AQI Prediction System

End-to-End Implementation Guide

A Comprehensive Serverless Machine Learning Pipeline

Executive Summary

This document provides a complete implementation guide for building an end-to-end Air Quality Index (AQI) prediction system using a 100% serverless architecture. The system leverages modern MLOps practices to deliver accurate 3-day AQI forecasts through automated data pipelines, feature engineering, model training, and real-time web dashboards.

Project Objectives

1. Develop automated data collection from external weather and pollutant APIs
2. Implement robust feature engineering with time-based and derived features
3. Build and compare multiple ML models (statistical and deep learning approaches)
4. Create fully automated CI/CD pipelines for continuous model improvement
5. Deploy interactive web dashboard with real-time predictions and alerts

Key Deliverables

- Scalable feature and training pipelines with cloud-based storage
- Multiple ML models evaluated on RMSE, MAE, and R^2 metrics
- Automated hourly data collection and daily model retraining
- Interactive dashboard with forecasts, EDA insights, and hazard alerts
- Comprehensive documentation and explainability (SHAP/LIME)

System Architecture

The AQI prediction system follows a modern MLOps architecture with four main components working together in a serverless environment.

Architecture Components

Component	Description
Feature Pipeline	Fetches weather/pollutant data from APIs, engineers features, and stores in Feature Store (Hopsworks/Vertex AI)
Training Pipeline	Retrieves historical data, trains/evaluates models (Random Forest, LSTM, etc.), stores best model in registry
CI/CD Pipeline	Automated scheduling via Apache Airflow or GitHub Actions: feature pipeline (hourly), training pipeline (daily)
Web Dashboard	Streamlit/Gradio frontend with Flask/FastAPI backend. Displays predictions, EDA visualizations, and hazard alerts

Technology Stack

Core Technologies

Category	Tools & Frameworks
Programming	Python 3.9+, NumPy, Pandas, Scikit-learn
Deep Learning	TensorFlow 2.x / PyTorch, Keras
Feature Store	Hopsworks (recommended) or Google Vertex AI Feature Store
Orchestration	Apache Airflow or GitHub Actions
Web Framework	Streamlit (frontend), Flask or FastAPI (backend)
Data APIs	AQICN, OpenWeather, or alternative air quality APIs
Explainability	SHAP, LIME for model interpretation
Version Control	Git, GitHub/GitLab for code and CI/CD integration

Implementation Guide

Phase 1: Feature Pipeline Development

1.1 API Integration

Select and integrate with an air quality API to fetch raw weather and pollutant data.

Recommended APIs:

- AQICN (https://aqicn.org/api/) - Global AQI data with free tier
- OpenWeather Air Pollution API - Comprehensive pollutant data
- Alternative: EPA AirNow (US only), PurpleAir, IQAir

Sample Implementation:

```
import requests
import pandas as pd
from datetime import datetime


def fetch_aqi_data(api_key, city, date):
    url = f'https://api.waqi.info/feed/{city}/'
    params = {'token': api_key}
    response = requests.get(url, params=params)
    data = response.json()
    return parse_aqi_response(data)
```

1.2 Feature Engineering

Transform raw API data into meaningful features for model training.

Feature Categories:

1. **Time-based features:**
   - Hour of day (0-23), day of week (0-6), month (1-12), season
   - Weekend indicator, rush hour flags
   - Cyclical encoding (sin/cos transforms for hour, month)
2. **Pollutant features:**
   - PM2.5, PM10, O3, NO2, SO2, CO levels
   - Rolling averages (3h, 6h, 12h, 24h windows)
   - Standard deviations and variance metrics
3. **Weather features:**
   - Temperature, humidity, pressure, wind speed/direction
   - Precipitation, cloud cover, visibility
4. **Derived features:**
   - AQI change rate (hour-over-hour, day-over-day)
   - Lagged features (AQI values from 1-7 days prior)
   - Interaction terms (temperature * humidity, wind * pollution)

Feature Engineering Code:

```
def engineer_features(df):
    # Time features
    df['hour'] = df['timestamp'].dt.hour
    df['day_of_week'] = df['timestamp'].dt.dayofweek
```

```
    df['month'] = df['timestamp'].dt.month

    # Cyclical encoding
    df['hour_sin'] = np.sin(2*np.pi*df['hour']/24)
    df['hour_cos'] = np.cos(2*np.pi*df['hour']/24)

    # Rolling statistics
    df['pm25_rolling_3h'] = df['pm25'].rolling(3).mean()
    df['pm25_rolling_24h'] = df['pm25'].rolling(24).mean()

    # Change rates
    df['aqi_change_1h'] = df['aqi'].diff(1)
    df['aqi_change_24h'] = df['aqi'].diff(24)

    return df
```

1.3 Feature Store Integration

Store processed features in a centralized feature store for versioning and reusability.

