CSC8112 - Internet of Things

Assessment 1 – Report - 230302626

November 2, 2023

Abstract

The developing Internet of Things (IoT) industry has the potential to revolutionise the way we interact with the digital world, with edge-cloud architecture playing a critical role in sensor data processing. This study describes the development and testing of a modular IoT data processing pipeline that analyses real-time urban data using lightweight virtualization via Docker and Python programming. The system built taps into the enormous sensor network of the Newcastle Urban Observatory, applying machine learning algorithms for predictive analytics and data visualisation. The study illustrates the creation of an IoT layer for data collecting and injection, an edge tier for preprocessing, and a cloud tier for advanced processing and visualisation by executing a series of more sophisticated tasks. This paper details the design, implementation, and outcomes procedures, which are supplemented with live demonstrations to demonstrate the efficacy of the proposed system.

1 Introduction

In the realm of IoT, data is king. The growth of IoT paradigms necessitates efficient data processing frameworks, especially when it comes to merging edge and cloud computing capabilities. This article digs into the development of such a pipeline for CSC8112, using Docker and Python to handle and analyse data from the Newcastle Urban Observatory. It covers the full workflow, from data collection with a Data Injector to data refining for machine learning algorithms and, finally, visualisation of fine particulate matter (PM2.5) concentrations.

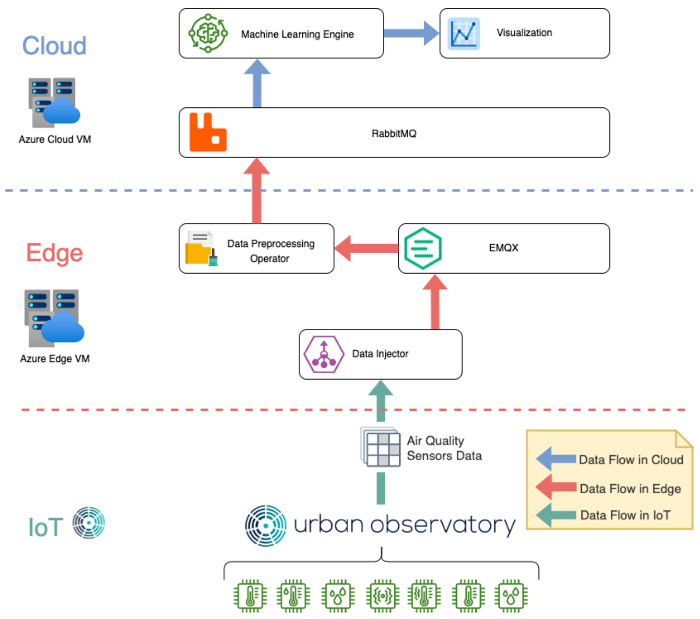


Figure 1 - Overall Overview Structure of System

Further, this document narrates the process of architecting an IoT solution that not only leverages virtualization technologies but also adapts to the dynamic nature of real-time sensor data processing. The endeavour includes crafting a Preprocessing Operator, developing a Predictive Model, and deploying a Visualization Tool, each a cog in the machine aimed at predicting air quality. Challenges encountered, solutions implemented, and lessons learned are woven into the fabric of this report, presenting a holistic view of the project's trajectory and outcomes.

2 Journey of the Project

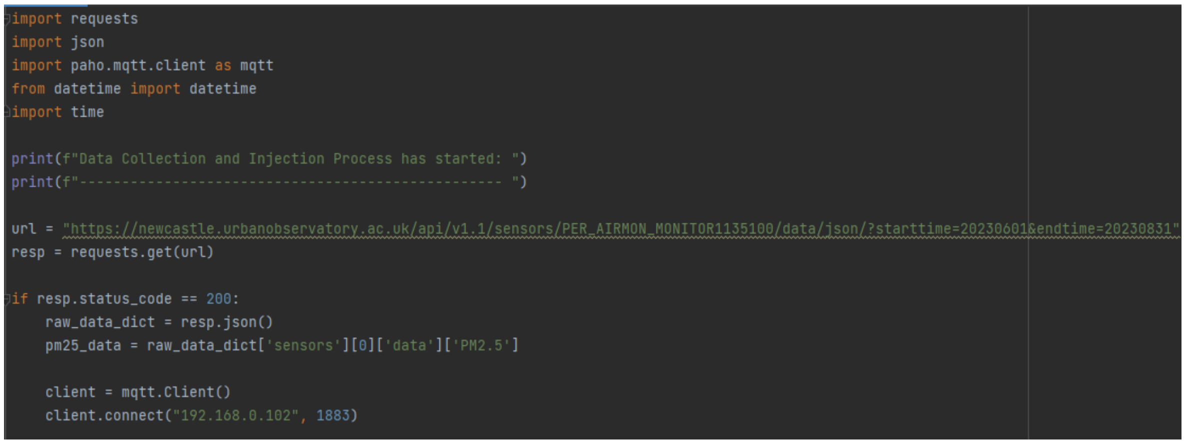
2.1 How to start IoT project

First of all, I started reading the whole project description and wrote down each main points for myself. As it is a quite detailed project, I used "divide and conquer" principle. I divide each task into 3 groups and defined what should I do to get to the next stage. After that I started to researching the points that I do not now deeply. As a start, I started researching the required python libraries and for what they are responsible for. For the next stage of research I tried to get overall understanding about the MQTT, RabbitMQ and of course Docker technologies, especially why and how I should use them. After finishing the initial research there was a big picture of what and why I should do to accomplish the final output. I fetch and did initial filtration in Azure(Windows) machine. After analysing the required output, it was time for the establishing MQTT channel in Azure(Edge) side and transferring data to there. As a last part, I made pre-processing and transferring data into Azure(Cloude) side using the RabbitMQ, At the end and after around 30 hours of research, debugging and re-writing code blocks for a good piece of work, I was successfully able to get the final output as it is intended. It was a brief explanation of my first IoT project. I the next stages, more deep detailed information about technical points are provided.

2.2 Data Collection and Injection Process

In this stage, all works are mainly done in Azure(Windows) machine, and it is the initial start of the project. Below, you can obviously get detailed information about how to get data from provided API, initially filter it and process of injection required data to the other Linux VM using the MQTT technology.

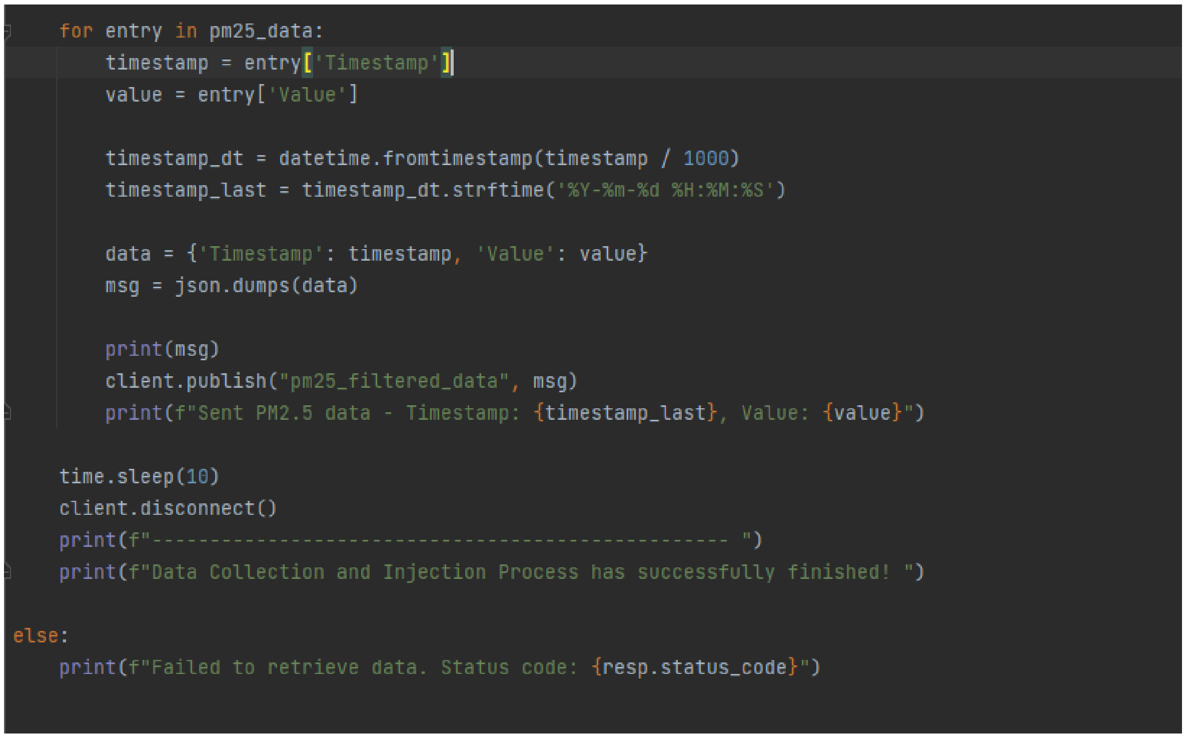
2.2.1 Initialization and Data Collection



```
import requests
import json
import paho.mqtt.client as mqtt
from datetime import datetime
import time

print(f"Data Collection and Injection Process has started: ")
print(f"------------------------------------------------ ")

url = "https://newcastle.urbanobservatory.ac.uk/api/v1.1/sensors/PER_AIRMON_MONITOR1135100/data/json/?starttime=20230601&endtime=20230831"
resp = requests.get(url)

if resp.status_code == 200:
    raw_data_dict = resp.json()
    pm25_data = raw_data_dict['sensors'][0]['data']['PM2.5']

    client = mqtt.Client()
    client.connect("192.168.0.102", 1883)
```

