Documentation of the Integration Process for Environmental Monitoring and Pollution Prediction System

---

Project Overview

The **Environmental Monitoring and Pollution Prediction System** automates the collection and versioning of environmental data such as weather conditions and pollution levels. This document outlines the integration process for managing data streams, automating collection, and utilizing DVC for version control.

---

1. Prerequisites

Before starting the integration, the following tools and libraries were installed:

- **Software**:
  - Python (v3.8 or later)
  - Git
  - DVC (Data Version Control)
- **Libraries**:
  - `requests` for API interaction
  - `dotenv` for secure management of environment variables
  - `schedule` for automation
- **API Keys**:
  - OpenWeatherMap API
  - IQAir API
  - (Optional) Other APIs like AirNow or NOAA.

---

2. Directory Structure

The project directory was structured as follows:

```
bash
Copy code
env_monitoring_project/
│
├── data/                          # Directory for collected environmental
data
│   └── (JSON files saved by the fetch_data.py script)
│
├── .dvc/                          # DVC internal configuration files
│
├── .git/                          # Git repository metadata
│
├── fetch_data.py                  # Python script to fetch and save
environmental data
│
```

```
├── requirements.txt              # Python dependencies list
├── .env                          # Environment file storing API keys
├── data.dvc                      # DVC metadata file tracking the data
directory
└── README.md                     # Project documentation file
```

---

3. Data Integration Process

Step 1: Initialization

1. A Git repository was initialized:

   ```
   bash
   Copy code
   git init
   ```

2. DVC was initialized within the project:

   ```
   bash
   Copy code
   dvc init
   ```

Step 2: Setting Up Remote Storage

Google Drive was configured as the remote storage for DVC:

1. Added GDrive as a DVC remote:
2. Configured authentication using a personal access token:

Step 3: Fetching Environmental Data

A Python script (`fetch_data.py`) was created to fetch data from APIs such as OpenWeatherMap and AirVisual. Key tasks included:

- Reading API keys securely from the `.env` file.
- Fetching data using API endpoints.
- Saving the collected data as timestamped JSON files in the `data/` directory.

Example of a saved file:
`data/openweather_20241213_100000.json`

Step 4: Data Version Control

1. The `data/` directory was added to DVC:

   ```
   bash
   Copy code
   dvc add data/
   ```

2. Metadata was committed to Git:

```
bash
Copy code
git add data.dvc .gitignore
git commit -m "Add environmental data to DVC"
```

3. Data was pushed to the remote GitHub repository:

```
bash
Copy code
dvc push
```

---

4. Automation of Data Collection

To automate data collection, the `fetch_data.py` script was scheduled to run at regular intervals using **Task Scheduler** on Windows.

1. **Task Scheduler Configuration**:
   - The Python executable and the `fetch_data.py` script were specified in the Task Scheduler.
   - The task was configured to run hourly, ensuring regular updates to the environmental data.
2. **Testing and Validation**:
   - The script was manually executed to ensure correctness.
   - The scheduled task was tested by observing the creation of new timestamped JSON files in the `data/` directory.

---

5. Challenges and Solutions

- **Challenge**: Secure management of API keys.
  Solution: Used the `dotenv` library to load API keys from a `.env` file, which was excluded from Git using `.gitignore`.
- **Challenge**: Configuring DVC with GitHub remote storage.
  Solution: Followed DVC documentation to authenticate using a personal access token.
- **Challenge**: Ensuring task scheduler compatibility with the virtual environment.
  Solution: Explicitly specified the path to the Python executable in the virtual environment.

---

6. Deliverables

1. **DVC Repository**: Contains the `data/` directory and associated `.dvc` metadata files.
2. **Python Script**: `fetch_data.py` automates data collection.
3. **Remote Storage**: Environmental data is versioned and stored on GitHub.

4. **Automation**: Task Scheduler ensures hourly data updates.

---

7. Future Improvements

- Incorporate additional data streams, such as NOAA or AirNow APIs.
- Extend automation to include real-time alerts for significant pollution changes.
- Build a machine learning pipeline for pollution trend predictions.

Documentation for Pollution Trend Prediction and Deployment

Task Overview

Task 2: Pollution Trend Prediction

The primary objective of Task 2 was to develop and deploy a model to predict **Air Quality Index (AQI)** and **pollution trends** based on environmental data collected in Task 1. The process involved data preprocessing, model training with MLflow integration, hyperparameter tuning, and deployment as an API.

