Internet of Things (IOT) Project

Introduction

This project demonstrates utilizing the data collected from IOT sensors and use it further to complete following tasks. These tasks include collection of data, preprocessing that data, visualizing the data and passing this data to machine learning model for data prediction of next 15 days. The data flows between three layers, namely IOT layer, edge layer and cloud layer. IOT layer comprises of all the sensors and its data is stored in Newcastle Urban Observatory. The edge layer comprises of two tasks wherein task 1 we need to collect data from urban observatory and publish that data to task 2 via EMQX docker image and in task 2, we collect the data published by task 1 and do preprocessing on that data and send it to cloud layer by using RabbitMQ docker image. Now once the data reaches to cloud layer i.e task 3, we use this data for visualization and prediction using the provided machine learning model.

Below figure depicts the overview diagram.

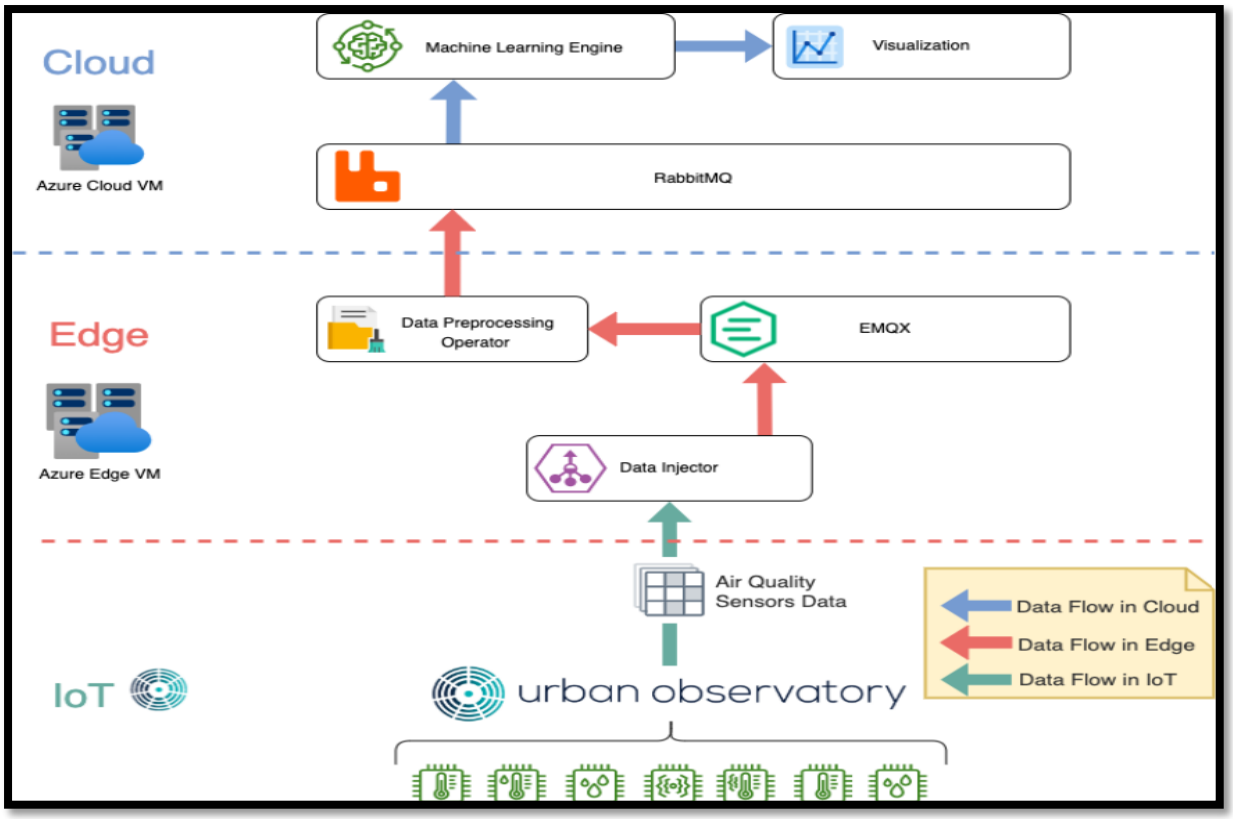

Figure 1: **Overview Diagram**

Before proceeding with the tasks, we need to turn on both the ubuntu virtual machines, one for edge and another one for cloud. Then connect to those VMs and install *net-tools* to the system by running the following command,

sudo apt install net-tools – This package contains important tools for controlling network subsystem.

Then we can run "*ifconfig*" command to check IP addresses of both VMs. Now we need to manually install python3, pip, and docker-compose tool in both ubuntu VMs.