The first section of the script lays the foundations for a data gathering procedure. It provides itself with tools for HTTP communication and MQTT protocol operations by importing key libraries such as ***requests*** and ***paho.mqtt.client***. The start of the data gathering procedure is marked by print statements to ensure user awareness. Following that, a targeted API request is sent to the Newcastle Urban Observatory. This request is for PM2.5 air quality data, which is an important environmental parameter. If the request is successful and getting data out of Json format, we take only the PM2.5 values which is the main component for our project. The provided IP address is the IP address of our Azure(Edge) machine with the default access port of MQTT channel. Of course we will also run our MQTT channel in the Docker environment before running our code, which will be explained in the next stage of the report. Following a successful retrieval, the script uses MQTT to establish a connection, laying the groundwork for further data distribution.

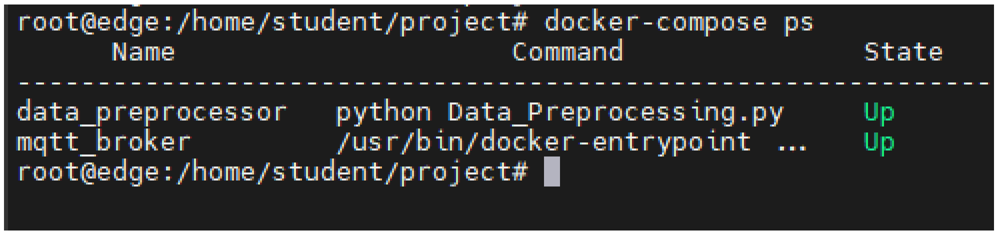
2.2.2 Initial Formatting and Data Transmission



```python
    for entry in pm25_data:
        timestamp = entry['Timestamp']
        value = entry['Value']

        timestamp_dt = datetime.fromtimestamp(timestamp / 1000)
        timestamp_last = timestamp_dt.strftime('%Y-%m-%d %H:%M:%S')

        data = {'Timestamp': timestamp, 'Value': value}
        msg = json.dumps(data)

        print(msg)
        client.publish("pm25_filtered_data", msg)
        print(f"Sent PM2.5 data - Timestamp: {timestamp_last}, Value: {value}")

    time.sleep(10)
    client.disconnect()
    print(f"-------------------------------------------------- ")
    print(f"Data Collection and Injection Process has successfully finished! ")

else:
    print(f"Failed to retrieve data. Status code: {resp.status_code}")
```

The second section moves from startup to action, delving into data refining and distribution. The script extracts timestamps and PM2.5 values painstakingly, reformulating the timestamp into an understandable format. This attention to detail demonstrates the script's dedication to data clarity. After processing, the data is converted into JSON, a widely used data transfer standard. The script's usage of MQTT, a strong communication protocol, is the highlight. It broadcasts the selected info to a certain subject, guaranteeing that subscribers may access it. Furthermore, the script's organised intervals and concise error-handling procedures highlight the script's balance of efficiency and dependability. The main points which worth to highlight include the processing and transferring data one-by-one is first out of two of them. The main point doing that is not to overload the communication channel and ensuring the atomicity of the data transfer. As a result of initial observation, we can clearly see that the data we are sending is not small. The other personal added approach to the default syntax was the **time.sleep(10)** line. It gives time to the subscriber side to get all the provided data before the publisher's disconnection.

2.2.3 Docker Side of the Azure(Edge) Machine



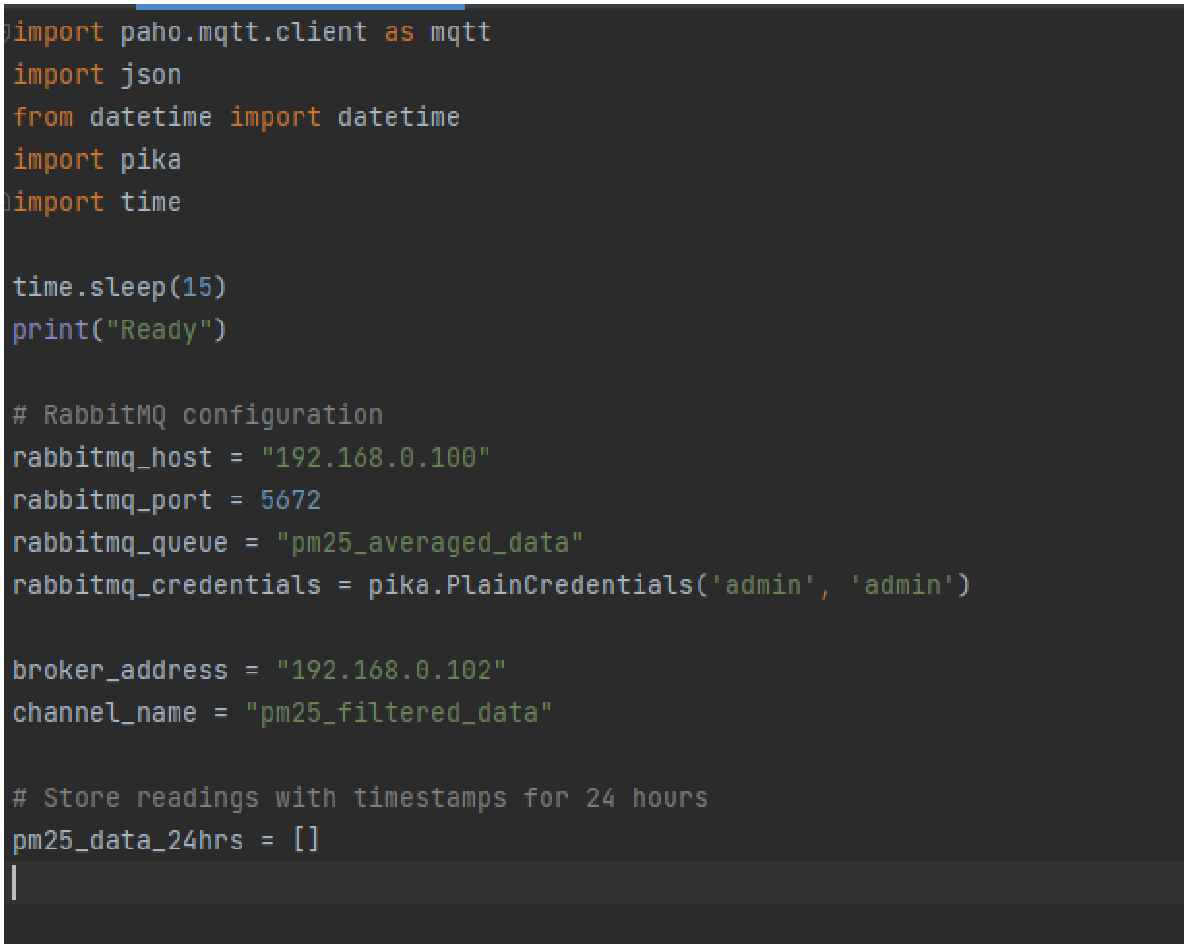
```
root@edge:/home/student/project# docker-compose ps
      Name                   Command                State
----------------------------------------------------------
data_preprocessor   python Data_Preprocessing.py    Up
mqtt_broker         /usr/bin/docker-entrypoint ...  Up
root@edge:/home/student/project# 
```

Although, this process is done in the Azure(Edge) side we should be sure that MQTT server is up and we can send our data through pre-defined communication channel. From the provided image, we can see that both ***mqtt-broker*** and ***data-preprocessor*** containers are up and running. System is waiting for its turn to accept, pre-process and transfer data to the next stage.

2.3 Data Pre-Processing and RabbitMQ Transfer

In this part of the project, we assume that our data receiving, injection processes and data transfer channel establishment are successful. Now, we will proceed in the pre-processing "pipeline" process, in which we will get published data, re-filter it, process it for the final stage and transfer data to RabbitMQ server. All these steps are done in Azure(Edge) machine which has 4GB Ram capacity and Ubuntu OS on it.

