NAME: ZAINAB FATIMA

INTERN: DATA SCIENCE

COHORT: 5

PROJECT: AQI-PREDICTOR FOR NEXT 3 DAYS

SUBMITTED ON: 18. AUG. 2025

Technical Report

Overview

The **AirLens** is a comprehensive **air quality monitoring and prediction platform** integrating a **FastAPI backend** for data processing, **machine learning-based forecasting**, and a **Streamlit frontend** for an engaging, user-friendly interface. The system is designed for **real-time AQI tracking**, **72-hour predictions**, and **interactive analytics**, making it suitable for both public awareness and research purposes.

Key Features

1. Frontend (Streamlit)

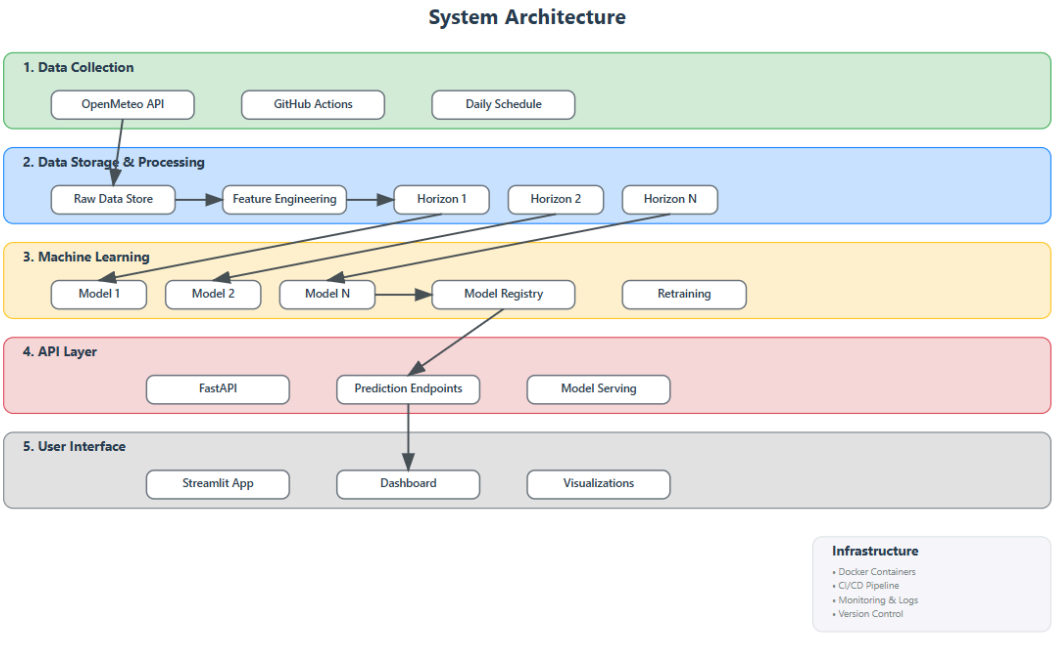
- **Live Dashboard**: Interactive real-time AQI monitoring with high-quality charts and metrics.
- **AI Predictions**: 72-hour AQI forecasts with hourly resolution
- **City Comparison**: Multi-city AQI comparison with live ranking.
- **Detailed Analytics**: Historical trends, seasonal patterns, and pollutant-specific analysis.
- **Alert System**: Real-time warning notifications for unhealthy air quality levels.

2. Backend (FastAPI)

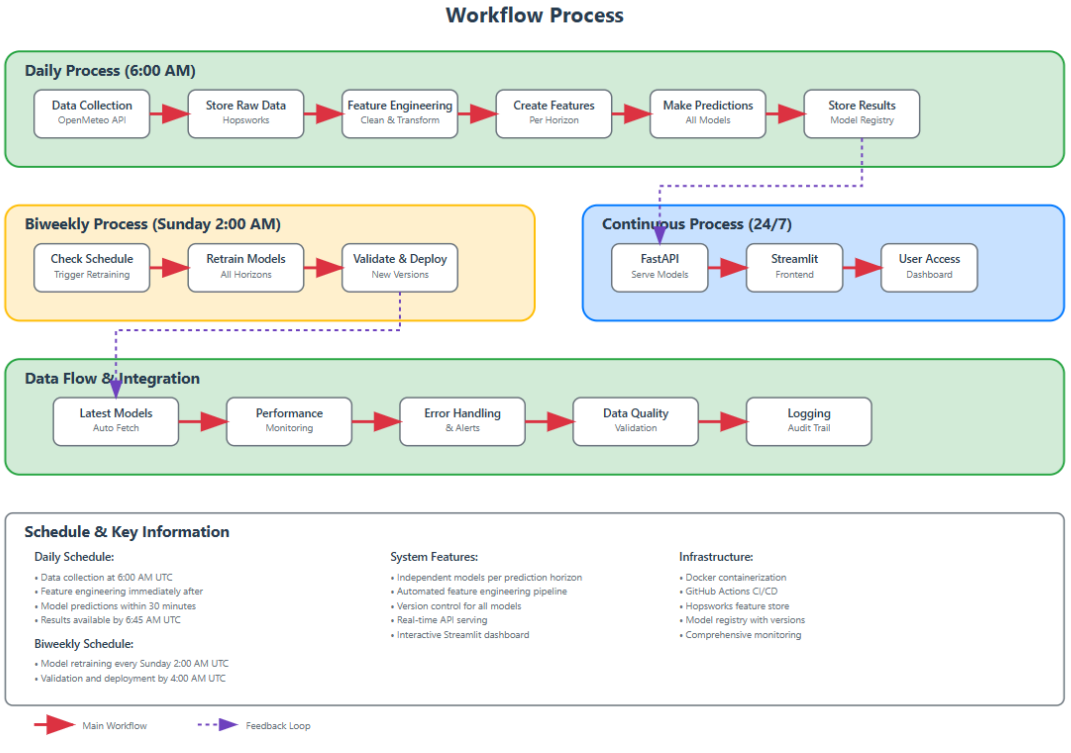
- **RESTful API**: Modular endpoints for forecasts, historical data, and system health.
- **ML Model Integration**: Pre-trained models for 24h, 48h, and 72h AQI predictions.
- **Data Pipeline**: Automated real-time AQI and weather data ingestion, transformation, and storage.
- **Health Monitoring**: API-level system health checks and status reports.

System Design

1. System Architecture



2. Workflow Diagram



Data Collection Pipeline

This pipeline automates **daily acquisition, processing, and storage** of Karachi's Air Quality Index (AQI) and weather data. The system combines **real-time monitoring** (secondary pipeline) with **historical data analysis** (primary pipeline) to provide comprehensive air quality insights for machine learning model training and public health monitoring.

System Architecture Overview

Pipeline Hierarchy

Primary Pipeline (Training & Historical Analysis)

- **Data Source**: OpenMeteo API
- **Coverage**: 92 days of historical data
- **Purpose**: Machine learning model training
- **Storage**: Hopsworks Feature Store
- **Update Frequency**: Daily

Secondary Pipeline (Real-time Monitoring)

- **Data Sources**: IQAir → AQICN → OpenWeather (priority order)
- **Coverage**: Real-time data for 3 cities
- **Purpose**: Current conditions monitoring
- **Storage**: CSV files with hourly Hopsworks uploads
- **Update Frequency**: Hourly

Data Source Integration

Primary Pipeline: OpenMeteo API

Why OpenMeteo? OpenMeteo provides the most comprehensive historical dataset, offering up to 92 days of hourly data with excellent API reliability.

Endpoint: https://archive-api.open-meteo.com/v1/archive
Coverage: 92 days historical data
Rate Limit: 10 calls per day
Data Quality: High accuracy, validated meteorological data from CAMS DATA PROVIDER.

