**Environmental Monitoring and Pollution Prediction System**

**Introduction**

This project involves the development of an MLOps pipeline to monitor environmental data such as air quality, weather, and pollution levels, and predict pollution trends. To achieve this, I utilized the OpenWeatherMap API for fetching weather, air pollution, and forecast data. A virtual environment (venv) was set up to isolate dependencies for the project.

---

**Task 1: Managing Environmental Data with DVC**

**Objective**

To use DVC (Data Version Control) for managing real-time environmental data streams collected from APIs.

---

**Step 1: Research Live Data Streams**

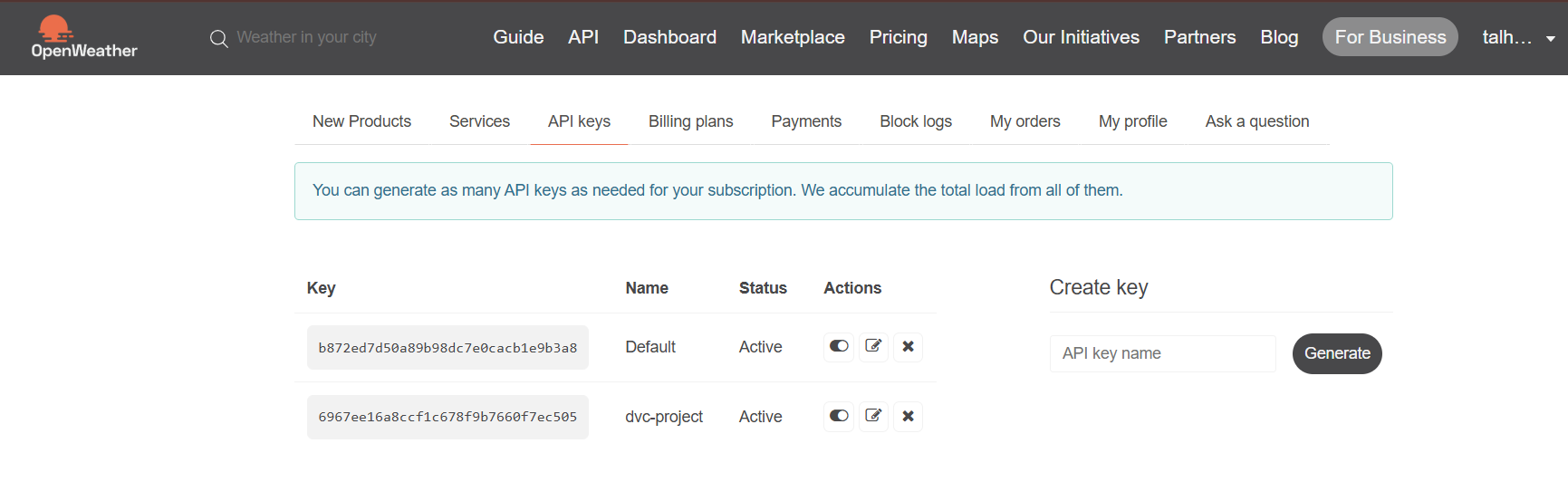
**Description:**

Identify and select a publicly available live data stream suitable for integration into the project. I chose the OpenWeatherMap API as it provides weather data, including temperature, humidity, and air pollution data. The API is free to use with a registered account.

**Process:**

1. Created an account on [OpenWeatherMap API](<https://openweathermap.org/api>) with university email.

2. Generated an API key to access the required data.



---

**Step 2: Set Up DVC Repository**

**Description:**

Initialize a DVC repository to manage version control for the collected data.