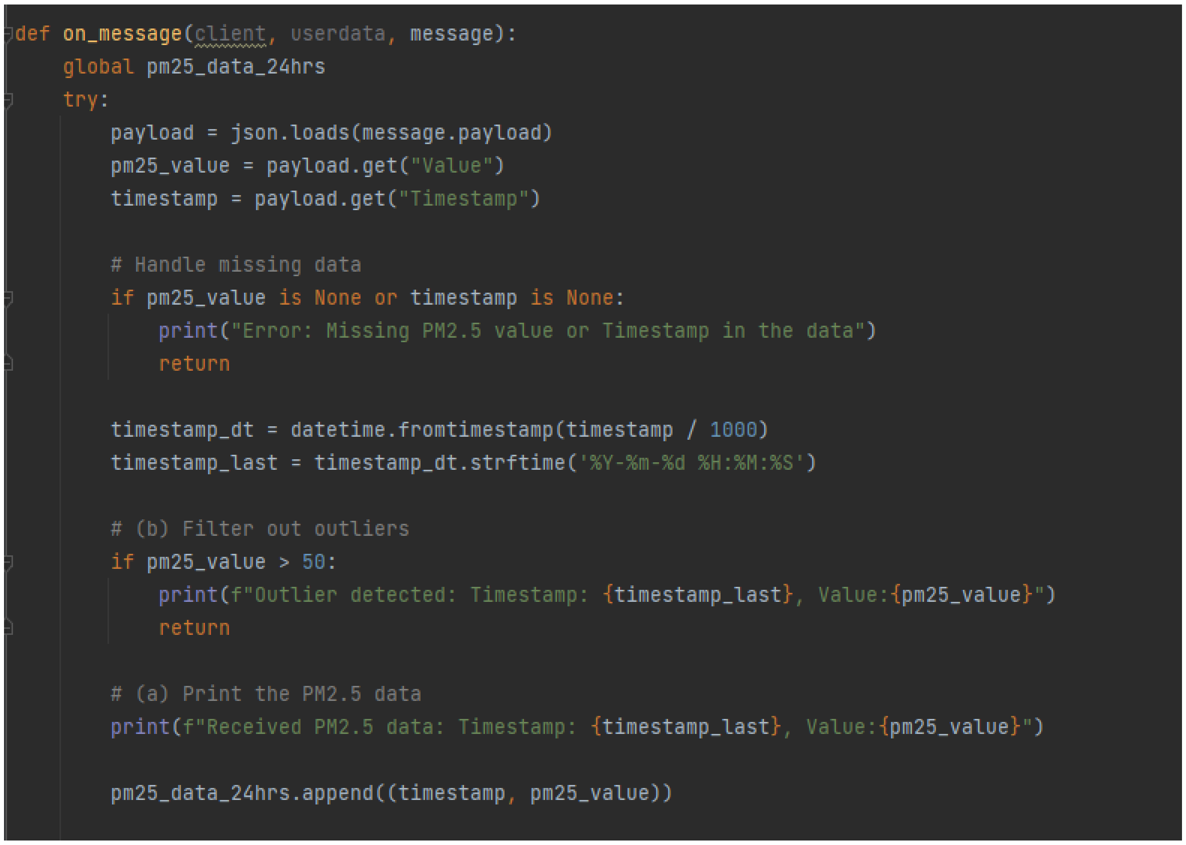
2.3.1 System Initialization and Configuration



```python
import paho.mqtt.client as mqtt
import json
from datetime import datetime
import pika
import time

time.sleep(15)
print("Ready")

# RabbitMQ configuration
rabbitmq_host = "192.168.0.100"
rabbitmq_port = 5672
rabbitmq_queue = "pm25_averaged_data"
rabbitmq_credentials = pika.PlainCredentials('admin', 'admin')

broker_address = "192.168.0.102"
channel_name = "pm25_filtered_data"

# Store readings with timestamps for 24 hours
pm25_data_24hrs = []
```

This section begins by loading the necessary libraries for MQTT communication, JSON parsing, and time operations. A preliminary sleep function adds an initial delay to the system, followed by a suggestive print statement. Initial delay is a specific approach to delay pre-processing process till the time that our MQTT server comes up and ready to function. Otherwise even if our ***data-preprocessor*** container is dependant from ***mqtt_broker*** container, we still have to wait to service to be up. Otherwise ***Data_Preprocessing.py*** exits without any error with code zero. RabbitMQ has detailed configuration parameters that describe the host, port, queue, and credentials. Even if default credentials are supplied while first start-up, I prefer to create a specific administrative user for this project. The provided ***rabbitmq_host*** is the IP address of our third VM which is Azure(Cloud). The broker's location and the chosen channel name are also supplied. The establishment of a list (***pm25_data_24hrs***) that is positioned to preserve PM2.5 measurements with their corresponding timestamps for a length of 24 hours is a critical component, establishing the framework for continuous data manipulation logic.

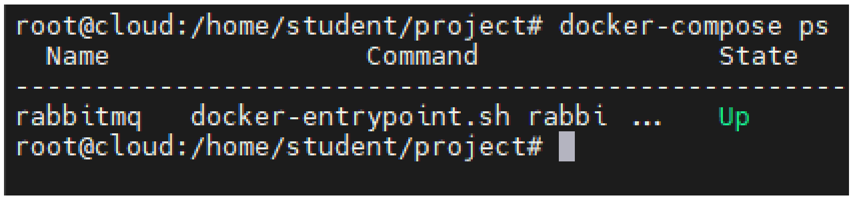
2.3.2 Message Reception and Data Management



```python
def on_message(client, userdata, message):
    global pm25_data_24hrs
    try:
        payload = json.loads(message.payload)
        pm25_value = payload.get("Value")
        timestamp = payload.get("Timestamp")

        # Handle missing data
        if pm25_value is None or timestamp is None:
            print("Error: Missing PM2.5 value or Timestamp in the data")
            return

        timestamp_dt = datetime.fromtimestamp(timestamp / 1000)
        timestamp_last = timestamp_dt.strftime('%Y-%m-%d %H:%M:%S')

        # (b) Filter out outliers
        if pm25_value > 50:
            print(f"Outlier detected: Timestamp: {timestamp_last}, Value:{pm25_value}")
            return

        # (a) Print the PM2.5 data
        print(f"Received PM2.5 data: Timestamp: {timestamp_last}, Value:{pm25_value}")

        pm25_data_24hrs.append((timestamp, pm25_value))
```

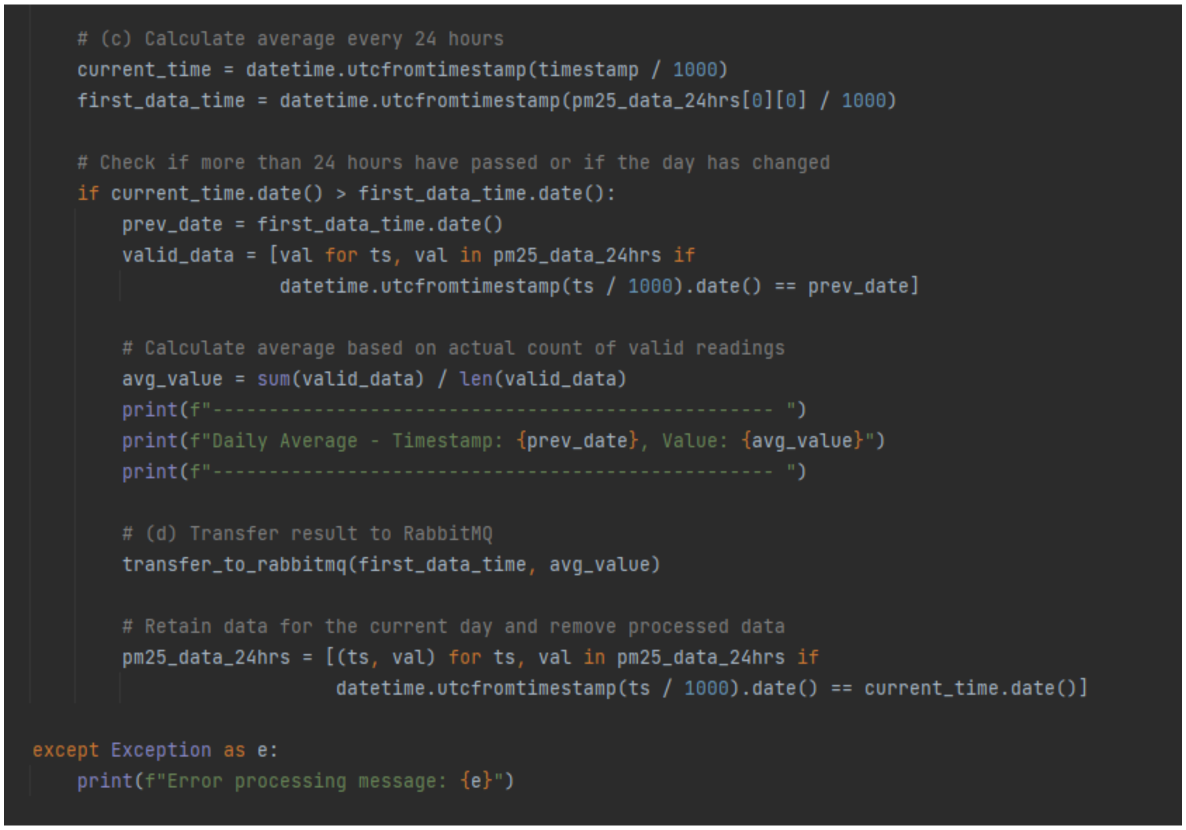
When fresh MQTT messages are received, the function ***on_message*** functions as an event handler. When called, it attempts to parse the incoming message and retrieve the **PM2.5** value and timestamp. An error message is produced if either piece of data is missing. Outliers in **PM2.5** values are discovered and highlighted, and timestamps are converted to a more visible format. Valid data, which extracts the outliers out of the concept, is subsequently shown for the user's viewing pleasure for being also monitored from docker log. Finally, the received data point, which includes the date and PM2.5 measurement, is added to the previously initialised 24-hour data list, strengthening the system's data reservoir. Adding process is the start of the daily average calculation logic is explained in the next section.