Data Collected:

- **Air Quality**: PM10, PM2.5, CO, CO_2, NO_2, SO_2, O_3
- **Weather**: Temperature, humidity, precipitation, wind (10m & 100m)
- **Temporal**: Hourly resolution with precise timestamps

Secondary Pipeline: Multi-Source Approach

1st Priority: IQAir API

Endpoint: `https://api.airvisual.com/v2/nearest_city`
Rate Limit: 10,000 calls/month
Strengths: Real-time AQI calculations, global coverage
Data Focus: Current AQI values, PM2.5 concentration

2nd Priority: AQICN API

Endpoint: `https://api.waqi.info/feed/geo:{lat};{lon}/`
Rate Limit: 1,000 calls/day
Strengths: Comprehensive pollutant data, local stations
Data Focus: PM2.5, PM10, NO_2, SO_2, CO, O_3, weather

3rd Priority: OpenWeather API

Endpoint: `https://api.openweathermap.org/data/2.5/weather`
Rate Limit: 1,000 calls/day
Strengths: Excellent weather data, air pollution layer
Data Focus: Weather conditions hourly

Target Cities & Coordinates

Karachi:

- **Coordinates**: 24.8607°N, 67.0011°E
- **Population**: ~15 million
- **Air Quality Challenges**: Industrial emissions, vehicle pollution, coastal humidity

Lahore:

- **Coordinates**: 31.5204°N, 74.3587°E
- **Population**: ~11 million
- **Air Quality Challenges**: Winter smog, crop burning, industrial pollution

Islamabad:

- **Coordinates**: 33.6844°N, 73.0479°E
- **Population**: ~1 million
- **Air Quality Challenges**: Seasonal variations, construction dust, regional transport

Rate Limiting & Request Management

Our system implements sophisticated rate limiting to respect API constraints while maximizing data availability:

```
class APIClient:
    def __init__(self, timeout=30, rate_limit_delay=1.0, max_retries=3):
        self.rate_limit_delay = 1.0
        self.max_retries = 3
        self.request_tracking = {}
```

Rate Limiting Strategy:

- **1-second delay** between consecutive requests
- **Exponential backoff**: 1s → 2s → 4s on retries
- **Request tracking** for success rate monitoring

Data Standardization & AQI Calculations

EPA Breakpoint Formula Implementation

Reference: U.S. EPA AQI Breakpoints and Calculation Formula

Our system implements the official EPA formula for AQI calculation:

$$\mathbf{AQI = ((IHi - ILo) / (BPHi - BPLo)) \times (Cp - BPLo) + ILo}$$

Where:

- **AQI** = Air Quality Index
- **Cp** = Pollutant concentration
- **BPHi** = Breakpoint concentration ≥ Cp
- **BPLo** = Breakpoint concentration ≤ Cp
- **IHi** = AQI value corresponding to BPHi
- **ILo** = AQI value corresponding to BPLo

Pollutant Standardization

Primary Pollutants & Units

Pollutant	Standard Unit	Conversion Notes
PM2.5	µg/m³	Fine particulate matter
PM10	µg/m³	Coarse particulate matter
O_3	ppm	8-hour average
CO	ppm	8-hour average
SO_2	ppb	1-hour average

NO_2	ppb	1-hour average

Unit Conversion Process

From µg/m³ to ppm/ppb:

```
def convert_concentration(value, pollutant, source_unit,
target_unit):
    """Convert pollutant concentrations using molecular weights"""
    molecular_weights = {
        'NO2': 46.0055,  # g/mol
        'SO2': 64.066,
        'CO': 28.01,
        'O3': 47.998
    }

    if source_unit == 'µg/m³' and target_unit == 'ppb':
        # Convert using standard temperature and pressure
        return (value * 24.45) / molecular_weights[pollutant]
```

EPA AQI Breakpoint Table

PM2.5 Breakpoints (µg/m³)

AQI Range	Concentration	Category
0-50	0.0-12.0	Good
51-100	12.1-35.4	Moderate
101-150	35.5-55.4	Unhealthy for Sensitive
151-200	55.5-150.4	Unhealthy
201-300	150.5-250.4	Very Unhealthy
301-500	250.5-500.4	Hazardous

PM10 Breakpoints (µg/m³)

AQI Range	Concentration	Category
0-50	0-54	Good
51-100	55-154	Moderate
101-150	155-254	Unhealthy for Sensitive
151-200	255-354	Unhealthy

201-300	355-424	Very Unhealthy
301-500	425-604	Hazardous

Data Processing Pipeline

Primary Pipeline Workflow

Daily Trigger (GitHub CI) → OpenMeteo API (92-day data) → Unit Conversion & AQI Calculate → Data Validation & Deduplication → Feature Store Upload

Technical Implementation

Data Quality & Validation

Data Validation Pipeline:

- **Range Checking**: Pollutant concentrations within realistic bounds
- **Temporal Validation**: Timestamp consistency and chronological order
- **Missing Value Handling**: Strategic imputation
- **Outlier Detection**: Statistical analysis for anomalous readings
- **Cross-Validation**: Multiple API sources for data verification

Standardization Process

1. **Unit Normalization**: Convert all pollutants to EPA-standard units
2. **Coordinate Validation**: Ensure GPS coordinates match target cities
3. **Temporal Alignment**: Standardize timestamps to UTC
4. **AQI Calculation**: Apply EPA breakpoints consistently
5. **Data Merging**: Combine multiple sources intelligently

Deployment & Automation

Scheduling Strategy

Primary Pipeline (Daily):

- **Trigger**: GitHub Actions cron job (daily at 5 AM UTC)
- **Duration**: ~2 minutes for 92-day data processing
- **Backup**: Manual trigger capability for missed runs

Secondary Pipeline (Hourly):

- **Trigger**: GitHub Actions cron job hourly
- **Duration**: ~2 minutes per city cycle

- **Resilience**: Automatic recovery from API failures

Cloud Integration

Hopsworks Feature Store:

- **Automated uploads** with version control
- **Schema validation** and data quality checks
- **Online/offline availability** for ML model serving
- **Historical versioning** for data lineage tracking

Data Processing and Model Preparation

This data processing and feature engineering pipeline is designed for multi-horizon Air Quality Index (AQI) forecasting in Karachi, Pakistan. The pipeline transforms raw meteorological and air quality data from Open-Meteo into a comprehensive, model-ready dataset optimized for tree-based machine learning algorithms. Through intelligent temporal feature engineering, multicollinearity reduction, and horizon-specific data preparation, this system ensures robust forecasting capabilities across 24, 48, and 72-hour prediction windows.

Pipeline Architecture Overview

Data Flow Architecture

Open-Meteo API → Hopsworks Feature Store → Data Retrieval →
Preprocessing Pipeline → Feature Engineering →
Multicollinearity Reduction → Horizon-Specific Storage

The pipeline operates on a foundation of temporal consistency and statistical rigor, processing 92 days of merged-hourly data to create features that capture both immediate patterns and longer-term atmospheric dynamics crucial for accurate AQI prediction.

Data Ingestion and Initial Processing

Source Data Integration

The pipeline begins with data retrieved from Hopsworks feature groups, containing comprehensive atmospheric measurements from Open-Meteo. The initial data structure includes core pollutants (PM2.5, PM10, NO_2, SO_2, CO, O_3), meteorological variables (temperature, humidity, wind patterns), and calculated AQI values.

Data Quality Assurance

Duplicate Removal and Sorting

The pipeline performs chronological sorting and duplicate elimination as its first quality control measure. This step is critical because temporal inconsistencies or duplicate timestamps would compromise the integrity of lag features and rolling calculations. The system maintains data lineage by tracking the number of duplicates removed, providing transparency in the cleaning process.

Advanced Outlier Management

Rather than using simple statistical removal, the pipeline implements a sophisticated outlier capping strategy that preserves data integrity while preventing extreme values from distorting model training. For the target variable (AQI), the system uses 5th and 95th percentiles as bounds, acknowledging that extreme air quality events are legitimate and valuable for prediction. For feature variables, traditional interquartile range (IQR) bounds with a 3-sigma multiplier are applied.

When outliers are detected, they are replaced with NaN values and subsequently filled using temporal interpolation methods. This approach maintains the temporal structure of the data while reducing the impact of measurement errors or sensor malfunctions that could otherwise create false patterns in the training data.

Missing Value Treatment Strategy

Forward and Backward Fill

The initial imputation stage uses forward fill (last observation carried forward) with a limit of 2 consecutive missing values. This method is particularly effective for slowly changing atmospheric conditions where the most recent valid measurement provides a reasonable estimate. Subsequently, backward fill addresses any remaining gaps, ensuring that isolated missing values surrounded by valid data are properly interpolated.

Temporal Feature Engineering

Cyclical Pattern Capture

The pipeline creates comprehensive temporal features designed to capture both human activity patterns and natural atmospheric cycles. Basic temporal features include hour, day of week, month, and season, providing the model with explicit knowledge of cyclical patterns that strongly influence air quality in urban environments.

Sinusoidal Transformations

To handle the cyclical nature of temporal variables, the system applies sine and cosine transformations to key time components. For hourly patterns, the transformation `sin(2π * hour / 24)` and `cos(2π * hour / 24)` ensures that hour 23 and hour 0 are recognized as adjacent rather than numerically distant. Similar transformations are applied to daily and monthly cycles, enabling the model to understand that Monday follows Sunday and January follows December.

These trigonometric features are particularly crucial for AQI forecasting in Karachi, where distinct patterns emerge during different parts of the day (morning rush hour, afternoon heat, evening activities) and different seasons (monsoon vs. dry periods).