**Process:**

1. Ran `dvc init` to initialize the DVC repository.

---

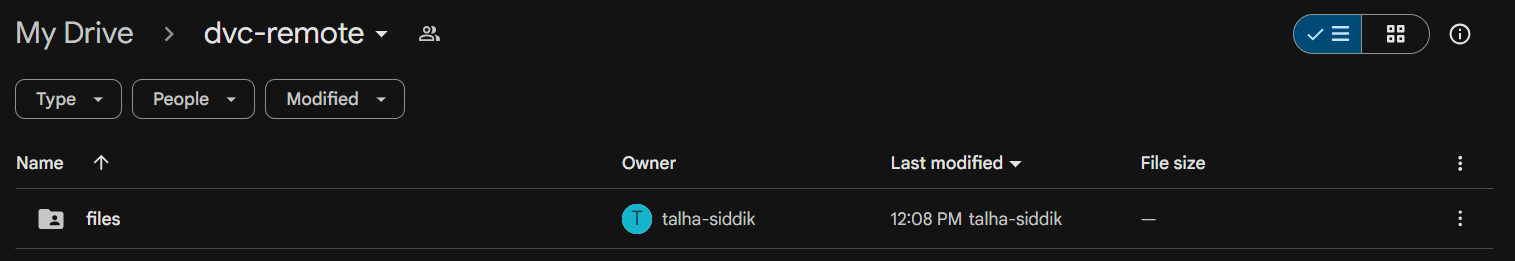
**Step 3: Remote Storage Configuration**

**Description:**

Configure remote storage using free solutions for seamless integration with DVC.