2.3.3 Daily Data Averaging and Transfer



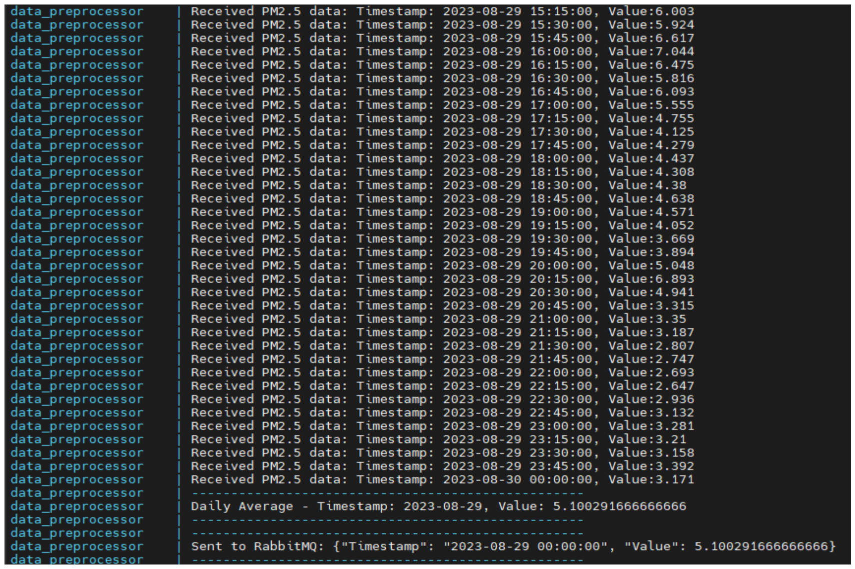
```
root@cloud:/home/student/project# docker-compose ps
  Name                Command              State
--------------------------------------------------
rabbitmq   docker-entrypoint.sh rabbi ...   Up
root@cloud:/home/student/project# 
```

Before going to the next stages, we have to be sure that our **RabbitMQ** server is up and ready to accept requests. In the provided image it is clearly showed that, the required RabbitMQ server is up in our Azure(Cloud) server inside our Docker environment by name of ***rabbitmq*** container. Now, we can proceed to the pre-processing and data transmission stage in our Azure(Edge) machine.



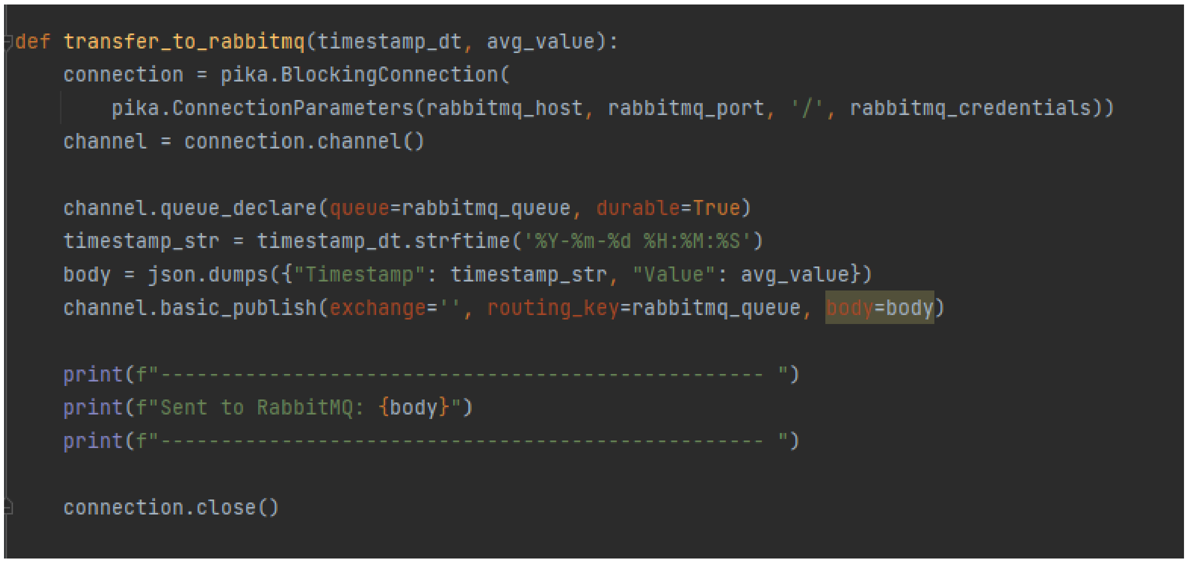
```python
        # (c) Calculate average every 24 hours
        current_time = datetime.utcfromtimestamp(timestamp / 1000)
        first_data_time = datetime.utcfromtimestamp(pm25_data_24hrs[0][0] / 1000)

        # Check if more than 24 hours have passed or if the day has changed
        if current_time.date() > first_data_time.date():
            prev_date = first_data_time.date()
            valid_data = [val for ts, val in pm25_data_24hrs if
                          datetime.utcfromtimestamp(ts / 1000).date() == prev_date]

            # Calculate average based on actual count of valid readings
            avg_value = sum(valid_data) / len(valid_data)
            print(f"-------------------------------------------------- ")
            print(f"Daily Average - Timestamp: {prev_date}, Value: {avg_value}")
            print(f"-------------------------------------------------- ")

            # (d) Transfer result to RabbitMQ
            transfer_to_rabbitmq(first_data_time, avg_value)

            # Retain data for the current day and remove processed data
            pm25_data_24hrs = [(ts, val) for ts, val in pm25_data_24hrs if
                               datetime.utcfromtimestamp(ts / 1000).date() == current_time.date()]

    except Exception as e:
        print(f"Error processing message: {e}")
```

The primary focus here is on data processing and transmission at 24-hour intervals as daily average. The current timestamp is first computed and compared to the earliest recorded timestamp. If a day has passed, data from the previous 24 hours is isolated. A critical procedure follows, in which an average **PM2.5** value for the day is determined using accurate data. This average is shown before being sent to **RabbitMQ** via the earlier detailed method. Following that, processed data is deleted to ensure that only the ***most recent*** day's data is maintained. It should be also stated that, the time variable which sent as an input to the function that responsible for data transmission to RabbitMQ is not in the raw timestamp format but rather in more understandable format for not to make repletion on conversion procedure in the cloud side also. Also, exception handling is expertly designed, preventing unexpected failures during message processing. A successful data transfer can be also monitored from docker logs. As an example, the peace of successful data transmission log output is provided in the below:



```
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 15:15:00, Value:6.003
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 15:30:00, Value:5.924
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 15:45:00, Value:6.617
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 16:00:00, Value:7.044
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 16:15:00, Value:6.475
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 16:30:00, Value:5.816
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 16:45:00, Value:6.093
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 17:00:00, Value:5.555
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 17:15:00, Value:4.755
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 17:30:00, Value:4.125
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 17:45:00, Value:4.279
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 18:00:00, Value:4.437
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 18:15:00, Value:4.308
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 18:30:00, Value:4.38
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 18:45:00, Value:4.638
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 19:00:00, Value:4.571
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 19:15:00, Value:4.052
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 19:30:00, Value:3.669
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 19:45:00, Value:3.894
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 20:00:00, Value:5.048
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 20:15:00, Value:6.893
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 20:30:00, Value:4.941
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 20:45:00, Value:3.315
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 21:00:00, Value:3.35
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 21:15:00, Value:3.187
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 21:30:00, Value:2.807
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 21:45:00, Value:2.747
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 22:00:00, Value:2.693
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 22:15:00, Value:2.647
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 22:30:00, Value:2.936
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 22:45:00, Value:3.132
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 23:00:00, Value:3.281
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 23:15:00, Value:3.21
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 23:30:00, Value:3.158
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-29 23:45:00, Value:3.392
data_preprocessor     | Received PM2.5 data: Timestamp: 2023-08-30 00:00:00, Value:3.171
data_preprocessor     | --------------------------------------------------
data_preprocessor     | Daily Average - Timestamp: 2023-08-29, Value: 5.100291666666666
data_preprocessor     | --------------------------------------------------
data_preprocessor     | --------------------------------------------------
data_preprocessor     | Sent to RabbitMQ: {"Timestamp": "2023-08-29 00:00:00", "Value": 5.100291666666666}
data_preprocessor     | --------------------------------------------------
```