Advanced Lag and Rolling Feature Engineering

Strategic Lag Feature Creation

The pipeline generates lag features with a minimum lag of the forecast horizon, ensuring that all historical information used by the model predates the forecast horizon. This approach prevents data leakage while providing the model with sufficient historical context to understand atmospheric trends and patterns.

Multi-Horizon Lag Strategy

Lag features are created at strategic intervals: 72h, 84h, 96h, 120h, 144h, and 168h. These intervals capture daily patterns (24-hour multiples), weekly patterns (168 hours), and intermediate patterns that often appear in atmospheric data.

For the target variable (AQI), lag features provide the model with information about recent air quality trends, enabling it to distinguish between isolated pollution events and sustained pollution episodes. Pollutant-specific lag features capture the temporal dynamics of individual contaminants, while meteorological lag features provide context about the atmospheric conditions that drive pollutant transport and transformation.

Rolling Feature Computation

Lagged Rolling Statistics

Rolling features are computed on data that is already lagged by the minimum forecast horizon (24 hours), ensuring temporal validity while providing trend information. The system calculates rolling means, standard deviations, and maxima over 24, 48, and 72-hour windows, capturing both short-term volatility and longer-term trends in air quality and meteorological conditions.

The rolling standard deviation features are particularly valuable for capturing air quality volatility, which often precedes significant changes in pollution levels. High volatility in PM2.5 concentrations, for example, may indicate unstable atmospheric conditions that could lead to rapid air quality deterioration or improvement.

Pollutant-Specific Rolling Windows

Different pollutants exhibit different temporal characteristics, and the pipeline adapts rolling window sizes accordingly. PM2.5 and PM10 use 24-hour windows to capture daily accumulation patterns, while gaseous pollutants (NO_2, SO_2, CO, O_3) use both 24 and 48-hour windows to capture their different atmospheric lifetimes and transport characteristics.

Interaction and Statistical Feature Engineering

Meteorologically-Informed Interactions

The pipeline creates interaction features based on physical understanding of atmospheric processes affecting air quality in Karachi's coastal urban environment.

Temperature-Humidity Interaction

The temperature-humidity interaction feature (`temperature × humidity / 100`) approximates heat index effects that influence pollutant chemistry and vertical mixing in the atmosphere. In Karachi's humid climate, this interaction is particularly important for understanding secondary pollutant formation and dispersion patterns.

Wind Decomposition

Wind speed and direction are decomposed into u and v components using trigonometric transformations:

- `wind_u = wind_speed × cos(wind_direction)`
- `wind_v = wind_speed × sin(wind_direction)`

This decomposition allows the model to better understand wind patterns relative to Karachi's geography and pollution sources. The coastal location means that sea breezes, land breezes, and monsoon flows have distinct signatures that are better captured through vector components than scalar speed and direction values.

Particulate Matter Ratios

The PM2.5/PM10 ratio provides insight into the composition of particulate pollution, distinguishing between fine particles (primarily from combustion) and coarse particles (dust, sea salt, biological material). This ratio is particularly informative in Karachi, where dust storms, construction activities, and vehicle emissions create different particulate signatures.

Statistical Trend Features

The pipeline computes rate-of-change features for AQI over 24, 48, and 72-hour periods using lagged data to maintain temporal validity. These features capture acceleration and deceleration patterns in air quality trends, providing the model with information about whether current conditions represent improving, stable, or deteriorating air quality trajectories.

Multicollinearity Reduction and Feature Selection

Correlation-Based Feature Pruning

The pipeline implements a sophisticated multicollinearity reduction strategy that balances information retention with model stability. The correlation threshold of 0.85 is carefully

chosen to remove redundant features while preserving complementary information from related but distinct atmospheric variables.

Triangle Method Implementation

The correlation reduction uses an upper-triangular matrix approach to systematically identify and remove highly correlated feature pairs. When two features exceed the correlation threshold i.e 0.85, the system retains the feature with higher correlation to the target variable, ensuring that the most predictive information is preserved while eliminating redundancy.

This method is particularly important in atmospheric data where many variables are physically related. For example, different lag periods of the same variable may be highly correlated, and the system intelligently selects the most informative lag while removing redundant temporal information.

Variance-Based Feature Filtering

Low Variance Elimination

Features with variance below 1×10^{-6} are automatically removed, as they provide no discriminatory information for model training. This threshold is set low enough to preserve features with small but meaningful variations while eliminating truly constant features that could arise from sensor failures or data processing errors.

Statistical Validation

Throughout the feature selection process, the pipeline tracks which features are removed and why, providing transparency and enabling validation of the feature selection decisions. This documentation is crucial for model interpretation and debugging.

Horizon-Specific Target Engineering

Multi-Horizon Target Creation

The pipeline creates target variables for 24, 48, and 72-hour forecasting horizons using forward-shifted AQI values. These targets enable multi-step forecasting while maintaining the ability to train separate models optimized for different prediction time frames.

Target Alignment Strategy

Future targets are created using `shift(-h)` operations, where h represents the forecast horizon. This approach ensures that each training sample includes features from time t and targets from time t+h, creating properly aligned input-output pairs for supervised learning.

The system then removes samples where future targets are unavailable (from both at the start and end of the dataset), ensuring that all training samples have complete target information across all forecast horizons.

Smart Data Finalization

Maximum Window Analysis

The pipeline implements a row-dropping strategy based on the maximum lag/rolling window size detected in the feature set. The `smart_drop_initial_rows` function automatically identifies the largest temporal window used in any feature and removes the corresponding number of initial rows from the dataset.

Window Detection Algorithm

The system uses regular expressions to identify temporal windows in feature names (e.g., "72h", "168h") and automatically determines the maximum lag period. This ensures that all samples in the final dataset have complete historical context for every engineered feature, preventing the model from training on samples with incomplete information.

Final Quality Control

Iterative Missing Value Treatment

After feature engineering, the system performs a final missing value check and applies linear interpolation with both forward and backward filling to handle any remaining gaps. This final step ensures that the sophisticated feature engineering process hasn't introduced new missing values through mathematical operations on boundary cases.

Model-Optimized Design Decisions

Tree-Based Algorithm Optimization

The pipeline is specifically optimized for tree-based machine learning algorithms (ExtraTrees, XGBoost, LightGBM), which are well-suited for time series forecasting with engineered features.

Feature Scaling Omission

Unlike neural networks or distance-based algorithms, tree-based methods are invariant to monotonic transformations and do not require feature scaling. By intentionally omitting scaling steps, the pipeline reduces computational overhead while preserving the interpretability of feature importance scores that tree-based models provide.

High-Dimensional Feature Space Management

Tree-based algorithms can effectively handle high-dimensional feature spaces and automatically perform feature selection during training. The pipeline leverages this capability by creating a comprehensive feature set and allowing the algorithms to identify the most predictive combinations during model training.

Feature Selection

A deliberately high-dimensional feature set (including lagged, rolling, and forecasted variables) was generated; final selection from this set was performed in the modeling pipeline using a hybrid MI + RF approach.

Hopsworks Integration and Storage Strategy

Feature Group Management

The processed features are stored in horizon-specific Hopsworks feature groups, enabling modular model development and deployment. Each forecast horizon (24h, 48h, 72h) maintains its own optimized feature set, allowing for specialized model architectures and hyperparameter tuning.

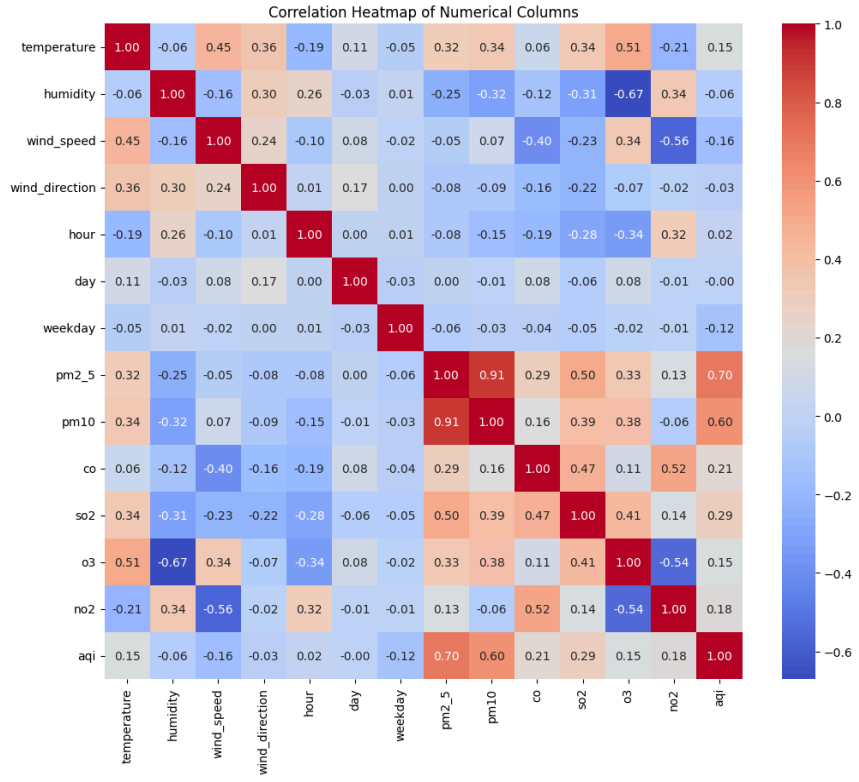
Version Control and Lineage

The Hopsworks integration provides automatic versioning and data lineage tracking, ensuring that model experiments can be reproduced and that feature engineering changes can be systematically evaluated across different forecast horizons.

