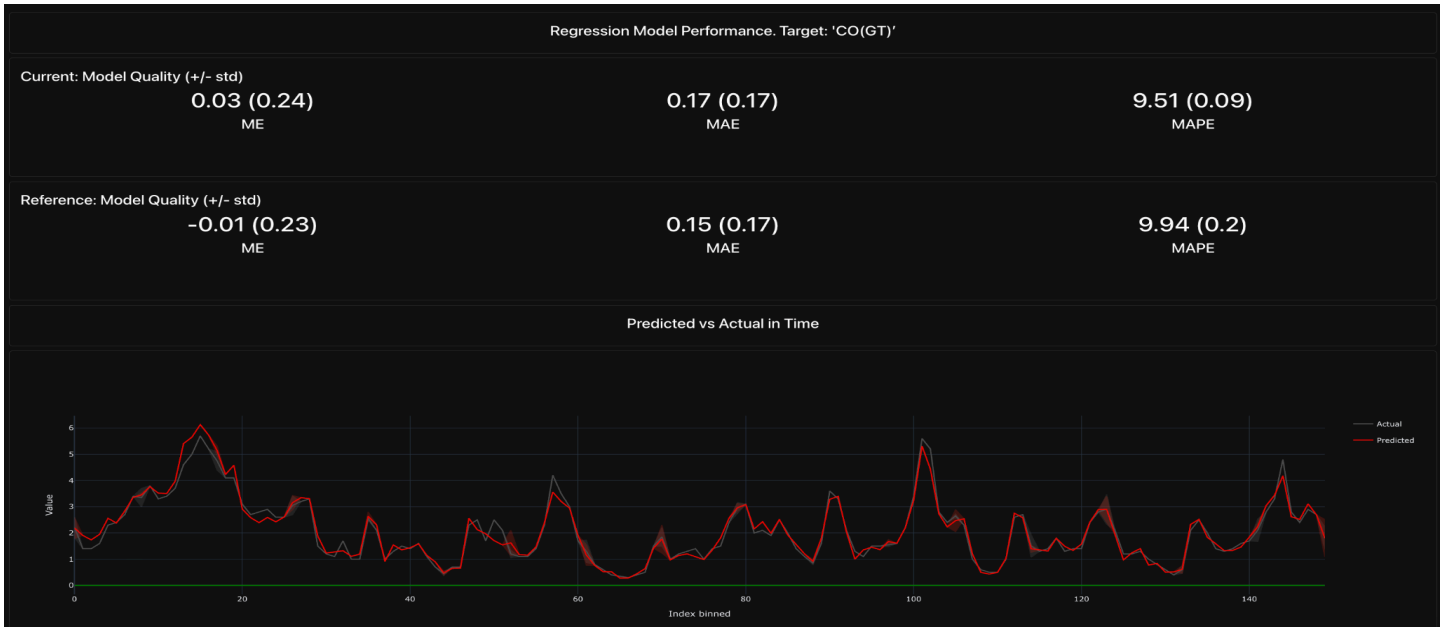
Model Performance Metrics (Evidently Weekly Report)

- **Model Quality (± std):**
  The overall model quality score increased slightly from -0.01 in the reference period to 0.03 in the current week, indicating a small improvement but with marginally higher variability.

- **Mean Error (ME)**:
  The average signed prediction error rose from 0.15 to 0.17, suggesting a minor positive bias where the model tends to slightly overpredict

- **Mean Absolute Error (MAE):**
  The mean absolute deviation between predicted and actual values improved from 9.94 to 9.51, reflecting stable accuracy and a small performance gain over the week.

- **Mean Absolute Percentage Error (MAPE):**
  The relative error remained consistent with previous results, showing no meaningful increase in percentage deviation, which implies minimal performance drift.



Bonus : Prometheus & Grafana

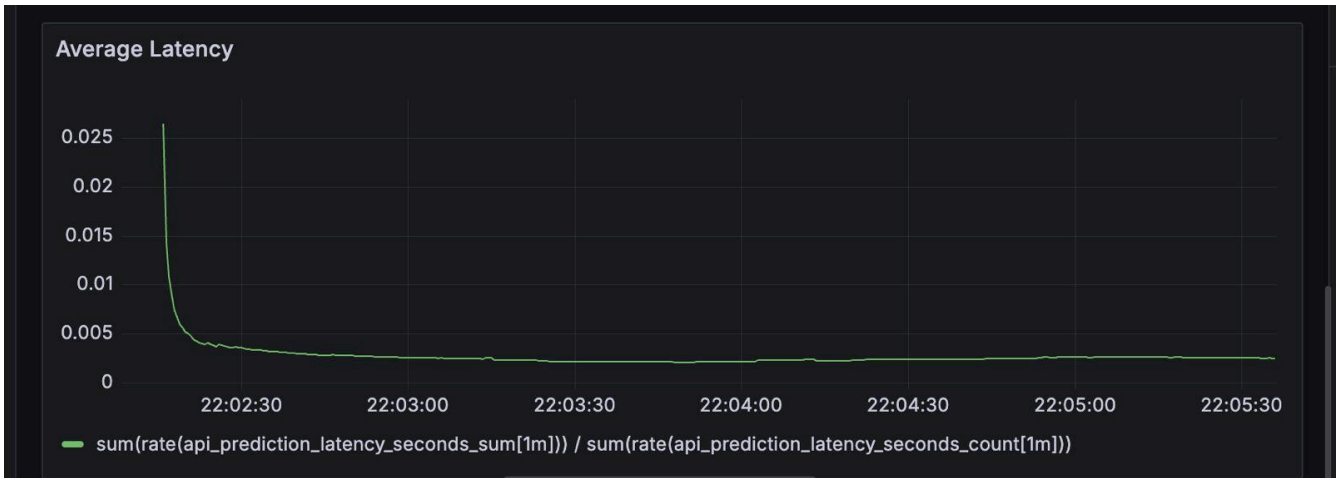
The Air Quality Prediction API was fully containerized, and both Prometheus and Grafana were deployed within the same Docker Compose environment to provide seamless monitoring and visualization of the system's performance. Prometheus continuously scrapes metrics from the FastAPI service's /metrics endpoint, where the API exposes key operational and model-serving indicators such as total predictions made, request latency, error counts, and missing features. These metrics are stored as time-series data within Prometheus, which acts as the centralized monitoring engine.