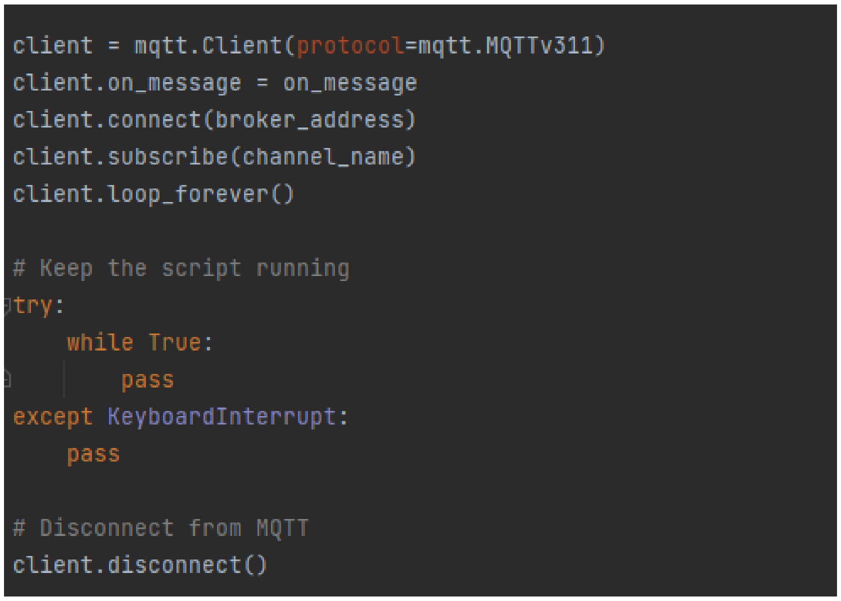
2.3.4 Data Transmission to RabbitMQ



```python
def transfer_to_rabbitmq(timestamp_dt, avg_value):
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(rabbitmq_host, rabbitmq_port, '/', rabbitmq_credentials))
    channel = connection.channel()

    channel.queue_declare(queue=rabbitmq_queue, durable=True)
    timestamp_str = timestamp_dt.strftime('%Y-%m-%d %H:%M:%S')
    body = json.dumps({"Timestamp": timestamp_str, "Value": avg_value})
    channel.basic_publish(exchange='', routing_key=rabbitmq_queue, body=body)

    print(f"-------------------------------------------------- ")
    print(f"Sent to RabbitMQ: {body}")
    print(f"-------------------------------------------------- ")

    connection.close()
```

The ***transfer_to_rabbitmq*** function is responsible for transmitting data to a RabbitMQ server. The **pika** library simplifies connecting to the server, and parameters such as the host, port, and credentials are used to do this. A queue is defined after specifying the connection channel to ensure data **integrity** and **durability**. Importantly, timestamps are transformed to a human-readable format before being coupled with a data value to generate the sent message. A brief confirmation message is produced after successful transmission to the RabbitMQ queue, and the connection is properly terminated, assuring system robustness.

2.3.5 MQTT Client Initialization and Continuous Monitoring



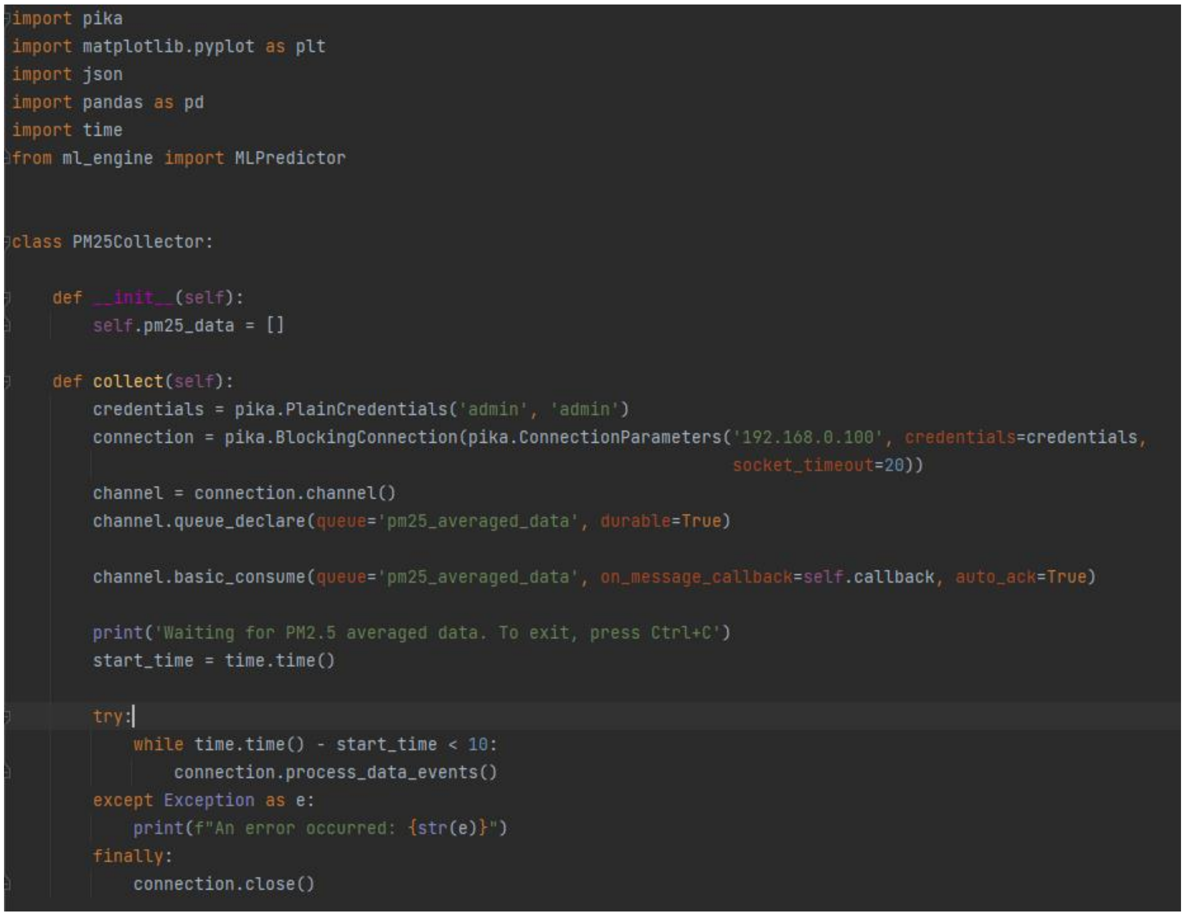
```python
client = mqtt.Client(protocol=mqtt.MQTTv311)
client.on_message = on_message
client.connect(broker_address)
client.subscribe(channel_name)
client.loop_forever()

# Keep the script running
try:
    while True:
        pass
except KeyboardInterrupt:
    pass

# Disconnect from MQTT
client.disconnect()
```

In this part of the code, The user initialises a **MQTT** client using the MQTTv311 protocol in the given code section. The reason of protocol specification was based on documentation content. The message-handling function of the client is set to **on_message**. Following that, the client connects to the specified **broker_address** and subscribes to a certain **channel_name.** The MQTT client then enters a state of eternal listening, waiting for incoming messages. A loop continues forever unless deliberately halted by a **KeyboardInterrupt** (e.g., pressing Ctrl+C) to ensure the script's ongoing running. In the event of such an interruption, the client gracefully disconnects from the MQTT broker. This configuration guarantees continuous **data monitoring and processing**.

2.4 Data Visualization and Prediction

This is the last stage of our IoT project which is responsible for fetching data from RabbitMQ server and make data visualization as well as the data prediction after it. This section is processed in the Azure(Cloud) machine which has 4GB of Ram and the Ubuntu OS on it. Program itself is run in the Ubuntu machine manually and the expected outputs are displayed on the screen. At the end of the report you will be able to see the results with the personal analytical discussions about them. Now it will be better to go step-by-step for the last of part the project. Also it should be also worth to be stated that, the Clean Code approach also tried to be implemented in this peace of program, which is led by using the Abstraction principle, use of abstract classes.