Hopsworks Implementation:

```
import hopsworks

project = hopsworks.login()
fs = project.get_feature_store()

# Create feature group
aqi_fg = fs.get_or_create_feature_group(
    name='aqi_features',
    version=1,
    primary_key=['city', 'timestamp'],
    event_time='timestamp'
)

# Insert features
aqi_fg.insert(features_df)
```

Phase 2: Historical Data Backfill

Generate training datasets by running the feature pipeline for historical dates.

Backfill Strategy:

1. Determine required historical range (typically 1-3 years for seasonal patterns)
2. Handle API rate limits with batching and exponential backoff
3. Implement data quality checks for missing values and outliers
4. Store processed historical features in Feature Store

Backfill Script:

```
from datetime import datetime, timedelta
import time

def backfill_historical_data(start_date, end_date, city):
    current_date = start_date

    while current_date <= end_date:
        try:
            # Fetch data for date
            raw_data = fetch_aqi_data(API_KEY, city, current_date)

            # Engineer features
            features = engineer_features(raw_data)

            # Store in feature store
            aqi_fg.insert(features)

            current_date += timedelta(days=1)
            time.sleep(1)  # Rate limit protection

        except Exception as e:
            print(f'Error on {current_date}: {e}')
            time.sleep(5)
```

Phase 3: Training Pipeline Implementation

3.1 Data Preparation

```
# Fetch historical features
feature_view = fs.get_feature_view('aqi_features', version=1)
training_data = feature_view.get_batch_data()

# Split features and target
X = training_data.drop(['aqi'], axis=1)
y = training_data['aqi']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, shuffle=False)
```

3.2 Model Experimentation

Implement and compare multiple modeling approaches:

A. Statistical Models:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge

# Random Forest
rf_model = RandomForestRegressor(n_estimators=100, max_depth=15)
rf_model.fit(X_train, y_train)

# Ridge Regression
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_train, y_train)
```

B. Deep Learning Models:

```
import tensorflow as tf
from tensorflow import keras

# LSTM for time series
model = keras.Sequential([
    keras.layers.LSTM(64, return_sequences=True),
    keras.layers.Dropout(0.2),
    keras.layers.LSTM(32),
    keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50, batch_size=32)
```

3.3 Model Evaluation

```
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score

def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)

    rmse = np.sqrt(mean_squared_error(y_test, predictions))
    mae = mean_absolute_error(y_test, predictions)
    r2 = r2_score(y_test, predictions)

    return {'RMSE': rmse, 'MAE': mae, 'R2': r2}
```

3.4 Model Registry

```
# Register best model
mr = project.get_model_registry()

aqi_model = mr.python.create_model(
    name='aqi_predictor',
    metrics={'rmse': rmse, 'mae': mae, 'r2': r2}
)

aqi_model.save('model_dir')
```

Phase 4: Automated CI/CD Pipeline

4.1 GitHub Actions Implementation

Create automated workflows for continuous data collection and model training.

Feature Pipeline Schedule (.github/workflows/feature-pipeline.yml):

```
name: Feature Pipeline
on:
  schedule:
    - cron: '0 * * * *'  # Every hour
  workflow_dispatch:

jobs:
  run-feature-pipeline:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'
      - name: Install dependencies
        run: pip install -r requirements.txt
      - name: Run feature pipeline
        env:
          API_KEY: ${{ secrets.AQI_API_KEY }}
        run: python pipelines/feature_pipeline.py
```

Training Pipeline Schedule (.github/workflows/training-pipeline.yml):

```
name: Training Pipeline
on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM
  workflow_dispatch:

jobs:
  run-training-pipeline:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run training pipeline
        run: python pipelines/training_pipeline.py
```

4.2 Apache Airflow Alternative

For more complex orchestration requirements, Apache Airflow provides advanced scheduling and monitoring.

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'retries': 3,
    'retry_delay': timedelta(minutes=5)
}

with DAG('aqi_feature_pipeline',
         default_args=default_args,
         schedule_interval='@hourly') as dag:

    fetch_task = PythonOperator(
        task_id='fetch_aqi_data',
        python_callable=fetch_aqi_data)

    engineer_task = PythonOperator(
        task_id='engineer_features',
        python_callable=engineer_features)

    fetch_task >> engineer_task
```

Phase 5: Web Application Dashboard

5.1 Streamlit Frontend

```
import streamlit as st
import pandas as pd
import plotly.express as px

st.title('AQI Prediction Dashboard')

# Load model and features
model = load_model_from_registry()
features = load_latest_features()

# Make predictions
predictions = model.predict(features)

# Display forecast
st.subheader('3-Day AQI Forecast')
fig = px.line(predictions, x='date', y='aqi')
st.plotly_chart(fig)

# Alert system
if predictions['aqi'].max() > 150:
    st.error('Unhealthy AQI levels predicted!')
```