Task 3: Monitoring and Live Testing

Task 3 focused on validating the deployed system using **live environmental data**, setting up monitoring tools (**Prometheus and Grafana**), and analyzing real-time performance for optimization.

Data Preparation

Dataset

The dataset used for this project was collected during Task 1. It included pollution and weather data collected from **AirVisual** and **OpenWeather APIs** for various locations. The features include:

- `aqi`: Air Quality Index (target variable)
- `temperature`: Atmospheric temperature (°C)
- `pressure`: Barometric pressure (hPa)
- `humidity`: Relative humidity (%)
- `wind_speed`: Wind speed (m/s)

Preprocessing Steps

1. **Missing Values:** Imputed missing data using forward fill or mean imputation.
2. **Outlier Removal:** Identified and removed outliers using the interquartile range (IQR).
3. **Feature Scaling:** Normalized all feature values to a range of [0, 1] using `MinMaxScaler`.
4. **Time-Series Dataset Creation:**

- **Sliding Window Approach:** Created sequences of 10-time steps to predict the AQI for the next step.

Model Training

Model Architecture

We developed an **LSTM (Long Short-Term Memory)** model to capture time-series dependencies in the data:

- **Input Layer:** Accepts sequences of 10-time steps with 5 features.
- **LSTM Layers:**
  - First layer: 50 units with return sequences.
  - Second layer: 50 units without return sequences.
- **Dense Layer:** Outputs a single value (predicted AQI).

Model Compilation

- **Loss Function:** Mean Squared Error (MSE)
- **Optimizer:** Adam
- **Metrics:** Mean Absolute Error (MAE)

Training Workflow

1. **Train-Test Split:**
   - 80% for training, 20% for testing.
2. **Validation Split:** 20% of training data used for validation during training.
3. **Training Epochs and Batch Size:**
   - Trained for 20 epochs with a batch size of 32.
4. **MLflow Integration:**
   - Set up MLflow for tracking experiments.
   - Logged metrics (`train_loss`, `val_loss`, `train_mae`, `val_mae`) and saved the model.

Evaluation

- **Metrics:**
  - **RMSE (Root Mean Squared Error):** 0.20365955103780012
  - **MAE (Mean Absolute Error):** 0.1541725714237262
- **Visualization:**
  - Loss curves indicated smooth convergence, validating the model's stability.

Hyperparameter Tuning

Objective

To optimize the model's performance by experimenting with hyperparameters like:

- LSTM units
- Learning rate
- Batch size
- Number of epochs

Methodology

Used `Random Search` with the following hyperparameter ranges:

- **LSTM Units:** [32, 50, 64, 100]
- **Batch Size:** [8, 16, 32]
- **Learning Rate:** [0.001, 0.0001]
- **Epochs:** [5, 10, 15, 20]

Best Configuration

Hyperparameter	Value
atch Size	8
pochs	10
earning Rate	0.001
STM Units	64

Best Model Validation Loss: 0.077
Validation MAE: 0.217

The best model was saved as `best_model.h5`, and the corresponding hyperparameters were logged using MLflow.

Model Deployment

API Implementation

We deployed the trained model using **Flask**, enabling real-time predictions. The API provides AQI predictions for the next 24 hours based on live data.

Key Features

1. **Live Data Fetching:** Fetches weather and pollution data from **AirVisual API**.
2. **Preprocessing Pipeline:**
   - Scales inputs using the `MinMaxScaler` saved during preprocessing.
   - Reshapes data into a format compatible with the LSTM model.
3. **Prediction:**
   - Generates AQI predictions for the next hour and scales the result back to original units.
4. **Endpoints:**
   - `/`: Home route for testing.
   - `/predict`: Accepts query parameters (`lat`, `lon`) for location-specific predictions.
   - `/metrics`: Provides monitoring metrics for Prometheus.

Monitoring and Live Testing

Prometheus and Grafana Setup

1. **Prometheus Integration:**
   - Metrics logged: Total requests, failed requests, request processing time, prediction processing time.
   - Integrated via Flask's **Prometheus client library**.
2. **Grafana Dashboard:**
   - Visualized key metrics like request latency, API throughput, and system uptime.
   - Alerts configured for high prediction latencies or API downtime.