EDA ANALYSIS:

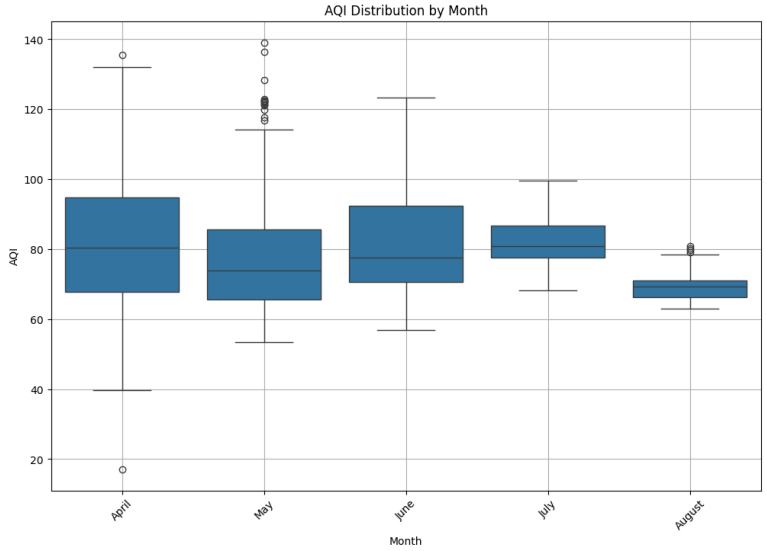
Correlation Heatmap of Numerical Columns

This heatmap shows the pairwise correlations between numerical features in the original raw data.



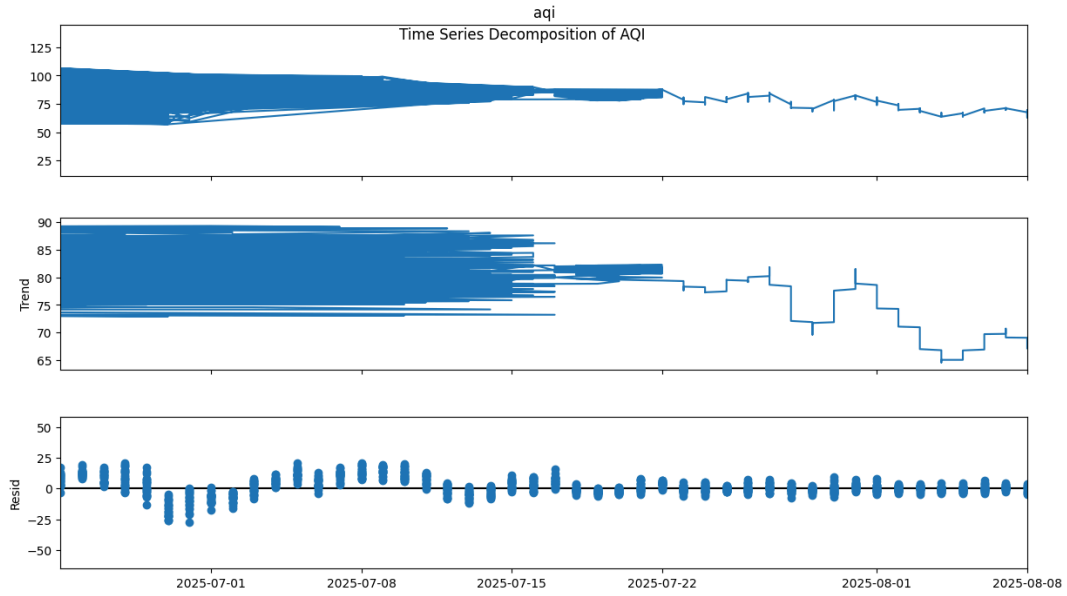
AQI Distribution by Month

This box plot visualizes the distribution of AQI across different months (April to August) in the original existing_df. It shows variations in the median AQI and the spread of values across these months, suggesting a seasonal pattern.



Time Series Decomposition of AQI

This plot decomposes the AQI time series into its trend, seasonal, and residual components. The trend component shows the overall long-term movement of AQI, while the seasonal component highlights repeating patterns within a fixed period (24 hours in this case).



MODEL PIPELINE AND EVALUATION

This forecasting system addresses the critical need for accurate AQI predictions across multiple time horizons (24, 48, and 72 hours) by implementing a comprehensive machine learning pipeline that combines advanced feature engineering, hybrid feature selection, and ensemble modeling techniques.

The pipeline employs a direct forecasting approach with sliding window cross-validation, specifically designed to handle the temporal dependencies inherent in air quality data while avoiding common pitfalls such as data leakage and poor generalization.

Feature Selection: Hybrid MI-RF Methodology

The feature selection process represents a critical breakthrough in the forecasting pipeline, implemented through a novel hybrid approach that combines mutual information (MI) and Random Forest (RF) importance scores. This methodology emerged from extensive experimentation that revealed the limitations of single-method approaches.

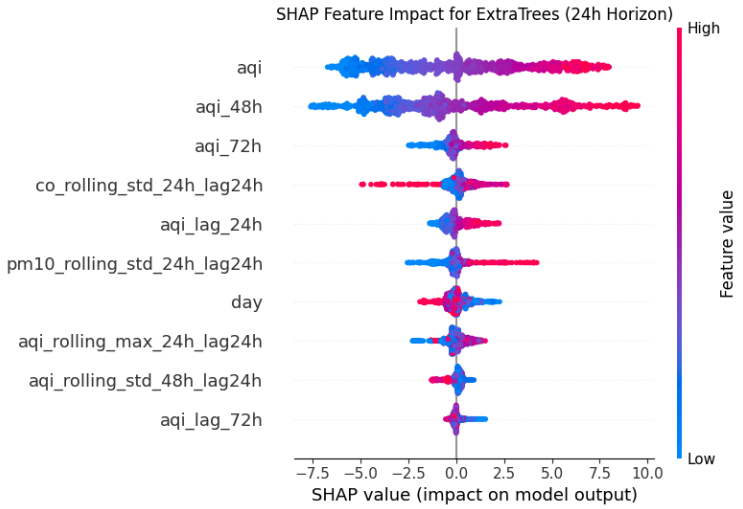
Mutual information quantifies the statistical dependence between feature variables and the target AQI by measuring how much information about the target variable is gained when the feature variable is known. MI captures both linear and non-linear relationships without making assumptions about data distribution, making it particularly valuable for identifying complex atmospheric interactions that traditional correlation measures might miss.

Random Forest importance operates differently, measuring each feature's contribution to decreasing weighted impurity across all decision trees in the forest ensemble. When a feature is used to split a node, it reduces the impurity (measured by variance for regression problems) of the resulting child nodes. The importance score represents the average reduction in impurity attributed to that feature across all trees, weighted by the number of samples passing through each node.

The hybrid approach leverages the complementary strengths of both methods through cross-validated feature selection. For each fold in a three-fold TimeSeriesSplit validation, both MI and RF importance scores are calculated independently on the training data. The system then aggregates these scores across folds, computing normalized scores for each method to ensure equal weighting.

This hybrid methodology consistently outperformed single-method approaches because MI alone sometimes selected redundant features with similar information content, while RF alone occasionally missed subtle non-linear relationships that MI could detect. The combination ensures selected features contribute unique and meaningful predictive power.