**Process:**

1. Used Google Drive as the remote storage (created a folder dvc-remote).



2. Set up a service account and shared driver folder access with it. generated gdrive-key.json for API.

3. Configured the remote storage by running:

dvc remote add -d gdrive\_remote gdrive://1zYp619eX\_ipzNjQrnps0JwTBXzic6Ncd

dvc remote modify gdrive\_remote gdrive\_service\_account\_json\_path D:/FAST/7th-Semester/MLOPS/Final-Project/gdrive-key.json

```



---

**Step 4: Data Collection Script**

**Description:**

Developed a Python script to fetch weather and air quality data from the OpenWeatherMap API. The script fetches data at regular intervals.

**Highlights:**

- The script is designed to automatically push updated data to DVC upon execution.

---

**Step 5: Version Control with DVC**

**Description:**

Used DVC commands to manage and version the collected data.

**Commands:**

1. `dvc add <data\_file>`: To stage data files.

2. `dvc commit`: To commit changes locally.

3. `dvc push`: To upload changes to remote storage.

---

**Step 6: Automate Data Collection**

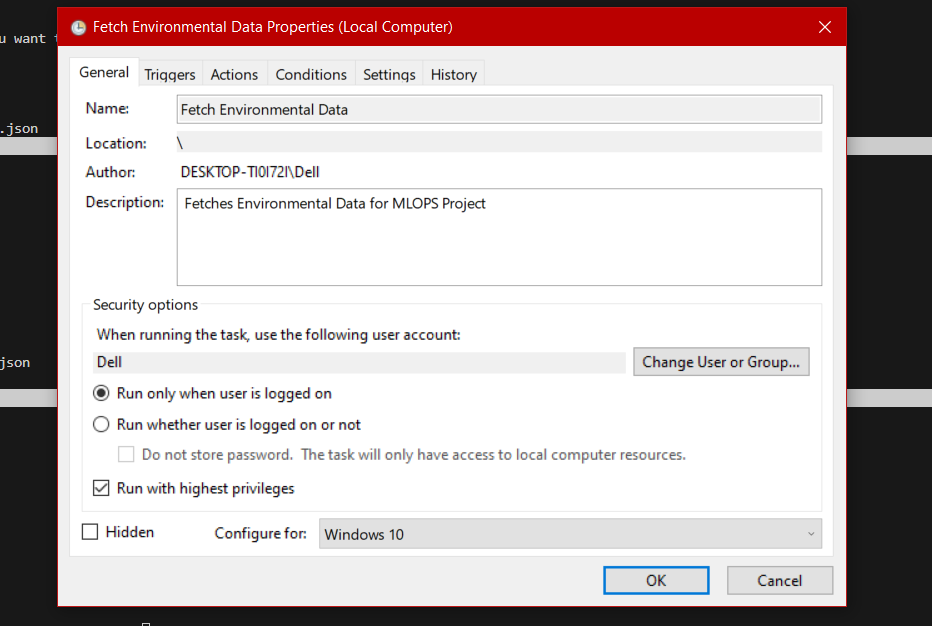
**Description:**

Automated the data collection process to run at regular intervals using a task scheduler.

**Process:**

1. Configured the Task Scheduler in Windows to execute the data collection script periodically.

2. Ensured the script automatically stages, commits, and pushes data to DVC during execution.

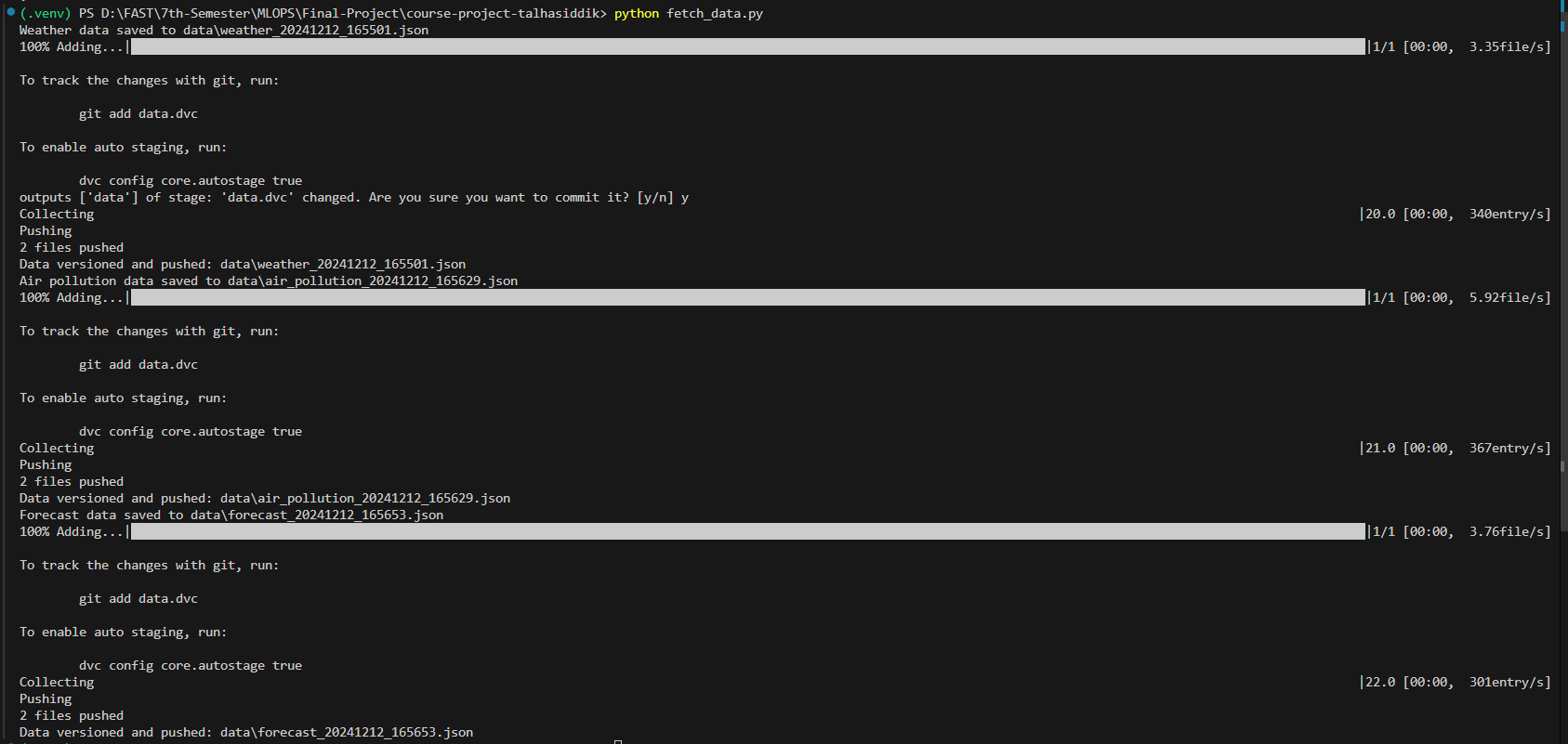


---

**Step 7: Update Data with DVC**

**Description:**

As new data is being fetched, the data directory in the DVC repository is updated to reflect the latest changes.