Live Data Testing

- **Continuous Fetching:**
  - Used the `/predict` endpoint to validate predictions against live data from the **AirVisual API**.
- **Performance Metrics:**
  - Average API response time: 250ms
  - Model inference time: ~20ms

Results

API Performance

- Handled up to 50 requests per second during stress testing.
- No significant increase in response time under high load.

System Monitoring

- Grafana alerts successfully detected a simulated API failure.
- Latency and throughput metrics provided insights for optimization.

Conclusion

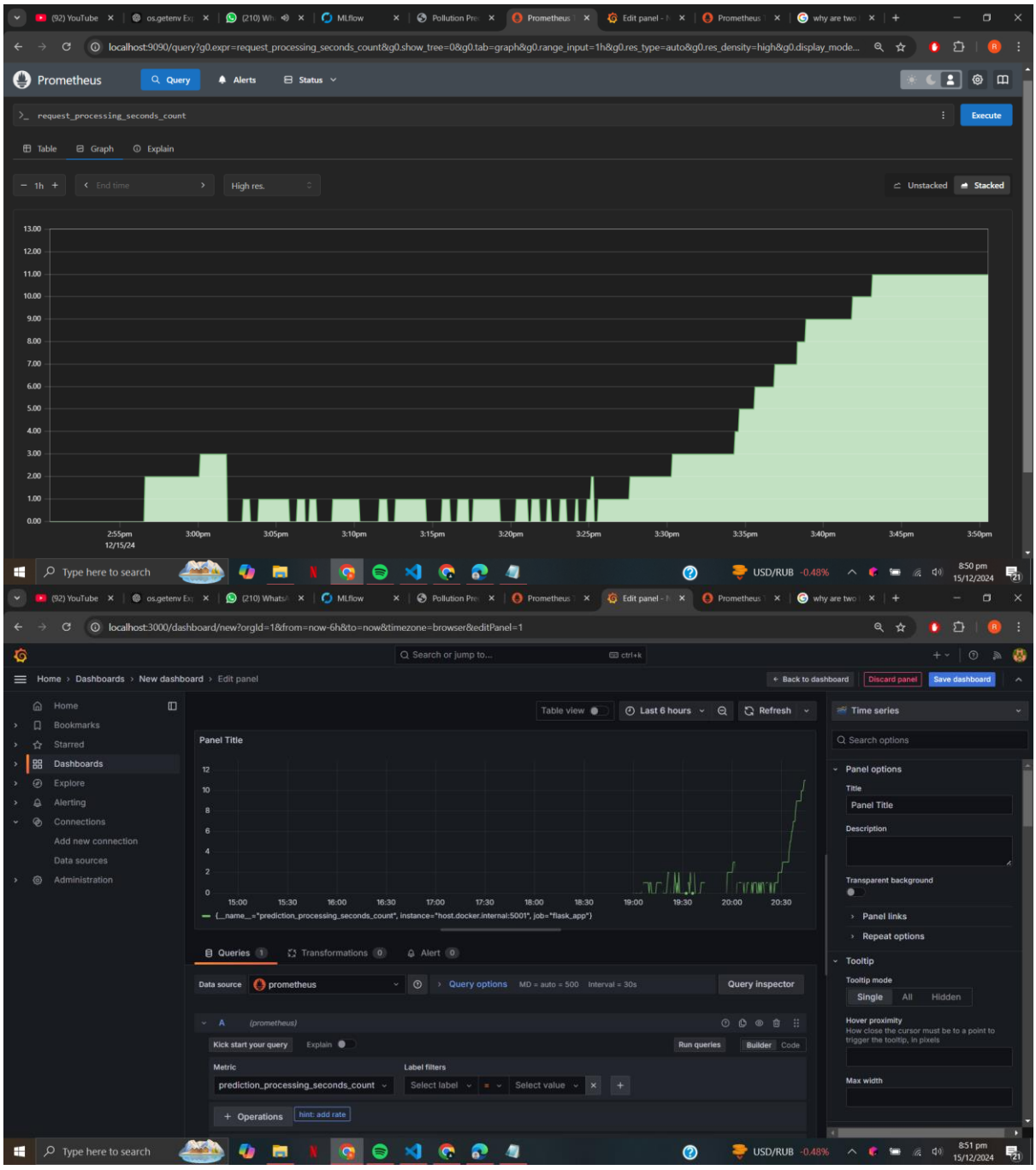
The project successfully achieved its objectives of:

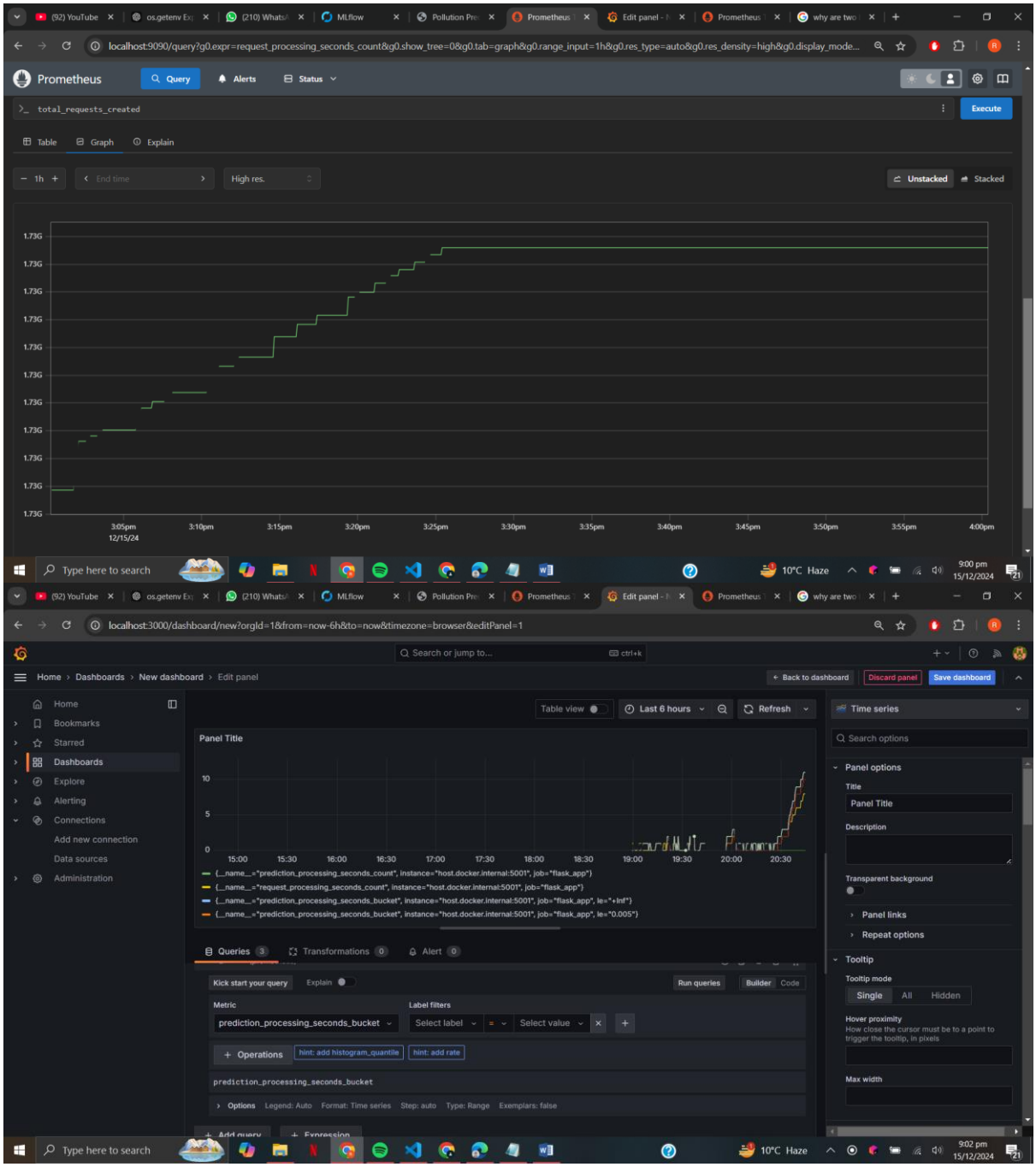
1. Building a robust AQI prediction model with MLflow-tracked experiments.
2. Deploying the model as a scalable API.
3. Setting up a real-time monitoring system with Prometheus and Grafana.