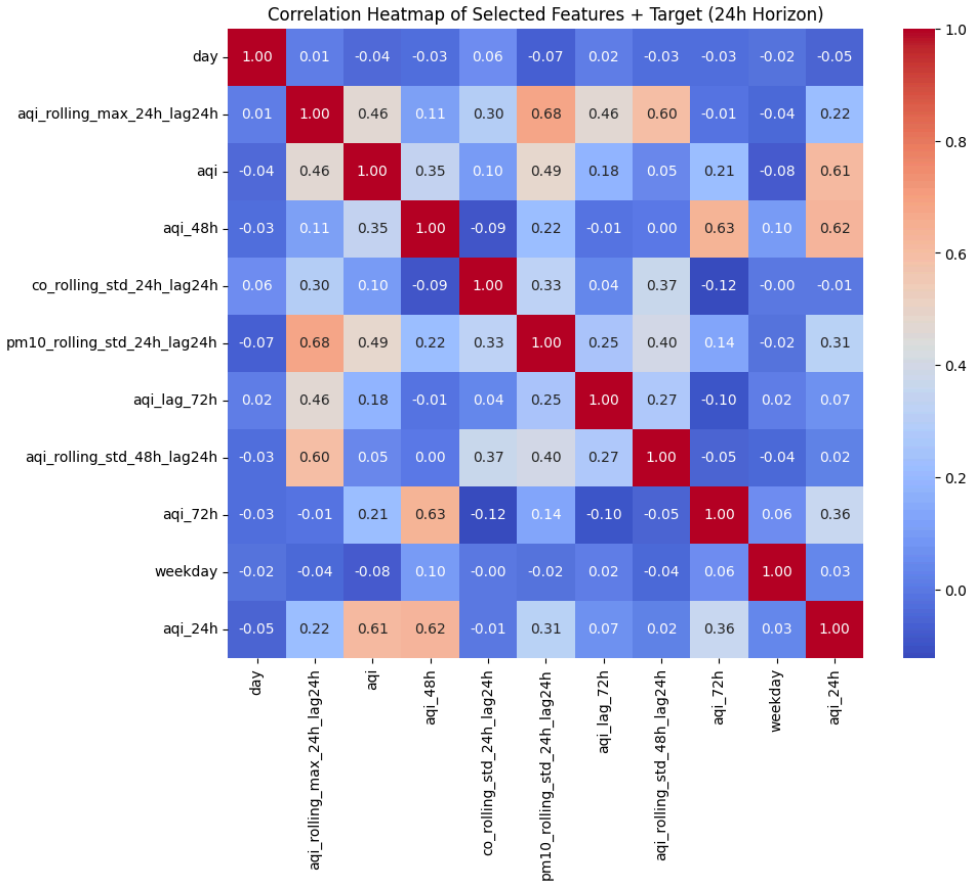
The maximum number of features was determined through SHAP analysis, which revealed diminishing returns beyond this threshold. SHAP values demonstrated that while the top 5-8 features contributed substantially to model predictions, additional features beyond 15 provided marginal improvements while increasing overfitting risk, particularly with the limited training data available for each sliding window.



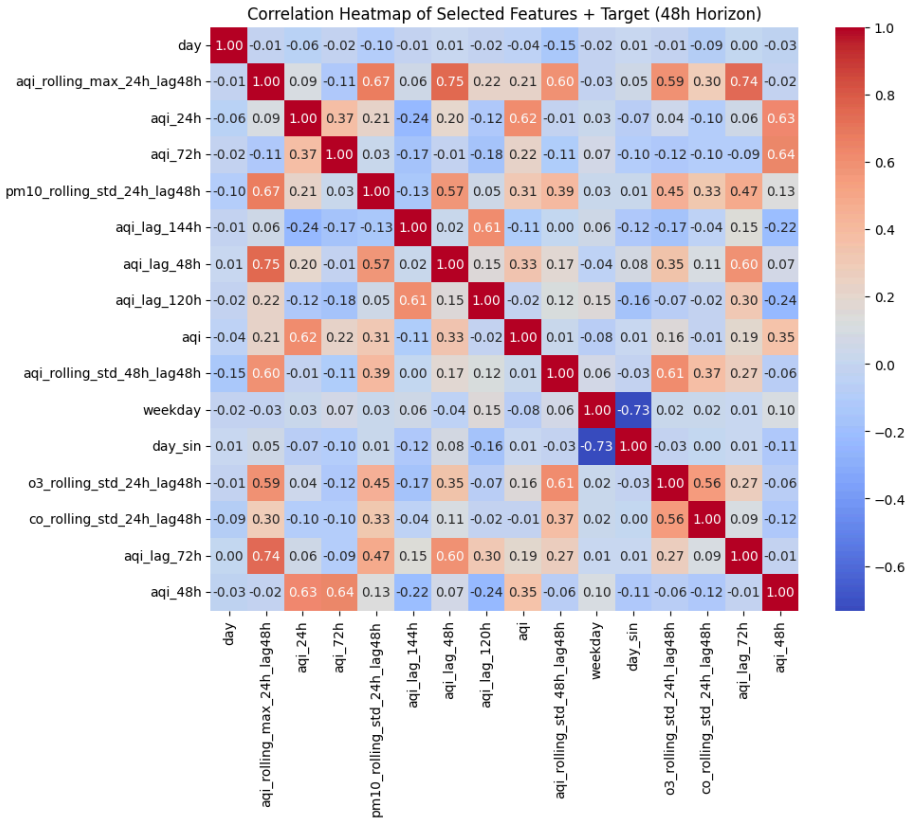
Correlation Heatmap of Numerical Columns

This heatmap shows the pairwise correlations between numerical features with their respective target columns:

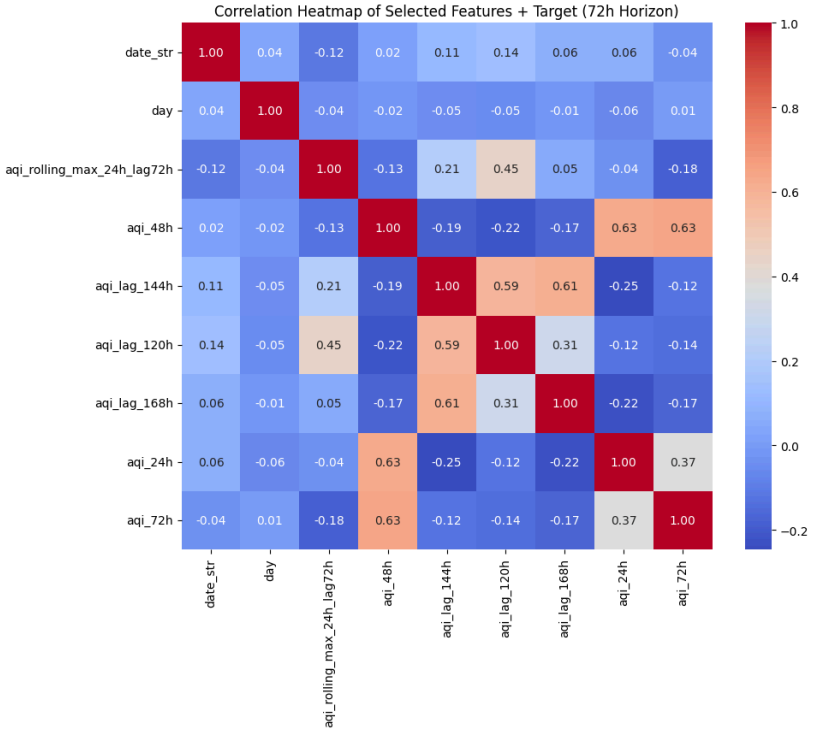
Target: aqi_24h



Target: aqi_48h



Target: aqi_72h



Iterative Approach Development and Experimentation

The development process involved systematic evaluation of multiple time series forecasting strategies, following the comprehensive framework outlined in Machine Learning Mastery's multi-step time series forecasting guide. Each approach revealed different challenges and insights that ultimately led to the final solution.

Approach 1: Direct Forecasting with All Features

Transitioning to direct forecasting (separate models for each horizon), the initial implementation used all available features - hundreds of variables including raw meteorological data, engineered features, lags, and interactions. While this approach demonstrated good performance on linear regression baselines, it suffered from severe overfitting when applied to more complex models. The abundance of features led to unstable predictions and poor generalization across different validation windows. After discussion with my mentor, it became clear that feature selection was essential for managing the high-dimensional feature space.

Approach 2: Single-Method Feature Selection

The first feature selection attempt used mutual information alone, which improved results but still wasn't delivering the expected performance levels. MI tended to select features with strong statistical relationships but occasionally included redundant variables providing similar information. Subsequently, Random Forest importance was tested independently, which showed good stability but sometimes missed subtle non-linear relationships that MI could capture.

Comparing MI-only and RF-only approaches revealed they were producing nearly identical results, suggesting that neither method alone was capturing the full spectrum of feature relevance. This insight led to developing the hybrid MI-RF approach, which combined both methods' scores. The hybrid approach improved R^2 performance and showed better cross-validated results with expanding window validation, though performance remained suboptimal without incorporating forecasted AQI values as features.