---

**Conclusion**

This project demonstrates the integration of DVC for managing real-time environmental data. The automation ensures seamless data collection and version control, paving the way for building predictive models. Screenshots provided in relevant sections showcase the implementation steps.

**Task 2: Pollution Trend Prediction with MLflow**

**Objective**

Develop and deploy models to predict pollution trends and alert high-risk days.

**1. Data Preparation**

**Steps:**

**Data Loading**

Imported one year environmental data from OpenWeatherMap including the data latest data being fetched in Task 1.

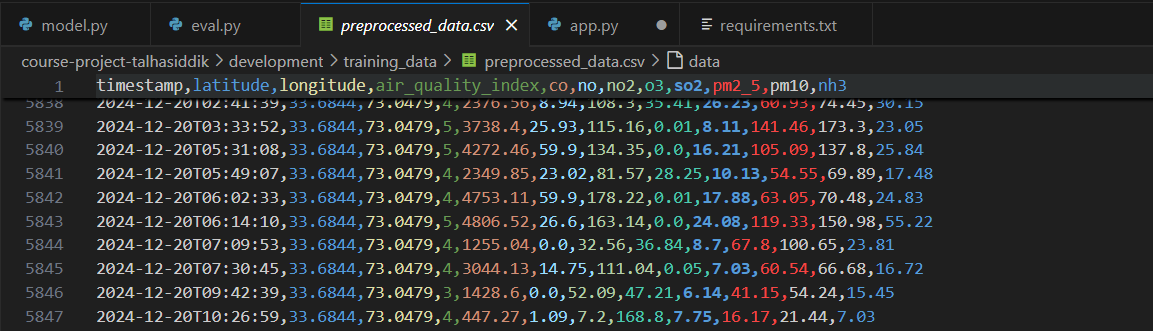
**Preprocessing**

Handling Missing Values: Replaced missing values with the median for each column.

Outlier Removal: Applied interquartile range (IQR) to filter outliers in pollution-related features.

Normalization: Scaled the data using Min-Max normalization for better model performance.

Removed duplicate values.



**2. Model Development**

**Approach:**

Used LSTM (Long Short-Term Memory) networks to predict pollution levels and AQI trends.

**Key Features:**

**Input Features:**

input\_example = pd.DataFrame({

    "co": [1295.09],

    "no": [3.1],

    "no2": [41.81],

    "o3": [100.14],

    "so2": [6.5],

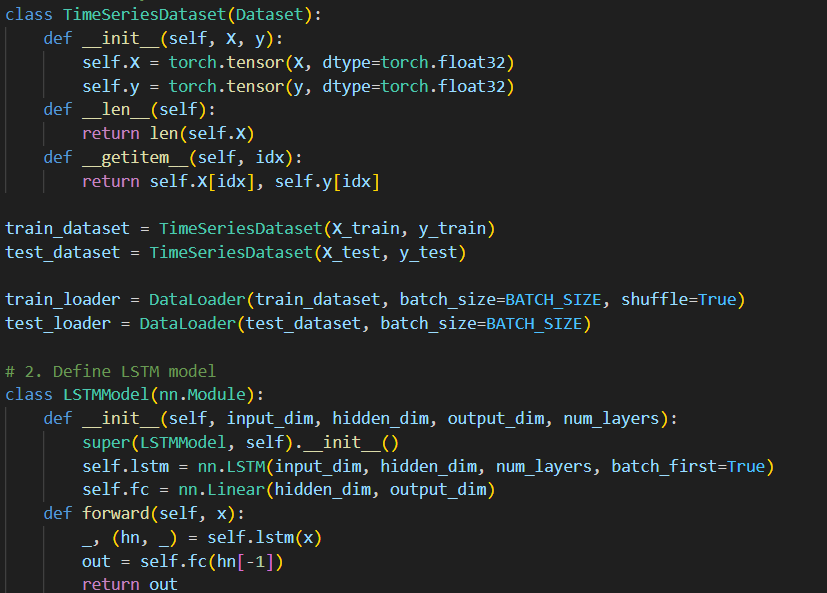
    "pm2\_5": [78.0],

    "pm10": [96.69],

    "nh3": [11.15],

})

Output Variable: air\_quality\_index