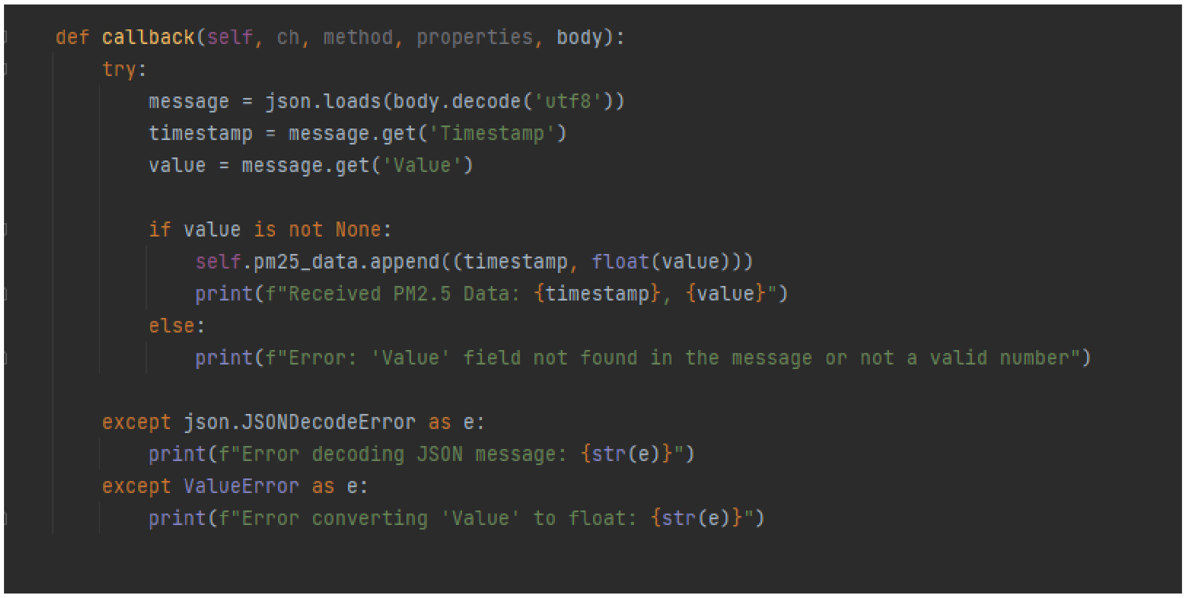
2.4.1 Data Fetching from RabbitMQ



```
import pika
import matplotlib.pyplot as plt
import json
import pandas as pd
import time
from ml_engine import MLPredictor


class PM25Collector:

    def __init__(self):
        self.pm25_data = []

    def collect(self):
        credentials = pika.PlainCredentials('admin', 'admin')
        connection = pika.BlockingConnection(pika.ConnectionParameters('192.168.0.100', credentials=credentials,
                                                                       socket_timeout=20))
        channel = connection.channel()
        channel.queue_declare(queue='pm25_averaged_data', durable=True)

        channel.basic_consume(queue='pm25_averaged_data', on_message_callback=self.callback, auto_ack=True)

        print('Waiting for PM2.5 averaged data. To exit, press Ctrl+C')
        start_time = time.time()

        try:
            while time.time() - start_time < 10:
                connection.process_data_events()
        except Exception as e:
            print(f"An error occurred: {str(e)}")
        finally:
            connection.close()
```

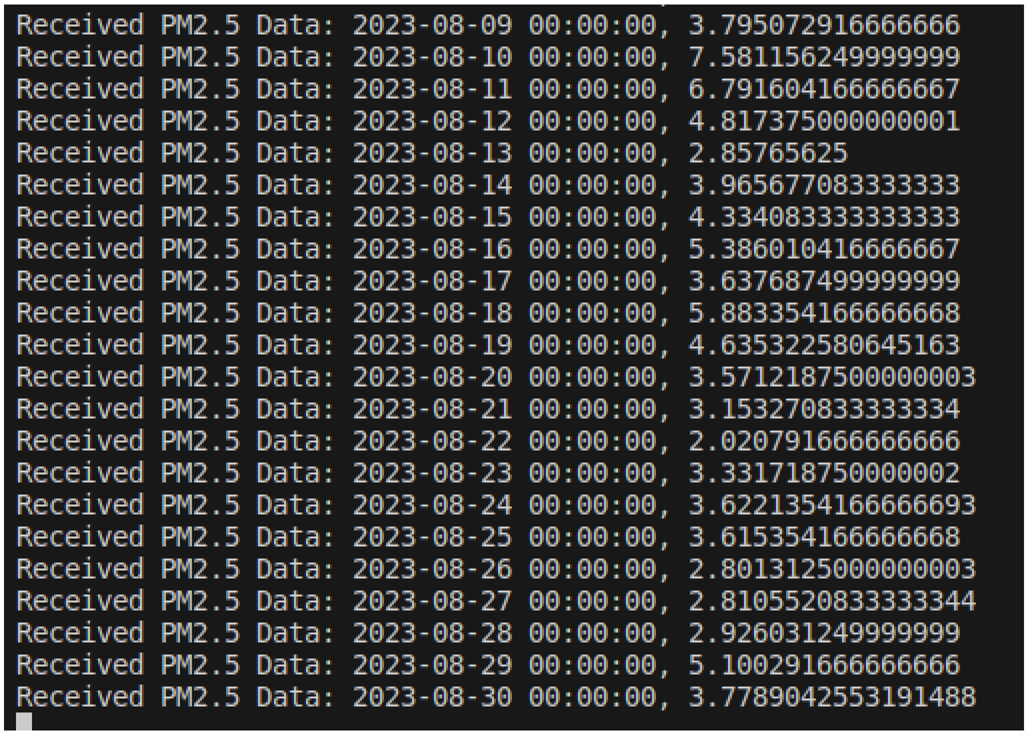
In the first section of the code, we establish the foundations for data gathering and processing from RabbitMQ. The **PM25Collector** class is a critical component. Its first role is to build an empty data store as a specialised collector for PM2.5 particle data. Once activated, the collect method takes precedence. It sets credentials, connects to the **Message Queue** server, and generates the required channels to begin data receiving by implementing the **pika** library. This configuration offers a **stable and safe** framework for handling incoming data, emphasising its value in real-time monitoring of environmental factors such as PM2.5. We also give 10 seconds of time frame for the system to fetch all the before stepping into next part of the execution. Otherwise, in some cases if lags happen in fetching time, big problems occur for the next stage and as a result non-expected output comes out.

2.4.2 Data Reception and Processing



```
def callback(self, ch, method, properties, body):
    try:
        message = json.loads(body.decode('utf8'))
        timestamp = message.get('Timestamp')
        value = message.get('Value')

        if value is not None:
            self.pm25_data.append((timestamp, float(value)))
            print(f"Received PM2.5 Data: {timestamp}, {value}")
        else:
            print(f"Error: 'Value' field not found in the message or not a valid number")

    except json.JSONDecodeError as e:
        print(f"Error decoding JSON message: {str(e)}")
    except ValueError as e:
        print(f"Error converting 'Value' to float: {str(e)}")
```

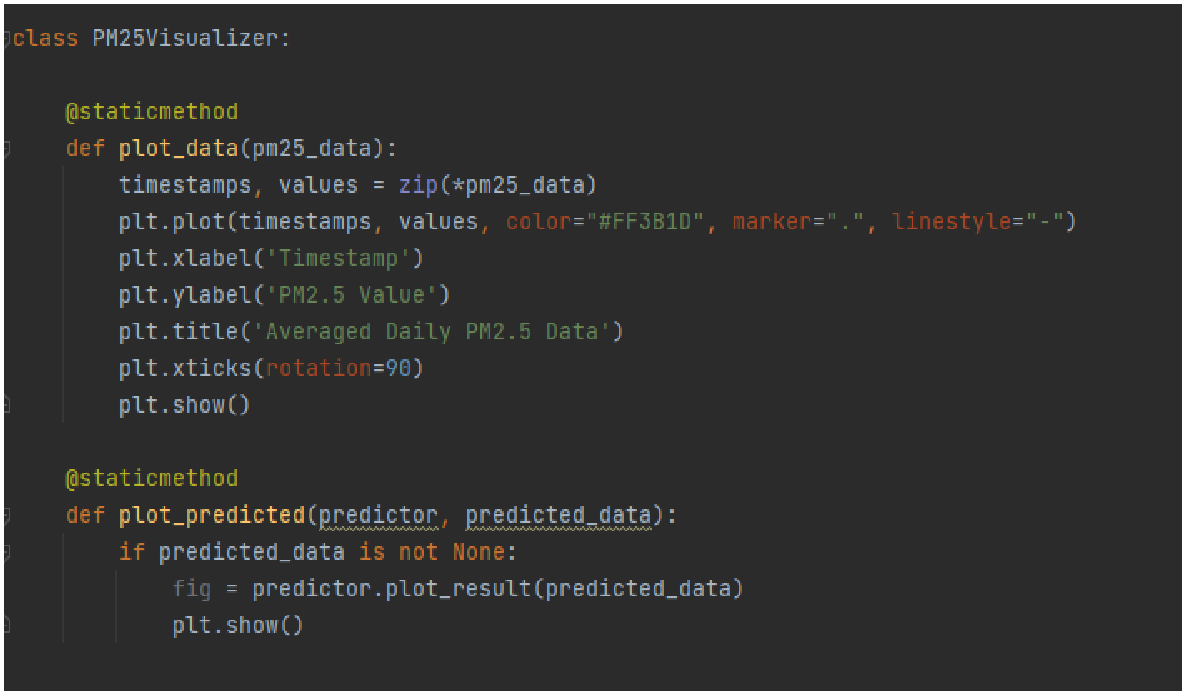
Within the PM25Collector class, the callback function acts as the backbone for data receiving and preliminary processing. Each incoming communication is decoded in order to obtain its content. The date and the PM2.5 value, for example, are exactly extracted and compiled. This approach not only verifies that the received data is consistent with the defined format, but it also oversees its inclusion in the PM2.5 data list. Furthermore, this section is thoughtfully constructed to address any abnormalities. The error-handling systems provide seamless and uninterrupted data processing, from JSON decoding problems to datatype conversion misalignments. As a confirmation, system prints the received data into the console.



```
Received PM2.5 Data: 2023-08-09 00:00:00, 3.795072916666666
Received PM2.5 Data: 2023-08-10 00:00:00, 7.581156249999999
Received PM2.5 Data: 2023-08-11 00:00:00, 6.791604166666667
Received PM2.5 Data: 2023-08-12 00:00:00, 4.817375000000001
Received PM2.5 Data: 2023-08-13 00:00:00, 2.85765625
Received PM2.5 Data: 2023-08-14 00:00:00, 3.965677083333333
Received PM2.5 Data: 2023-08-15 00:00:00, 4.334083333333333
Received PM2.5 Data: 2023-08-16 00:00:00, 5.386010416666667
Received PM2.5 Data: 2023-08-17 00:00:00, 3.637687499999999
Received PM2.5 Data: 2023-08-18 00:00:00, 5.883354166666668
Received PM2.5 Data: 2023-08-19 00:00:00, 4.635322580645163
Received PM2.5 Data: 2023-08-20 00:00:00, 3.5712187500000003
Received PM2.5 Data: 2023-08-21 00:00:00, 3.153270833333334
Received PM2.5 Data: 2023-08-22 00:00:00, 2.020791666666666
Received PM2.5 Data: 2023-08-23 00:00:00, 3.331718750000002
Received PM2.5 Data: 2023-08-24 00:00:00, 3.6221354166666693
Received PM2.5 Data: 2023-08-25 00:00:00, 3.615354166666668
Received PM2.5 Data: 2023-08-26 00:00:00, 2.8013125000000003
Received PM2.5 Data: 2023-08-27 00:00:00, 2.8105520833333344
Received PM2.5 Data: 2023-08-28 00:00:00, 2.926031249999999
Received PM2.5 Data: 2023-08-29 00:00:00, 5.100291666666666
Received PM2.5 Data: 2023-08-30 00:00:00, 3.7789042553191488
```