Approach 3: Manual Feature Selection

Stepping back from algorithmic selection, a manual approach was implemented using domain knowledge to select raw meteorological variables combined with specifically engineered volatility features and AQI rate change indicators. This approach provided interpretable and stable results with R^2 values ranging from 0.39 to 0.45 and RMSE around 14+. While offering good baseline performance and clear interpretability, it lacked the sophistication to capture complex atmospheric interactions and seasonal patterns that algorithmic methods could identify.

Approach 4: Delta Prediction with Cascading

An alternative strategy attempted to predict AQI changes (deltas) rather than absolute values, using a cascading approach with sliding window validation. The theory was that

changes might be more predictable and stable than absolute values. However, this approach suffered from error propagation issues - small errors in delta predictions compounded across the forecast horizon when converted back to absolute AQI values. Additionally, the increased dimensionality when incorporating multiple delta predictions created new modeling challenges without delivering the expected performance improvements.

Final Approach: Enhanced Direct Forecasting with Sliding Window

Following mentor consultation, the focus shifted to sliding window validation for direct AQI prediction. The breakthrough came from incorporating forecasted AQI values as features - for example, when predicting AQI at T+24 hours, using AQI at T, AQI at T+48, and AQI at T+72 as additional input features. Without these forecasted AQI features, the models produced negative R^2 values, indicating worse-than-baseline performance.

This enhancement transformed the results dramatically, improving cross-validated test R^2 from 0.40 to 0.55 and training R^2 to 0.80-0.85. The sliding window approach was particularly important for Karachi's highly fluctuating air quality patterns during the study period, as it provided more realistic validation by testing on truly unseen future data.

Future Enhancement: Deep Learning Transition

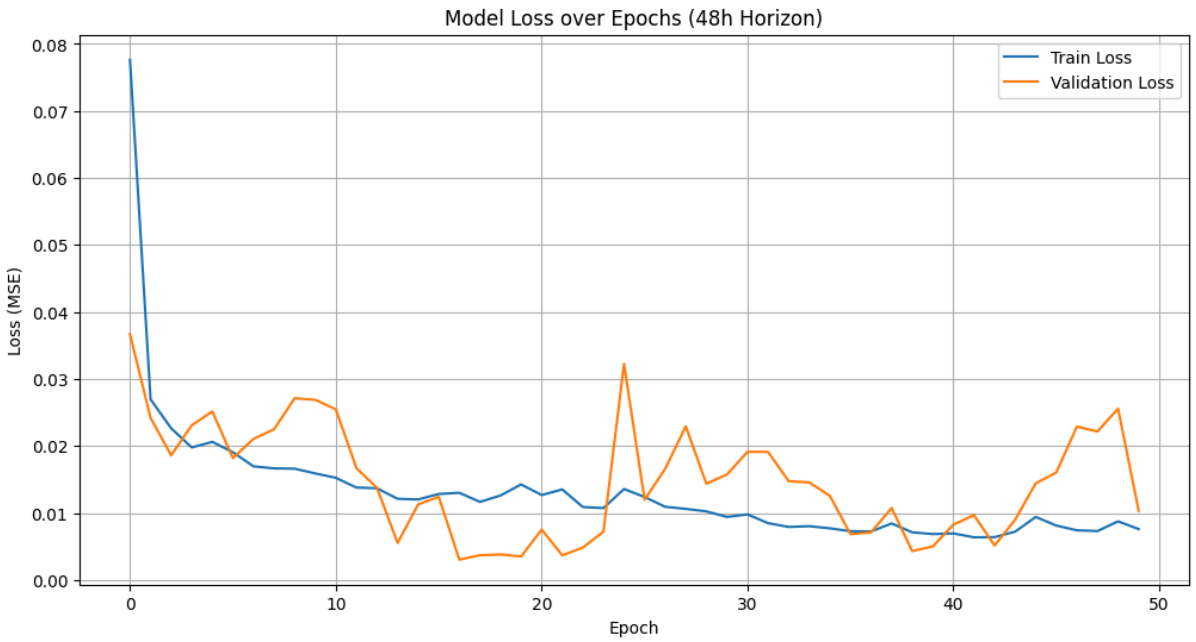
The planned solution is to transition to a single deep learning model architecture that can handle all prediction horizons efficiently.

What Has Been Tested So Far

Initial experiments were conducted with **LSTM-based deep learning models** for predicting AQI at 24h, 48h, and 72h horizons.

- Models were trained using **Mean Squared Error (MSE)** as the loss function.
- Evaluation was done with **time-series split validation** to respect temporal ordering.
- Basic architectures were implemented, and **RMSE** and **MAE** metrics were recorded for early benchmarking.

These experiments were exploratory and aimed at validating feasibility. Due to time constraints, the models have not yet been optimized or deployed.



Model Architecture and Hyperparameter Optimization

The modeling pipeline employs seven tree-based algorithms, each selected for specific strengths in handling air quality data characteristics. The models include CatBoost (robust categorical handling and overfitting protection), XGBoost (optimized gradient boosting), LightGBM (fast training and memory efficiency), RandomForest and ExtraTrees (ensemble diversity), GradientBoosting (classic boosting control), and DecisionTree (baseline comparison).

All hyperparameters were optimized using Optuna's Tree-structured Parzen Estimator algorithm, with R^2 as the primary optimization objective while monitoring RMSE, MAE, and MAPE. The optimization process incorporated early stopping for boosting algorithms and used cross-validated objective functions to ensure hyperparameters generalized across time periods with almost 20 folds for each model.

The sliding window cross-validation strategy used training windows of 8 weeks and test windows of 4 weeks, with 2-day step sizes. These parameters were determined through experimentation with different window sizes, showing consistent performance improvements with the selected configuration. This approach ensures temporal ordering is respected while providing adequate data for both training and validation.

Model Training and Deployment Strategy

Models are trained separately for each forecast horizon using the selected features and optimized hyperparameters. To ensure unbiased results, collinearity checks are performed during feature selection, removing features with correlation >0.85 with the target variable. The training process incorporates baseline model comparisons to validate that complex models provide meaningful improvements over simple approaches.

All trained models are saved in a model registry with comprehensive metadata including feature lists, hyperparameters, and performance metrics. The system implements daily retraining through GitHub Actions, automatically incorporating new data and updating model parameters. Since tree-based models can benefit from retraining with additional data, the entire pipeline is re-executed daily using the stored metadata and cross-validation framework to maintain optimal performance as new atmospheric data becomes available.

Final Performance Results

The complete evaluation reveals distinct performance characteristics across forecast horizons, with the expected accuracy degradation for longer forecasting periods.

Horizon	Best Model	Test R^2	Test RMSE	Test MAE	Test MAPE
24h	ExtraTrees	0.59	4.88	3.85	4.71%
48h	ExtraTrees	0.61	4.55	3.67	4.23%
72h	ExtraTrees	0.21	6.52	5.39	6.75%

Key Insights

The iterative development process revealed several critical insights for air quality forecasting. The hybrid MI-RF feature selection approach consistently outperformed single-method strategies. The incorporation of forecasted AQI values as features proved essential for achieving positive R^2 values, transforming model performance from negative to competitive levels.

ExtraTrees emerged as the most consistent performer across all horizons, likely due to its additional randomization providing better generalization for the highly variable Karachi air quality patterns. The sliding window validation approach was crucial for realistic performance estimation, particularly given the significant fluctuations in local air quality during the study period.

The comprehensive approach validation demonstrates that proper time series methodology is essential for atmospheric forecasting applications, with sliding window cross-validation providing the most reliable performance estimates for operational deployment.

CI/CD Pipeline Overview

My GitHub Actions pipeline consists of 5 automated workflows that orchestrate a complete machine learning pipeline for AQI (Air Quality Index) prediction, integrated with Hopsworks feature store:

1. Data Collection (`data-collect-aqi.yml`)

- **Schedule**: Daily at 5 AM UTC
- **Purpose**: Collects fresh AQI data from external sources
- **Integration**: Pushes raw data directly to Hopsworks feature store for centralized storage

2. Feature Preprocessing (`feature-preprocessing.yml`)

- **Schedule**: Daily at 6 AM UTC
- **Purpose**: Runs feature engineering pipeline to transform raw data into ML-ready features
- **Integration**: Processes data stored in Hopsworks and prepares it for model training

3. Model Training (`aqi-model-training-pipeline.yml`)

- **Schedule**: Daily at 7 AM UTC + manual trigger
- **Purpose**: Trains AQI prediction models using processed features from Hopsworks
- **Integration**: Stores trained models back to Hopsworks model registry

4. Update Models and Data (`update-models-and-data.yml`)

- **Schedule**: Hourly data updates + daily model updates (8 AM UTC)
- **Purpose**: Downloads trained models from Hopsworks model registry and updates local AQI data
- **Key Actions**: Fetches models, updates data files, commits changes to repository

5. Feature Data Extraction (`daily-feature-data-extraction.yml`)

- **Schedule**: Daily at 8 AM UTC
- **Purpose**: Extracts processed features from Hopsworks for different prediction horizons (24h, 48h, 72h)
- **Output**: Creates CSV files for each time horizon and commits to repository

This pipeline ensures continuous data ingestion, feature engineering, model training, and deployment with Hopsworks serving as the central feature store and model registry.