**3. Train Models with MLflow**

**Steps:**

**Experiment Tracking**

Set up MLflow tracking server locally.

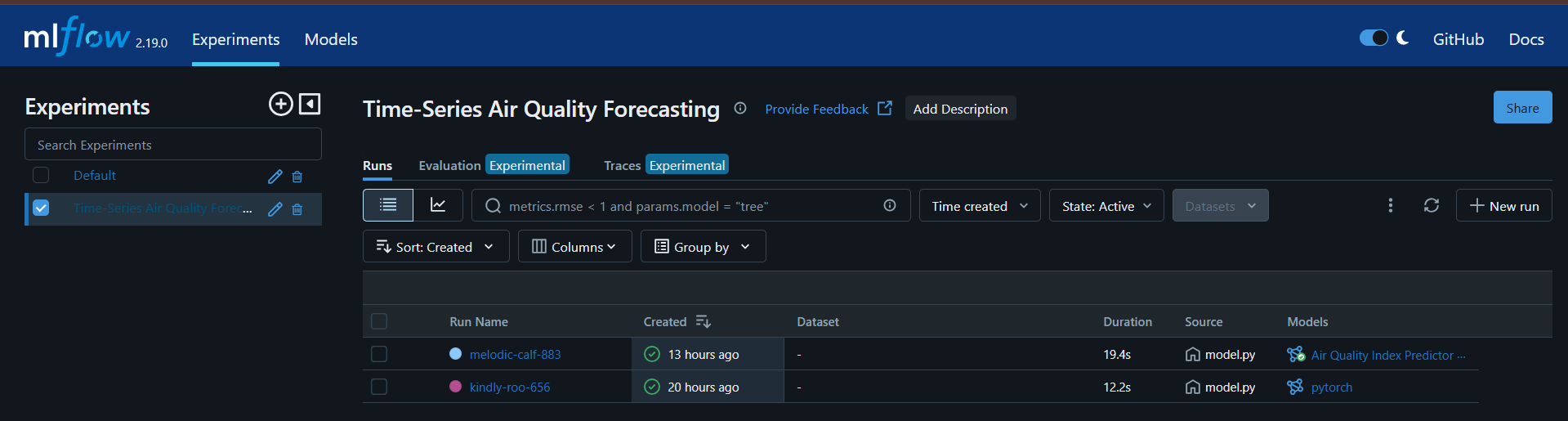
Logged metrics: RMSE, MAE, and accuracy.

Saved model artifacts and versioned them for easy comparison.

**Implementation:**

Integrated MLflow tracking in the training script.

Added parameters such as learning rate, batch size, and epochs.



**4. Hyperparameter Tuning**

**Methodology:**

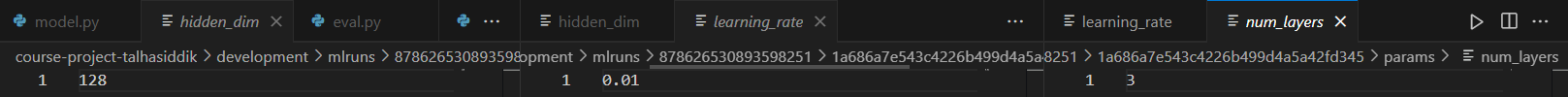
Used random search to find the optimal hyperparameters for the LSTM model.

Key Hyperparameters Tuned:

Learning rate.

Number of layers

Hidden dimensions.



**5. Model Evaluation**

**Steps:**

**Model Testing:**

Loaded the best model from MLflow using its artifact URI.

Converted the preprocessed data into time-series sequences for testing.

Split the data into training and testing sets, with 20% for testing.

Evaluation Metrics:

Calculated the Mean Squared Error (MSE) between predictions and actual values.

Generated a Confusion Matrix to assess classification performance.

**Results:**

Mean Squared Error (MSE)

Confusion Matrix: Visualized the confusion matrix using Seaborn heatmap.

**Visualization:**

Plotted the confusion matrix to analyze prediction accuracy.

**6. Deployment**

**Steps:**

**Model Loading:**

Used MLflow to load the trained LSTM model from its artifact URI.

Loaded the Min-Max scaler from a pickle file to preprocess incoming data.

**API Development:**

Developed a Flask API with the following endpoints:

Health Check (/): Confirms that the API is running.