In the provided example, you can also see part of the sample of data fetching process of the active system in Azure(Cloud).

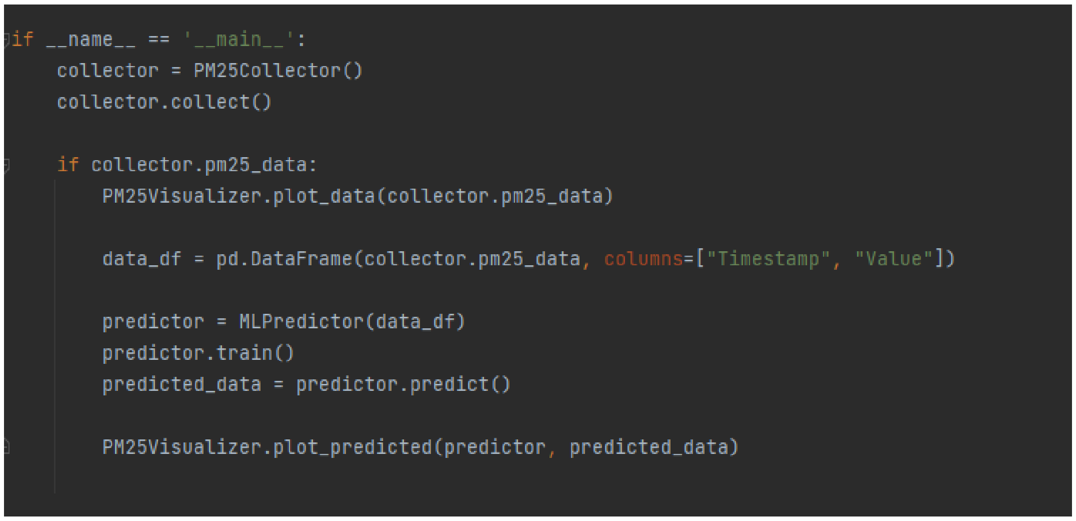
2.4.3 Data Visualization Methods



```
class PM25Visualizer:

    @staticmethod
    def plot_data(pm25_data):
        timestamps, values = zip(*pm25_data)
        plt.plot(timestamps, values, color="#FF3B1D", marker=".", linestyle="-")
        plt.xlabel('Timestamp')
        plt.ylabel('PM2.5 Value')
        plt.title('Averaged Daily PM2.5 Data')
        plt.xticks(rotation=90)
        plt.show()

    @staticmethod
    def plot_predicted(predictor, predicted_data):
        if predicted_data is not None:
            fig = predictor.plot_result(predicted_data)
            plt.show()
```

As we all know, visualisation is crucial in data comprehension, and the **PM25Visualizer** class perfectly meets this demand. It provides a full visualisation suite with two major techniques in our system, **plot_data** and **plot_predicted.** The former displays raw PM2.5 data, matching timestamps with their appropriate PM2.5 levels to create an easy-to-follow time-series graph. The latter, on the other hand, is concerned with displaying data obtained from predictive analysis. These approaches, which make use of the **matplotlib** toolkit, ensure that both raw and predicted data are depicted with clarity, precision, and visual appeal, appealing to both technical and non-technical audiences.

2.4.4 Execution and Data Flow



```
if __name__ == '__main__':
    collector = PM25Collector()
    collector.collect()

    if collector.pm25_data:
        PM25Visualizer.plot_data(collector.pm25_data)

        data_df = pd.DataFrame(collector.pm25_data, columns=["Timestamp", "Value"])

        predictor = MLPredictor(data_df)
        predictor.train()
        predicted_data = predictor.predict()

        PM25Visualizer.plot_predicted(predictor, predicted_data)
```

The main execution block is in charge of orchestrating the whole data processing operation. It starts with the **PM25Collector** and goes on a data collecting journey. Following data collection, a check confirms that the data repository is not empty. Once determined, raw data visualisation occurs, offering the initial insights. As the data enters the analytical phase, it is structured into a **DataFrame** format, preparing it for machine learning applications. Once **trained** on this structured data, a specialised model takes on the responsibility of making future predictions. The visualisation of these forecasts is the result of this approach, providing a comprehensive picture of both present and expected PM2.5 levels.

3 Analytical Discussion of Results and Conclusion

3.1 Averaged Daily PM2.5 Data Visualization Output

This graph elucidates the averaged daily PM2.5 values over a span of days, depicted by the horizontal axis. PM2.5 refers to atmospheric particulate matter that has a diameter of less than 2.5 micrometres, which is regarded for its health implications when inhaled, as it can easily penetrate deep into the lungs.

The red line chart offers an immediate visual impression of variability in the PM2.5 values. We observe a pronounced fluctuation, with certain days experiencing significantly elevated PM2.5 values, potentially surpassing the recommended safety thresholds. Notably, there are sharp peaks, suggesting days where the air quality could have been particularly poor, possibly due to specific events, such as industrial activities, vehicular emissions, or lack of wind dispersion. Conversely, there are valleys indicative of days with comparatively better air quality. The gaps in the data points might suggest missing data or days where measurements were not taken.