FastAPI Service

This documents the development, optimization, and deployment challenges of an Air Quality Index (AQI) Forecast API built with FastAPI. The project focuses on creating a secure, high-performance web service that predicts air quality for Karachi, Pakistan using machine learning models. This report covers the security implementations, performance optimizations, deployment challenges, and lessons learned during development.

System Architecture

Core Framework

- **Framework**: FastAPI 2.0.0
- **Language**: Python 3.x
- **Deployment**: Docker containerized
- **API Design**: RESTful architecture with SwaggerUI documentation

Directory Structure

```
backend/
├── data/           # Historical and real-time AQI data
├── config/         # Model configuration and feature definitions
├── models/         # Trained ML models for different horizons
└── main.py         # FastAPI application
```

Machine Learning Pipeline

Model Architecture

The system supports multiple machine learning frameworks:

- **CatBoost**: Gradient boosting (`.cbm` format)
- **XGBoost**: Extreme gradient boosting (`.json` format)
- **LightGBM**: Light gradient boosting (`.txt` format)
- **Scikit-learn**: Various algorithms (`.pkl` format)

Multi-Horizon Forecasting

- **24-hour horizon**: Short-term predictions
- **48-hour horizon**: Medium-term forecasts
- **72-hour horizon**: Long-term projections

Each horizon uses:

- Dedicated trained models selected based on R^2 performance
- Horizon-specific feature sets stored in JSON configuration
- Dynamic feature engineering from time-series data

Model Selection Process

```
# Automatic model format detection and loading

if model_format == 'cbm':

    model_obj = CatBoostRegressor()

    model_obj.load_model(model_path)

elif model_format == 'json':

    model_obj = XGBRegressor()

    model_obj.load_model(model_path)
```

Data Management

Data Sources

1. **Historical Data** (`aqi_data.csv`): Long-term AQI trends/secondary data-collection pipeline
2. **Horizon-Specific Data** (`horizon_{24,48,72}h_data.csv`): Feature-engineered datasets
3. **Real-time Updates**: GitHub Actions pipeline integration with Hopsworks

Feature Engineering

- Dynamic feature selection per horizon
- Automatic fallback to default values for missing features
- Last 72 records used for prediction context
- Features include: AQI, PM2.5, PM10, O_3, NO_2, SO_2, CO, temperature, humidity, wind metrics

Data Pipeline

```
def get_features_for_prediction(horizon, location="Karachi"):

    horizon_df = load_horizon_data(horizon)

    latest_row = horizon_df.iloc[-1]

    # Feature vector preparation with fallback handling
```

API Endpoints

1. Root Endpoint (`/`)

- **Rate Limit**: 20 requests/minute
- **Purpose**: Service health check and metadata
- **Response**: Service status, loaded models count

2. Forecast Endpoint (`/forecast`)

- **Rate Limit**: 10 requests/minute
- **Parameters**: Optional horizon (24, 48, 72 hours)
- **Response**: Predicted AQI with risk levels
- **Features**: Single or multi-horizon predictions

3. Hourly Forecast (`/forecast/hourly`)

- **Rate Limit**: 10 requests/minute
- **Interpolation**: Linear interpolation between major forecast points
- **Granularity**: Hour-by-hour predictions up to 72 hours
- **Default**: "Next 3 days" comprehensive forecast

4. Historical Data (`/historical/{location}`)

- **Rate Limit**: 10 requests/minute
- **Time Range**: 1-21 days (configurable)
- **Data Completeness**: Forward/backward fill for missing values
- **Columns**: Full environmental parameter set

5. Locations (`/locations`)

- **Rate Limit**: 15 requests/minute
- **Purpose**: Available monitoring locations
- **Data**: Latest AQI readings per city

6. Dashboard Overview (`/dashboard/overview`)

- **Rate Limit**: 10 requests/minute
- **Analytics**: Current AQI, weekly averages, trend analysis
- **Trend Detection**: Improving/stable/worsening classification

Security Implementation

Making the API as Secure as Possible

The FastAPI service was designed with security as a top priority. Multiple layers of protection were implemented to ensure safe and reliable operation.

Input Validation with Pydantic

Every endpoint uses strict input validation to prevent malicious or invalid data from entering the system:

```
class Forecast(BaseModel):

    horizon_hours: int

    predicted_aqi: float

    risk_level: str

# Query parameter validation with strict constraints

days: int = Query(default=7, ge=1, le=21, description="Number of
days (max 21)")

horizon: Optional[int] = Query(None, ge=24, le=72,
description="Specific horizon")
```

This approach ensures that:

- Only valid data types are accepted
- Numerical values stay within safe ranges
- Required fields are always present
- Invalid requests are rejected before processing

CORS Middleware Configuration

Cross-Origin Resource Sharing (CORS) was carefully configured to control which domains can access the API:

```
origins = [

    "http://localhost",

    "http://localhost:8080",

    "http://localhost:8501",

]

app.add_middleware(

    CORSMiddleware,

    allow_origins=origins,
```

```
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

This setup allows controlled access for development while preventing unauthorized cross-origin requests from unknown domains.

Rate Limiting Protection

Each endpoint has customized rate limits based on computational cost:

```
@limiter.limit("10/minute")  # Heavy ML prediction endpoints
async def get_forecast(request: Request, ...):


@limiter.limit("20/minute")  # Lightweight status endpoint
async def root(request: Request):
```

Rate limiting prevents:

- API abuse and spam requests
- Resource exhaustion attacks
- Uncontrolled usage that could crash the service

Admin Key Experimentation

During development, an admin key authentication system was tested for additional security layers on the model's health endpoint. However, this was removed from the final version to keep the API simple for the current use case.

Performance Optimization Journey

The Initial Problem: Heavy Memory Usage

Originally, the main.py file was directly loading machine learning models from Hopsworks feature store during startup. This approach had serious problems:

- **Memory Usage**: Over 1GB of RAM required
- **Loading Time**: Several minutes to start the service
- **Network Dependency**: Required constant connection to Hopsworks
- **Deployment Issues**: Too heavy for platforms like Vercel

The Solution: Local Model Storage

The architecture was redesigned to improve performance:

Before Optimization:

```
# Old approach - loading from Hopsworks directly

models = load_models_from_hopsworks()  # Very heavy!
```

After Optimization:

```
# New approach - local file system

def load_models_and_data():

    model_path = os.path.join(base_dir, "models", model_file)

    model_obj = joblib.load(model_path)  # Much faster!
```

Performance Improvements Achieved

Memory Usage Reduction

- **Before**: >1GB RAM required
- **After**: <200MB RAM required
- **Improvement**: 80%+ memory reduction

Startup Time Improvement

- **Before**: 5-10 minutes startup time
- **After**: 1-2 minutes startup time
- **Improvement**: 80%+ faster startup

Directory Structure for Performance

The optimized architecture uses three key directories:

```
backend/

├── data/            # Historical AQI data (CSV files)

├── models/          # Pre-trained ML models (joblib, cbm, json files)

└── config/          # Feature configurations (JSON files)
```

Model Loading Strategy

Different model formats are supported for flexibility:

```
if model_format == 'cbm':      # CatBoost models

    model_obj = CatBoostRegressor()
```

```
    model_obj.load_model(model_path)
elif model_format == 'json':   # XGBoost models
    model_obj = XGBRegressor()
    model_obj.load_model(model_path)
elif model_format == 'txt':    # LightGBM models
    model_obj = lgb.Booster(model_file=model_path)
else:                          # Scikit-learn models
    model_obj = joblib.load(model_path)
```

This approach allows the system to work with any machine learning framework while keeping memory usage minimal.

Risk Assessment & AQI Classification

AQI Risk Levels

- **Good**: 0-50
- **Moderate**: 51-100
- **Unhealthy for Sensitive Groups**: 101-150
- **Unhealthy**: 151-200
- **Very Unhealthy**: 201-300
- **Hazardous**: 300+

Trend Analysis

- **Calculation**: Recent vs. historical weekly averages
- **Thresholds**: 10% change for trend classification
- **Categories**: Improving (<90%), Stable (90-110%), Worsening (>110%)

Monitoring & Observability

Logging Strategy

```
logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(__name__)
```

Key Metrics Tracked

- Model loading success/failure
- Prediction generation time

- Data availability status
- Feature vector completeness
- API endpoint hit rates

Deployment Challenges and Lessons Learned

Vercel Deployment Attempt

The initial deployment target was Vercel, but several constraints were encountered:

Memory Limitations

- **Vercel Limit**: 512MB maximum memory for serverless functions
- **Our Requirement**: Originally >1GB due to heavy model loading
- **Solution**: Reduced to <200MB through local storage optimization

Cold Start Issues

- **Problem**: Models needed to reload on every function call
- **Impact**: 30+ second response times
- **Learning**: Serverless platforms aren't ideal for ML model hosting

Current Docker Deployment

The solution was containerization using Docker:

```
# Backend container setup

FROM python:3.9-slim

COPY backend/ /app/

RUN pip install -r requirements.txt

EXPOSE 8000

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

Testing and Validation

Swagger UI Testing

Local development used FastAPI's built-in Swagger UI:

- **Access**: `http://localhost:8000/docs`
- **Features**: Interactive API testing
- **Validation**: Real-time request/response testing
- **Documentation**: Auto-generated API documentation

Docker Container Testing

Current testing setup uses a dedicated Docker container:

```
# Running the backend container

docker run -p 8000:8000 backend-container
```

This provides a production-like environment for testing while maintaining consistency across different deployment environments.