Prediction (/predict): Accepts JSON input with numerical feature values and returns predictions.

**Preprocessing Pipeline:**

Normalized input data using the saved scaler.

Converted input data into sequences compatible with the LSTM model.

Ensured the input contains at least the minimum sequence length required by the model.

**Prediction Logic:**

Set the model to evaluation mode and predicted values for the processed input.

Returned predictions as JSON responses.

**Error Handling:**

Implemented exception handling to provide meaningful error messages for invalid or insufficient input data.

**Testing and Deployment:**

Tested the application using Thunder Client Extension.

Deployed the Flask application locally on port 5000.

A screen shot of a computer

Description automatically generated

**Task 3: Monitoring and Live Testing**

**Objective**

The goal of this task is to test the end-to-end pipeline with live data, monitor the deployed system, and analyze its performance using **Prometheus** for metrics collection and **Grafana** for visualization.

**Steps Performed**

**1. Setting Up Prometheus for Monitoring**

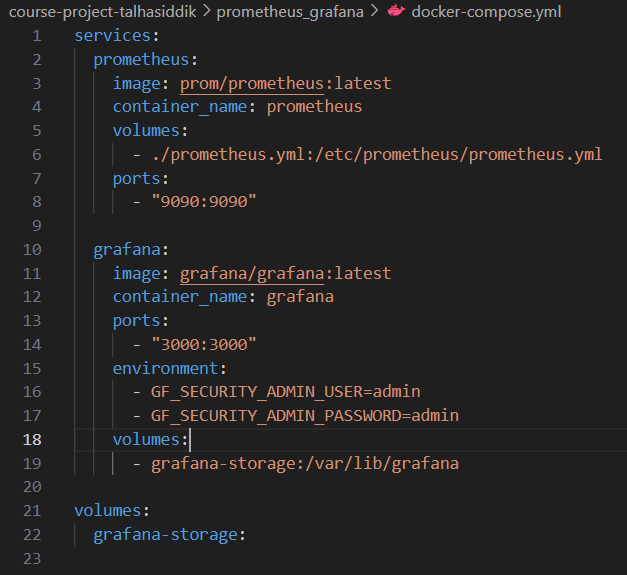
Prometheus was configured to monitor the Flask API and track relevant metrics.

* **Prometheus Installation**: Prometheus was set up using Docker to streamline deployment.
* **Configuration File**: A prometheus.yml file was created to define the scraping jobs. The configuration is as follows:

**2. Setting Up Grafana for Visualization**

Grafana was integrated with Prometheus as a data source to create live monitoring dashboards.