Grafana was configured to use Prometheus as its primary data source and visualize these metrics through dynamic, real-time dashboards. Using PromQL queries, several panels were built to track average latency, prediction throughput, missing feature trends, and error occurrences over time. The dashboards provided an immediate view into how the API performed under varying loads, showing a short warm-up phase followed by consistently low latency values around 2–4 milliseconds. System health indicators like uptime and memory usage were also monitored to ensure the API remained stable and responsive throughout testing.

By dockerizing both Prometheus and Grafana alongside the model API, the entire observability pipeline became fully reproducible and portable across environments. Every component starts automatically through Docker Compose, simplifying deployment and eliminating configuration drift. This setup mirrors production-grade MLOps practices, where real-time monitoring, alerting, and performance analysis are embedded into the lifecycle of deployed machine learning models. The resulting monitoring stack provided continuous visibility into model behavior, data quality, and infrastructure health, ensuring a robust and scalable deployment.

The **Predictions** graph visualizes the latest prediction values generated by the API in real time. The continuous oscillations indicate active model inference responding to changing input data, with prediction values varying dynamically across the monitored window. This behavior confirms that the API is functioning consistently, producing outputs at a steady frequency and reflecting expected fluctuations in air quality levels across successive requests.

The **Missing Features** graph tracks the number of absent input features in each prediction request. A consistently flat line at zero indicates that all incoming payloads contain complete feature sets, confirming proper data validation and schema consistency between the client and model. This metric is useful for identifying malformed or incomplete requests in real time.



The **Average Latency** graph represents the mean time taken by the API to process each prediction request over time. The initial spike corresponds to model warm-up and resource initialization, after which the latency stabilizes at around 2–4 milliseconds, reflecting consistently fast and efficient inference performance. This trend confirms the API's readiness for real-time deployment with minimal response delays.

The **Prediction Throughput** graph shows the rate of predictions made by the API, measured as the number of successful inferences per second. The steady rise followed by a flat plateau indicates the system ramping up to a stable request frequency, achieving around two predictions per second. This consistent throughput demonstrates that the API is handling requests efficiently and maintaining a steady processing rate under load.

Limitations and Recommendations

The current implementation runs within a single-node Docker Compose setup. While this is ideal for demonstration and reproducibility, it lacks the fault tolerance, auto-scaling, and load-balancing capabilities of a distributed setup for instance, multi-broker Kafka cluster and Kubernetes-based orchestration. As a result, throughput and resilience would degrade under production-scale data loads. The Evidently monitoring service currently uses a static reference dataset generated from validation data. Over time, this baseline may become outdated as environmental conditions evolve, for instance, seasonal or long-term pollution shifts, leading to false drift alerts or delayed detection. Periodic reference re-generation or adaptive baseline updates would improve robustness.

Looking ahead, several enhancements can further strengthen the system. Incorporating dimensionality reduction techniques such as PCA or autoencoders could reduce computational complexity while retaining signal quality. Expanding to multi-source forecasting, including external data like weather or traffic, could improve generalization in real-world deployments. Finally, integrating LLMs for automated business insights could allow stakeholders and leadership teams to interpret predictions, identify key drivers, and make data-driven decisions with natural language interfaces, bridging the gap between analytics and strategic planning.

Appendix

AI Usage:

AI tools were used across multiple stages of the Air Quality Prediction project to support data preparation, code generation, documentation, and visualization.

1. Dataset Creation and Feature Engineering
Tool Name/Version: ChatGPT 5 and Claude 4 Sonnet
Purpose: To assist in developing and refining the `create_dataset.py` and `generate_reference_dataset.py` scripts. These scripts handle time-series preprocessing, temporal splits, lag and rolling feature generation, and reference dataset creation for Evidently monitoring.
Input Query: Generate efficient Python code to clean and transform air quality data, apply interpolation, and engineer lag-based and cyclical time features.
AI Output Modification: The code was reviewed, tested, and adjusted for compatibility with project structure and dataset schema.
Verification: All generated artifacts (`eval_data_engineered.csv`, `reference.csv`, `features.json`) were validated through manual inspection and data integrity checks.

2. Documentation Refinement
Tool Name/Version: Grammarly
Purpose: To enhance clarity, coherence, and readability of the written report and project documentation. Grammarly was used to ensure grammatical precision and consistency throughout the report.
Verification: All revisions were reviewed manually to preserve technical accuracy and original context.

3. Visualization and Diagram Design
Tool Name/Version: Mermaid Flowchart Maker (AI-assisted)
Purpose: To automatically generate architecture flowcharts representing the data pipeline, model serving architecture, and monitoring workflow.
Verification: All AI-generated diagrams were reviewed and modified for accuracy and alignment with the implemented system components.

4. Development Efficiency
Tool Name/Version: AI-assisted Tab Completion (Visual Studio Code Copilot-style suggestions)
Purpose: To improve code writing efficiency during development by providing context-aware autocompletions and syntax corrections.
Verification: Each auto-completed code segment was manually tested, debugged, and validated to ensure correctness and alignment with intended functionality.

No AI assistance was used in core model training, evaluation metric computation, or validation workflows. All experiments and results were independently executed and verified.