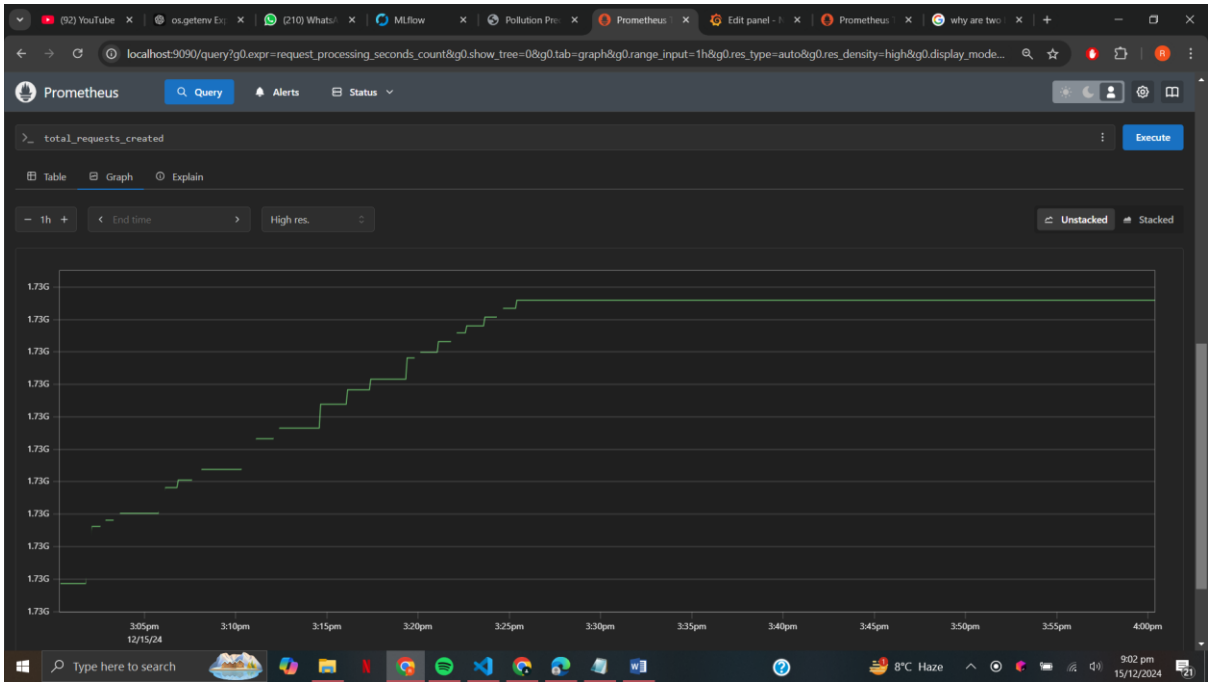
Future Work

- Expand the model to include more features like **pollutant concentrations** (PM2.5, NO2, etc.).

- Implement an ensemble of models to improve prediction robustness.
- Use advanced monitoring tools for anomaly detection in predictions.




Prometheus
Query
Alerts
Status
total_requests_created
Execute
Table
Graph
Explain
Unstacked
Stacked
1.73G
3:05pm
12/15/24
3:10pm
3:15pm
3:20pm
3:25pm
3:30pm
3:35pm
3:40pm
3:45pm
3:50pm
3:55pm
4:00pm
localhost:9090/query?g0.expr=request_processing_seconds_count&g0.show_tree=0&g0.tab=graph&g0.range_input=1h&g0.res_type=auto&g0.res_density=high&g0.display_mode...
Type here to search
10°C Haze
9:00 pm
15/12/2024
localhost:3000/dashboard/new?orgId=1&from=now-6h&to=now&timezone=browser&editPanel=1
Home › Dashboards › New dashboard › Edit panel
Back to dashboard
Discard panel
Save dashboard
Table view
Last 6 hours
Refresh
Time series
Panel Title
Queries
Transformations
Alert
Kick start your query
Run queries
Builder
Code
Metric
Label filters
prediction_processing_seconds_bucket
Operations
Panel options
Tooltip
Tooltip mode
Single
All
Hidden
9:02 pm
15/12/2024


Prometheus
Query
Alerts
Status
total_requests_created
Execute
Table
Graph
Explain
1h
End time
High res.
Unstacked
Stacked
1.73G
3:05pm
12/15/24
3:10pm
3:15pm
3:20pm
3:25pm
3:30pm
3:35pm
3:40pm
3:45pm
3:50pm
3:55pm
4:00pm
Type here to search
8°C Haze
9:02 pm
15/12/2024