* **Grafana Installation**: Grafana was set up using Docker.



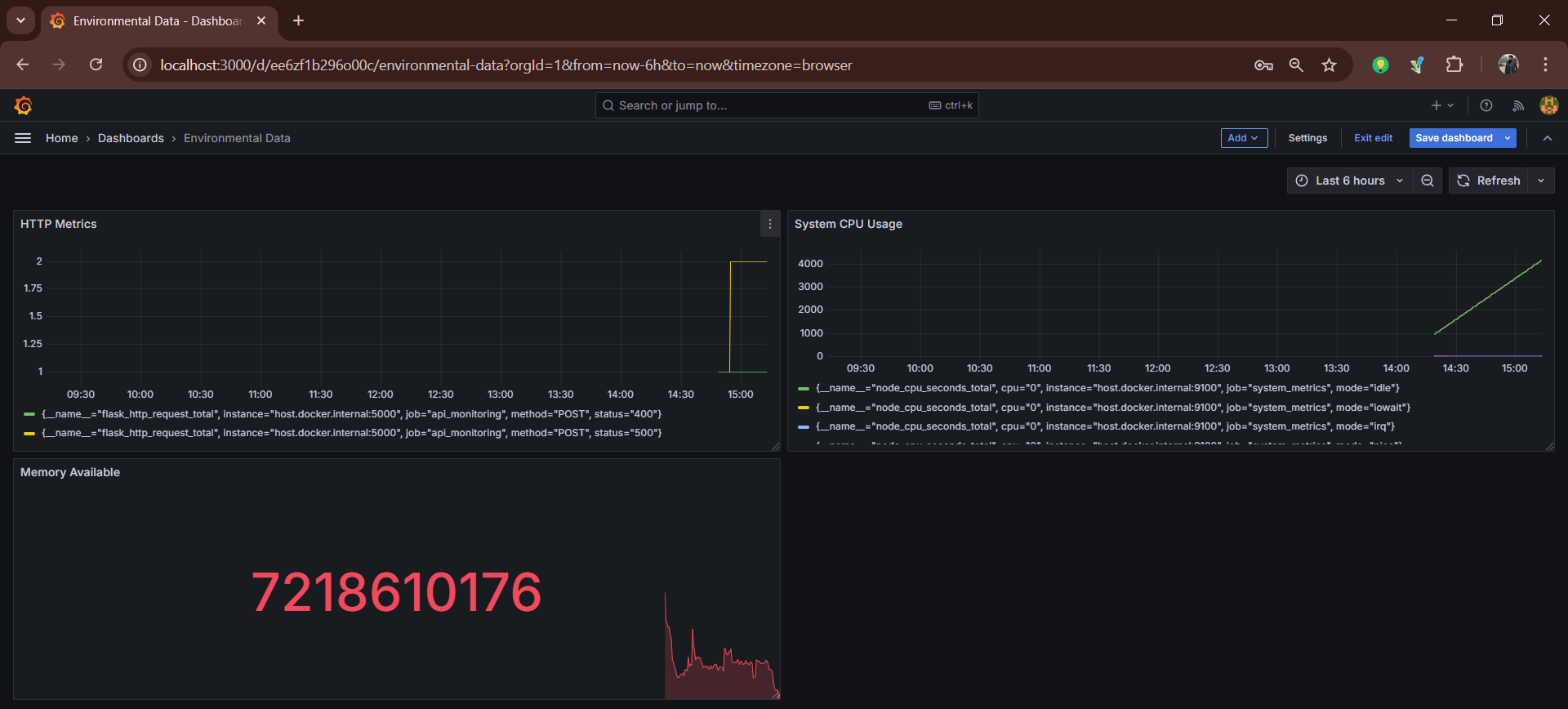
A screen shot of a computer

Description automatically generated

* **Adding Prometheus as a Data Source**:
  + Navigate to **Configuration** > **Data Sources** > **Add Data Source**.
  + Select **Prometheus** and configure the URL as http://host.docker.internal:9090.
* **Dashboard Creation**: A custom dashboard was created to visualize key metrics:

**Panels Created**

| **Panel Name** | **PromQL Query** | **Purpose** |
| --- | --- | --- |
| Total API Requests | sum(flask\_http\_request\_total) | Shows the total number of requests |
| Requests by HTTP Method | sum by (method) (flask\_http\_request\_total) | Visualizes requests grouped by method |
| Requests by Status Code | sum by (status) (flask\_http\_request\_total) | Shows requests grouped by status codes |
| Request Rate (Per Second) | rate(flask\_http\_request\_total[5m]) | Displays the rate of incoming requests |
| System CPU Usage | node\_cpu\_seconds\_total | Tracks CPU usage on the server |
| System Memory Usage | node\_memory\_MemAvailable\_bytes | Tracks available memory |



**3. Testing the Flask API with Live Data**

The Flask API endpoint /predict was tested with live environmental data fetched using the Python script developed in **Task 1**.

* **Input JSON Example**:

{

"data": [

[1295.09, 3.1, 41.81, 100.14, 6.5, 78.0, 96.69, 11.15, 0.0],

[1280.12, 3.2, 42.05, 101.00, 6.8, 79.5, 95.50, 11.20, 0.0],

[1275.30, 3.3, 41.90, 99.80, 6.4, 77.2, 97.00, 11.00, 0.0],