5.2 Feature Importance Visualization

```
import shap

# SHAP explainer
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(features)

st.subheader('Feature Importance')
fig = shap.summary_plot(shap_values, features)
st.pyplot(fig)
```

Advanced Features & Best Practices

Exploratory Data Analysis (EDA)

Conduct thorough EDA to understand data patterns and inform feature engineering:

- Time series decomposition (trend, seasonality, residuals)
- Correlation analysis between pollutants and weather variables
- Seasonal and weekly patterns visualization
- Outlier detection and anomaly analysis
- Distribution analysis of target variable (AQI)

Model Selection Strategy

Model Type	Strengths	Use Cases
Random Forest	Handles non-linear relationships, provides feature importance, robust to outliers	Baseline model, interpretability needed
LSTM	Captures temporal dependencies, learns sequence patterns	Long-term trends, complex temporal patterns
XGBoost	High accuracy, handles missing data, fast training	Production deployment, performance critical
Prophet	Seasonal decomposition, handles holidays, interpretable	Strong seasonal patterns, quick prototyping

Alert System Implementation

```
def check_air_quality_alerts(aqi_value):
    if aqi_value <= 50:
        return 'Good', 'green'
    elif aqi_value <= 100:
        return 'Moderate', 'yellow'
    elif aqi_value <= 150:
        return 'Unhealthy for Sensitive Groups', 'orange'
    elif aqi_value <= 200:
        return 'Unhealthy', 'red'
    elif aqi_value <= 300:
        return 'Very Unhealthy', 'purple'
    else:
        return 'Hazardous', 'maroon'
```

Project Structure & Organization

Recommended Directory Structure:

```
aqi-prediction-system/
├── pipelines/
│   ├── feature_pipeline.py
│   ├── training_pipeline.py
│   └── backfill.py
├── models/
│   ├── random_forest.py
│   ├── lstm.py
│   └── xgboost_model.py
├── features/
│   ├── feature_engineering.py
│   └── feature_store.py
├── app/
│   ├── streamlit_app.py
│   ├── api.py  # FastAPI backend
│   └── utils.py
├── notebooks/
│   ├── eda.ipynb
│   └── model_experiments.ipynb
├── .github/workflows/
│   ├── feature-pipeline.yml
│   └── training-pipeline.yml
├── tests/
│   └── test_pipelines.py
├── requirements.txt
├── README.md
└── config.yaml
```

Final Deliverables Checklist

1. **End-to-End System Components:**
   - Functional feature pipeline with API integration
   - Historical data backfill script
   - Training pipeline with multiple model implementations
   - Model registry integration
   - Feature store implementation (Hopsworks/Vertex AI)
2. **Automated Pipeline:**
   - Hourly feature pipeline automation (GitHub Actions/Airflow)
   - Daily training pipeline automation
   - Error handling and retry logic
   - Monitoring and logging setup
3. **Interactive Dashboard:**
   - Streamlit frontend with clean UI
   - 3-day forecast visualization
   - Historical AQI trends
   - Real-time model predictions
   - Hazardous level alerts
   - SHAP/LIME feature importance visualizations
4. **Comprehensive Documentation:**
   - Detailed README with setup instructions
   - Architecture diagram
   - EDA findings and insights
   - Model comparison results (RMSE, MAE, R^2)
   - Feature engineering rationale
   - Deployment and CI/CD documentation
   - Future improvements and limitations

Resources & References

API Documentation

AQICN API - Global air quality data
OpenWeather Air Pollution API - Weather and pollutant data

MLOps Platforms

Hopsworks Documentation - Feature store and model registry
Google Vertex AI - Alternative feature store

Frameworks & Libraries

Scikit-learn Documentation - Machine learning models
TensorFlow Documentation - Deep learning framework
Streamlit Documentation - Web dashboard framework
SHAP Documentation - Model explainability

CI/CD Tools

GitHub Actions - Automation and CI/CD
Apache Airflow - Workflow orchestration

Conclusion

This comprehensive guide provides a complete roadmap for building a production-ready AQI prediction system. By following the phased implementation approach, you will develop skills in modern MLOps practices, serverless architecture, and end-to-end machine learning pipeline development.

Key success factors include thorough exploratory data analysis, robust feature engineering, experimentation with multiple modeling approaches, and implementation of automated pipelines for continuous improvement. The interactive dashboard with model explainability features ensures transparency and builds user trust in the predictions.

Remember to iterate on your models, continuously monitor performance metrics, and incorporate feedback from stakeholders. This project serves as an excellent foundation for understanding real-world machine learning deployment challenges and modern MLOps best practices.

Good luck with your implementation!