Streamlit Frontend

The application serves as an intelligent user interface that connects to a FastAPI backend, providing real-time AQI monitoring, AI-powered predictions, and health-focused features. This report covers the frontend architecture, advanced alert system implementation, responsive design principles, and user experience optimization strategies.

Application Overview

Project Identity

- **Application Name**: AirLens, Smart Air Quality Companion
- **Technology Stack**: Streamlit, Python, Plotly, Docker
- **Target Users**: General public, health-conscious individuals, environmental researchers
- **Geographic Focus**: Pakistan (with multi-city support)
- **Deployment**: Docker containerized

Frontend Architecture

Multi-Page Application Structure

```
frontend/

├── app.py                      # Main application entry point

├── pages/                      # Modular page components

│   ├── dashboard.py           # Real-time monitoring dashboard

│   ├── prediction.py          # AI-powered forecasting interface

│   ├── city_comparison.py     # Multi-city comparison tools

│   └── analytics.py           # Historical analysis and trends

├── utils/                      # Utility modules

│   ├── alerts.py             # Comprehensive alert system
```

```
│   ├── api_client.py        # Backend communication layer
│   ├── helpers.py           # Helper functions
│   └── styles.py            # Custom CSS styling
└── requirements.txt          # Python dependencies
```

Session State Management

The application uses Streamlit's session state for persistent data management:

```
# Core session variables
if 'alerts' not in st.session_state:
    st.session_state.alerts = []
if 'current_aqi' not in st.session_state:
    st.session_state.current_aqi = None
if 'is_hazardous_conditions' not in st.session_state:
    st.session_state.is_hazardous_conditions = False
```

This approach ensures:

- **Data Persistence**: User data survives page navigation
- **Real-time Updates**: Dynamic content updates without page refresh
- **Cross-Component Communication**: Shared state across different pages

Advanced Alert System Implementation

Alert System Architecture

The alert system is built as a comprehensive class-based solution that provides automatic notifications:

Hazardous Condition Monitoring

```
def check_hazardous_aqi(self, aqi_value: float, location: str = "Karachi") -> bool:
    if aqi_value >= 301:  # Hazardous
        self.add_hazardous_alert('error', '🚨 CRITICAL: Hazardous Air Quality', ...)
    elif aqi_value >= 201:  # Very Unhealthy
```

```
        self.add_hazardous_alert('error', '⚠️ DANGEROUS: Very Unhealthy
Air', ...)
```

Alert Features

Automatic Health Monitoring

- **Real-time AQI Checking**: Monitors current air quality every 5 minutes
- **Emergency Notifications**: Special styling and longer duration for critical conditions
- **Location-Specific Alerts**: Contextual warnings based on geographic data

Visual Design Elements

- **Pulsing Animations**: Critical alerts feature pulsing effects for attention
- **Icon Integration**: Contextual emojis for quick visual recognition
- **Responsive Layout**: Adapts to different screen sizes seamlessly

Alert System Integration

API Integration

The alert system seamlessly integrates with the FastAPI backend:

```
def fetch_current_aqi_from_api(self, api_base_url: str) -> Optional[float]:

    try:

        response = requests.get(f"{api_base_url}/dashboard/overview",
timeout=10)

        if response.status_code == 200:

            data = response.json()

            return data.get('current_aqi')

    except requests.exceptions.RequestException as e:

        logger.error(f"Failed to fetch AQI from API: {e}")

        return None
```

Error Handling and Fallbacks

- **Connection Failures**: Graceful handling of API unavailability
- **Data Validation**: Ensures AQI values are within expected ranges
- **Logging Integration**: Comprehensive error logging for debugging
- **User Feedback**: Clear messages when data is unavailable

Responsive Design Implementation

CSS Framework

The application uses custom CSS with mobile-first responsive design principles:

```
/* Global styling with Google Fonts integration */

@import
url('https://fonts.googleapis.com/css2?family=Poppins:wght@300;400;500;600;700&disp
lay=swap');



/* Responsive breakpoints */

@media (max-width: 768px) {

    .main-title {

        font-size: 2.5rem;  /* Reduced from 3.5rem */

    }

    .sub-title {

        font-size: 1.2rem;  /* Reduced from 1.4rem */

    }

}
```

Layout Responsiveness

Grid System

```
col1, col2, col3 = st.columns([1, 2, 1])

with col2:

    st.markdown('<h3 class="features-title">✨ Explore Features</h3>',
unsafe_allow_html=True)
```

The application uses Streamlit's column system with responsive ratios:

- **Desktop**: Three-column layout with emphasis on center content
- **Mobile**: Automatically collapses to single-column layout
- **Tablet**: Adaptive column widths for optimal content flow

Navigation Features

- **Full-Width Buttons**: Optimized for touch interaction
- **Clear Labeling**: Descriptive button text for functionality clarity
- **State Management**: Maintains user's current location in application flow

Performance Optimization

Loading Strategy

- **Lazy Loading**: Components load only when accessed
- **Session Caching**: Frequently accessed data cached in session state
- **API Optimization**: Minimal API calls with caching

Docker Integration

Container Configuration

```
FROM python:3.9-slim

COPY frontend/ /app/

RUN pip install -r requirements.txt

EXPOSE 8501

CMD ["streamlit", "run", "app.py", "--server.port=8501",
"--server.address=0.0.0.0"]
```

Multi-Service Architecture

```
# docker-compose.yml integration

frontend:

  build: ./frontend

  ports:

    - "8501:8501"

  environment:

    - BACKEND_URL=http://backend:8000

  depends_on:

    - backend
```

The frontend is part of a comprehensive multi-service architecture:

- **Service Discovery**: Automatic backend service discovery
- **Environment Configuration**: Dynamic API endpoint configuration
- **Health Monitoring**: Integration with service health checks

FUTURE ENHANCEMENT:

Current Challenges:

The Heavy Model Problem

Currently, the system trains and stores 7 different machine learning models for each prediction horizon (24h, 48h, 72h). This creates several issues:

Storage Challenges

- **Total Models**: 21 models (7 models × 3 horizons)
- **Storage Size**: Several GB of model files
- **Hopsworks Cost**: High storage costs in feature store
- **Version Management**: Multiple model versions add complexity

Deployment Issues

- **Container Size**: Large Docker images due to multiple models
- **Memory Usage**: Even optimized, still significant memory footprint
- **Update Time**: Slow model updates due to large file transfers

Planned Deep Learning Upgrade

The future goal is to **transition to a single deep learning model** capable of predicting all horizons in one unified architecture.

Expected Benefits

- **Single Model**: One model file instead of 21 separate ones on Hopsworks model registry.
- **Weight Updates**: Ability to incrementally train on existing weights.
- **Smaller Size**: Typically **10–100 MB** vs several GB for multiple models.
- **Faster Training**: Leverage transfer learning for quicker convergence.

Implementation Plan

1. **Model Architecture Design** – Build a multi-horizon neural network.
2. **Weight Migration** – Transfer learned patterns from current models.
3. **Incremental Training** – Enable continuous learning from new data.

4. **Deployment Pipeline** – Automate container updates with new weights.

5. **Performance Monitoring** – Track and maintain prediction accuracy after deployment.

Conclusion

AirLens successfully demonstrates production-ready air quality forecasting with significant technical innovations in feature selection methodology and system optimization. The hybrid MI-RF approach and architectural optimizations provide a scalable foundation for environmental monitoring applications, while the planned deep learning transition addresses current storage and deployment challenges.

The system delivers immediate public health value through accurate predictions and real-time alerts, establishing a comprehensive framework for urban air quality management in Pakistan's major cities.