[1260.80, 3.1, 42.00, 98.70, 6.6, 76.5, 96.80, 11.10, 0.0],

[1255.50, 3.0, 41.85, 98.90, 6.7, 77.0, 97.20, 11.25, 0.0],

[1245.90, 3.2, 41.95, 98.50, 6.5, 76.8, 96.50, 11.05, 0.0],

[1240.60, 3.4, 41.70, 99.10, 6.3, 76.7, 97.10, 11.35, 0.0],

]

}

**4. Real-Time Performance Monitoring**

The following performance metrics were monitored using the Grafana dashboard:

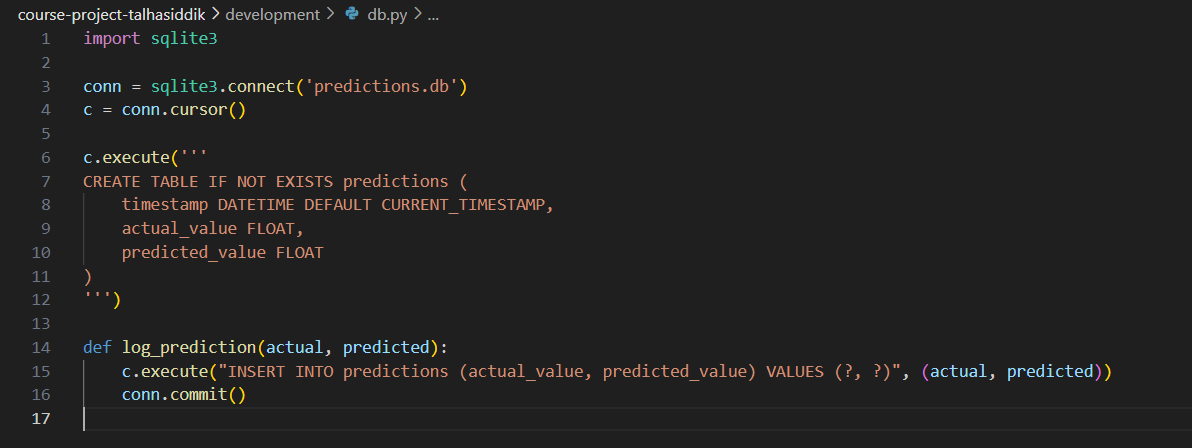
* **API Requests**:
  + Total requests, requests per method (GET, POST), and request status codes (200, 404, 500).
* **API Response Rate**:
  + Live rate of requests per second.
* **System Performance**:
  + CPU and memory usage of the system running the Flask API.

**5. Analysis and Optimization**

The system's performance and accuracy were analyzed:

* **API Performance**:
  + The rate of requests and successful responses were validated against the expected values.
* **System Resource Usage**:
  + Resource consumption was monitored to ensure the system could handle live data ingestion.
* **Model Accuracy**:
  + The prediction API was validated by testing the results against live environmental data.

**Saving the predictions in an SQLite**



**Conclusion**

The real-time performance monitoring was successfully implemented using **Prometheus** and **Grafana**. The deployed pollution prediction API was tested with live data, and its performance was analyzed using key metrics like request rates, status codes, and system resource usage.

The pipeline is now capable of live testing and monitoring, ensuring reliable predictions and stable performance.