Task 1: Designing a data injector component

In task 1, first of all we need to install “requests” and “paho-mqtt” dependencies in edge virtual machine using the below mentioned commands,

pip install requests
pip install paho-mqtt

requests is a simple HTTP library that allows us to perform HTTP based communications.

paho-mqtt enables applications to connect to an MQTT broker and publish message and subscribe the message from topic.

After installing required dependencies, we need to pull and run EMQX docker image on edge virtual machine by executing below command on terminal, here we map port number 1883 of host with port 1883 of container.

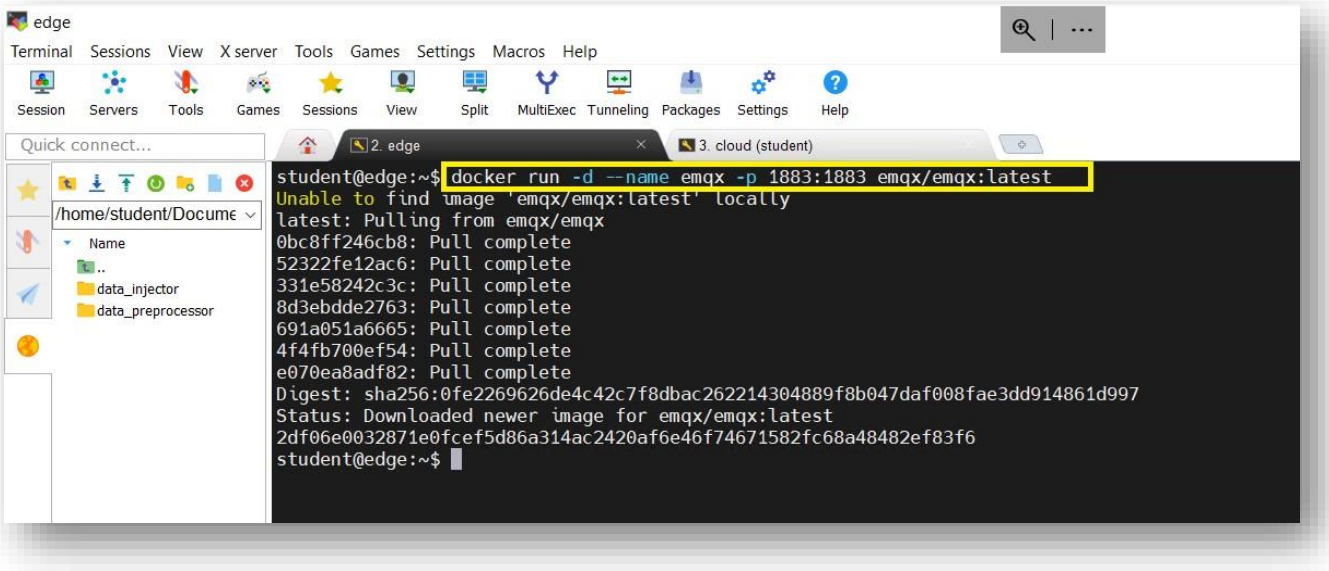

Figure 2: **Pulling and running EMQX docker image on edge virtual machine**

EMQX docker image is an open source MQTT broker and is used to publish and subscribe messages, to and from topic.

Now we need to write a python code for data injector component.

Code description:

Firstly, we are importing all the required libraries such as json, time, requests and paho mqtt client. Then we create mqtt client object and connect to the mqtt service. We provide IP address of edge virtual machine and port number 1883 to which EMQX image is mapped while connecting to mqtt service. Proceeding further, we have to call Newcastle Urban Observatory API and collect the raw air quality sensors data and convert it from json response to dictionary format. This raw data consists of many metrics such as CO_2, NO_2, NO, PM2.5, PM10 and so on, which we print on the console. However, we only need the data related to PM2.5, so we fetch only PM2.5 data and store it in a list. This PM2.5

data has parameters like Sensor Name, Units, Variable, Timestamp, Value and Flagged as Suspect Reading. Out of these parameters we need only timestamp and value field, as a result we are filtering out other parameters and storing and printing only timestamp and value data of PM2.5. Now we will publish this data on topic which will be further subscribed by task 2.