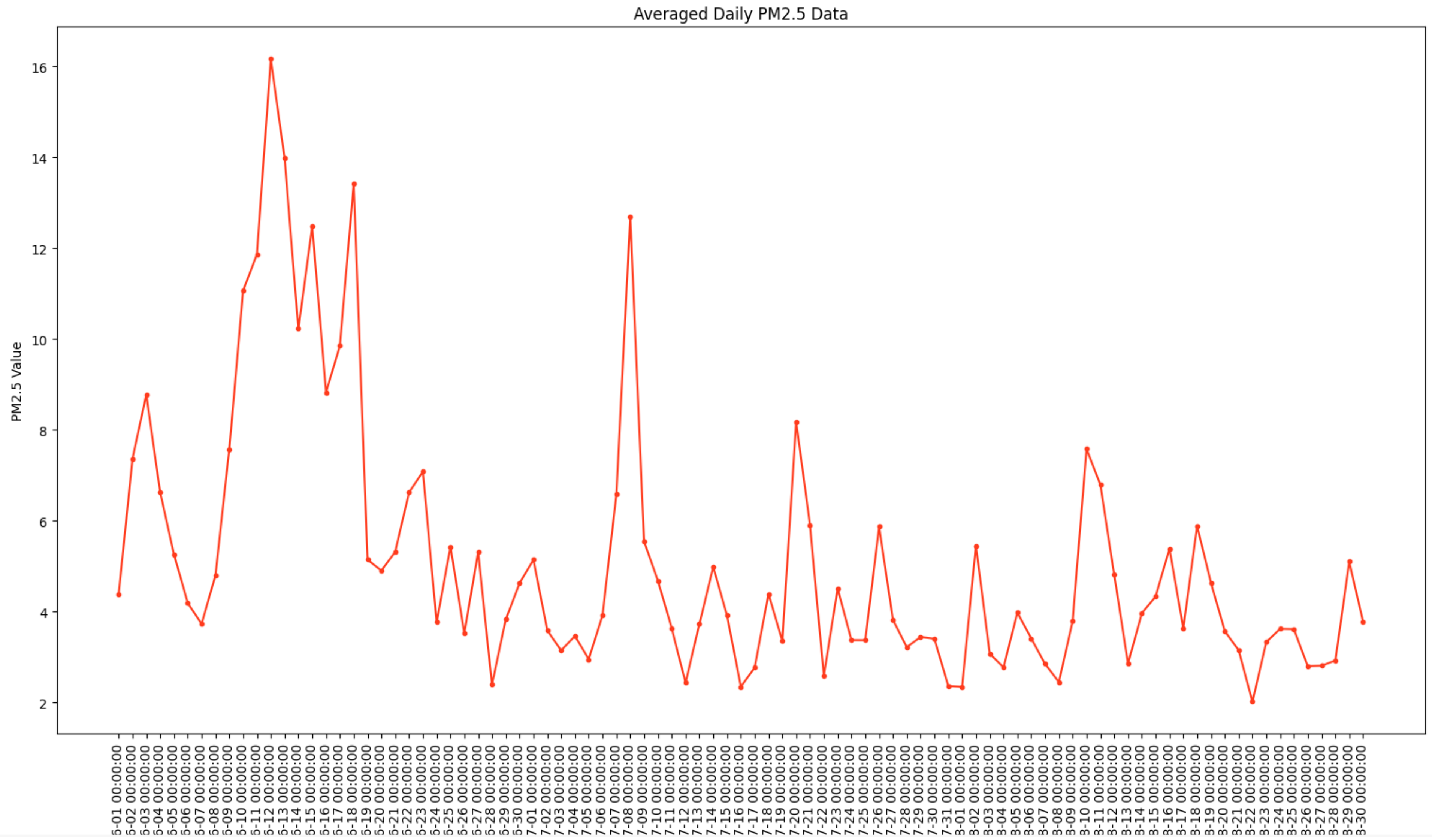
One essential takeaway from this visualization is the dynamic nature of PM2.5 concentrations. It emphasizes the importance of continuous monitoring and mitigation strategies, especially during those peak days, to ensure public health and safety.

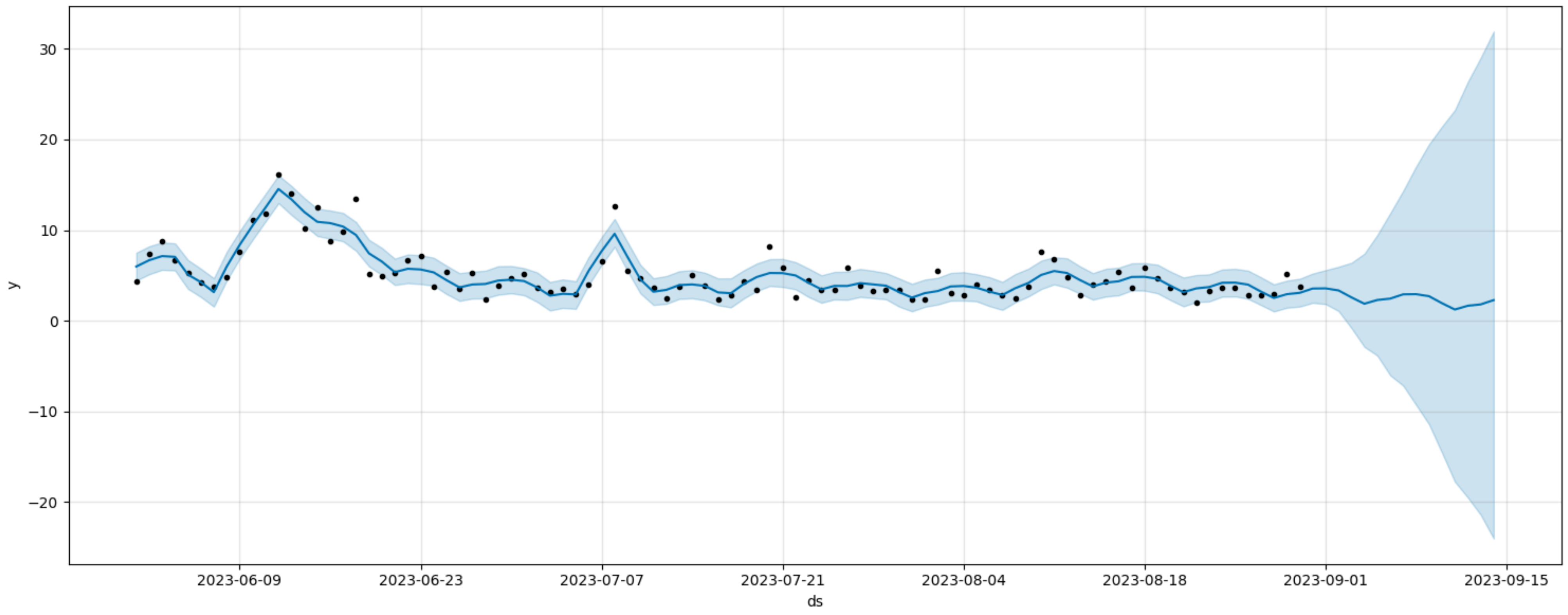
3.2 Time Series Forecasting with Confidence Interval

The second chart provides a time series forecast of a metric (denoted as 'y') across specific dates in 2023. This type of visualization is widely utilized in predictive analytics to project future values based on historical data. The dark blue line denotes the observed values of 'y'. These data points appear to be exhibiting a relatively stable pattern with some periodic peaks and troughs, indicative of cyclical behaviours or seasonal influences.

The light blue shaded area represents the confidence interval, providing a range within which future values of 'y' are likely to fall. As we move further to the right, this interval widens, suggesting increasing uncertainty in the predictions as we project farther into the future. This is a common characteristic in time series forecasting, where near-term predictions are often more accurate than long-term ones. Towards the extreme right, the solid blue line transitions into a dotted one. This demarcates the boundary between observed and predicted values. The predictions appear to follow the trend of the historical data, but the widening confidence interval underscores the need for caution in interpreting these forecasts.

This visualization is invaluable for decision-makers, allowing them to anticipate future trends and make informed decisions, all the while understanding the inherent uncertainties associated with such predictions.


Averaged Daily PM2.5 Data
PM2.5 Value
16
14
12
10
8
6
4
2
6-01 00:00:00
6-02 00:00:00
6-03 00:00:00
6-04 00:00:00
6-05 00:00:00
6-06 00:00:00
6-07 00:00:00
6-08 00:00:00
6-09 00:00:00
6-10 00:00:00
6-11 00:00:00
6-12 00:00:00
6-13 00:00:00
6-14 00:00:00
6-15 00:00:00
6-16 00:00:00
6-17 00:00:00
6-18 00:00:00
6-19 00:00:00
6-20 00:00:00
6-21 00:00:00
6-22 00:00:00
6-23 00:00:00
6-24 00:00:00
6-25 00:00:00
6-26 00:00:00
6-27 00:00:00
6-28 00:00:00
6-29 00:00:00
6-30 00:00:00
7-01 00:00:00
7-02 00:00:00
7-03 00:00:00
7-04 00:00:00
7-05 00:00:00
7-06 00:00:00
7-07 00:00:00
7-08 00:00:00
7-09 00:00:00
7-10 00:00:00
7-11 00:00:00
7-12 00:00:00
7-13 00:00:00
7-14 00:00:00
7-15 00:00:00
7-16 00:00:00
7-17 00:00:00
7-18 00:00:00
7-19 00:00:00
7-20 00:00:00
7-21 00:00:00
7-22 00:00:00
7-23 00:00:00
7-24 00:00:00
7-25 00:00:00
7-26 00:00:00
7-27 00:00:00
7-28 00:00:00
7-29 00:00:00
7-30 00:00:00
7-31 00:00:00
8-01 00:00:00
8-02 00:00:00
8-03 00:00:00
8-04 00:00:00
8-05 00:00:00
8-06 00:00:00
8-07 00:00:00
8-08 00:00:00
8-09 00:00:00
8-10 00:00:00
8-11 00:00:00
8-12 00:00:00
8-13 00:00:00
8-14 00:00:00
8-15 00:00:00
8-16 00:00:00
8-17 00:00:00
8-18 00:00:00
8-19 00:00:00
8-20 00:00:00
8-21 00:00:00
8-22 00:00:00
8-23 00:00:00
8-24 00:00:00
8-25 00:00:00
8-26 00:00:00
8-27 00:00:00
8-28 00:00:00
8-29 00:00:00
8-30 00:00:00



30
20
10
0
−10
−20
y
2023-06-09
2023-06-23
2023-07-07
2023-07-21
2023-08-04
2023-08-18
2023-09-01
2023-09-15
ds

3.3 Conclusion

As a conclusion, I want to state that all above explanations are made in the assumption that all required dependencies are installed or pulled properly. Un-showed parts of the project (for example docker-compose.yml and Dockerfile) are same as it stated in the official documentations and no analytical approach are done there. Final results of the system are same as it intended to be. The system answers all the requirements in the way that it was provided in the assessment documentation.