Code snippet for data injector component:

datainjector.py

```python
import json
import time
import requests

from paho.mqtt import client as mqtt_client

mqtt_ip = "192.168.0.102"
mqtt_port = 1883
topic = "CSC8112"

# Create a mqtt client object
client = mqtt_client.Client()

# Callback function for MQTT connection
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT OK!")
    else:
        print("Failed to connect, return code %d\n", rc)

# Connect to MQTT service
client.on_connect = on_connect
client.connect(mqtt_ip, mqtt_port)

# Publish message to MQTT
# Storing urban observatory url in a variable
link = "http://uoweb3.ncl.ac.uk/api/v1.1/sensors/PER_AIRMON_MONITOR1135100/data/json/?starttime=20230601&endtime=20230831"

# Getting data by calling API
getdata = requests.get(link)

# Converting the response data to dictionary
convert = getdata.json()

# Printing raw data
print(convert)

# Fetching only PM2.5 data and filtering rest and storing in a variable result of type list
result = convert["sensors"][0]["data"]["PM2.5"]

# Keep only Timestamp and Value data, remove remaining data
for i in result:
    del i['Sensor Name']
    del i['Units']
    del i['Variable']
    del i['Flagged as Suspect Reading']

print(result)

client.publish(topic, json.dumps(result)) # json.dumps(result) converts it to string and publishes
time.sleep(2)
```

Figure 3: **Task 1 python code**

Output of Task 1 – datainjector component

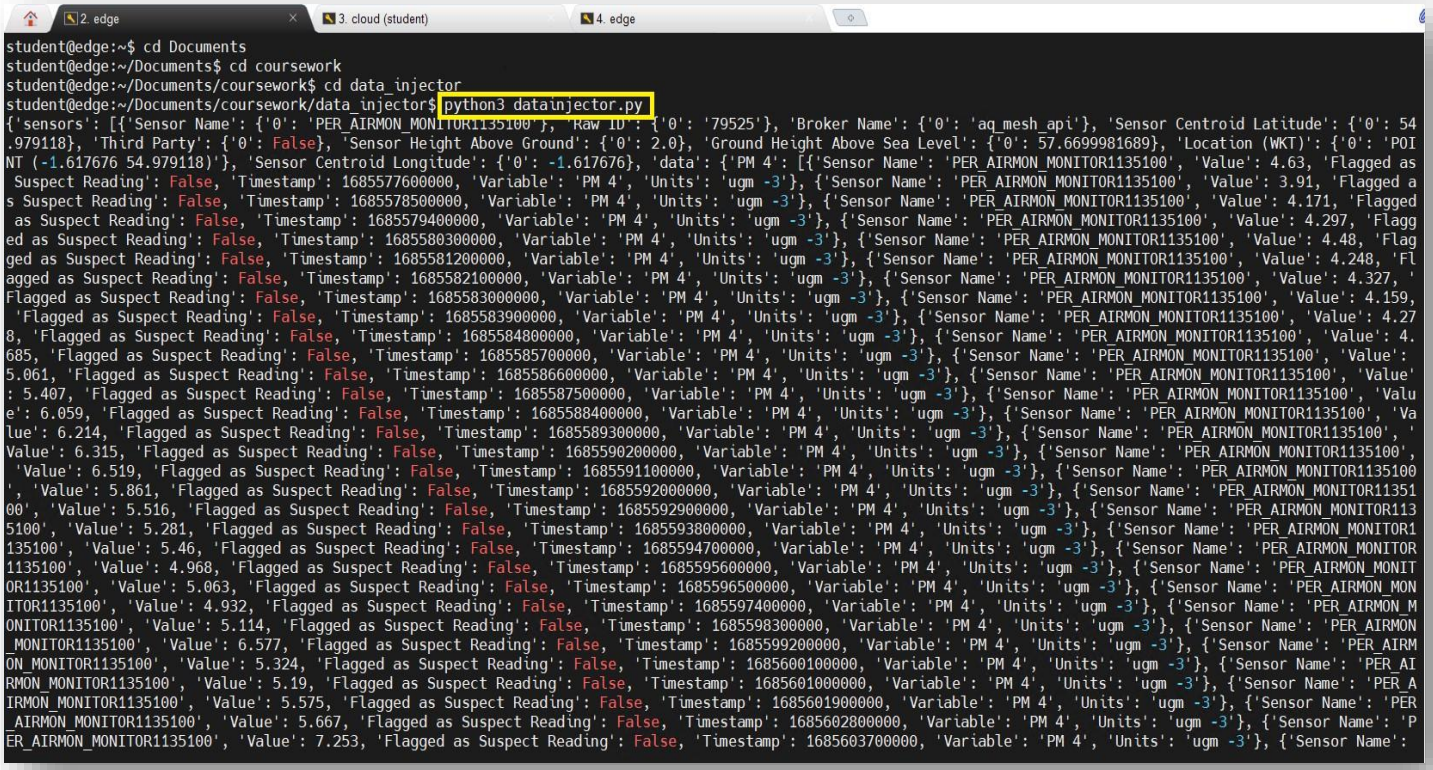

Figure 4: **Printing raw air quality sensors data collected from urban observatory**

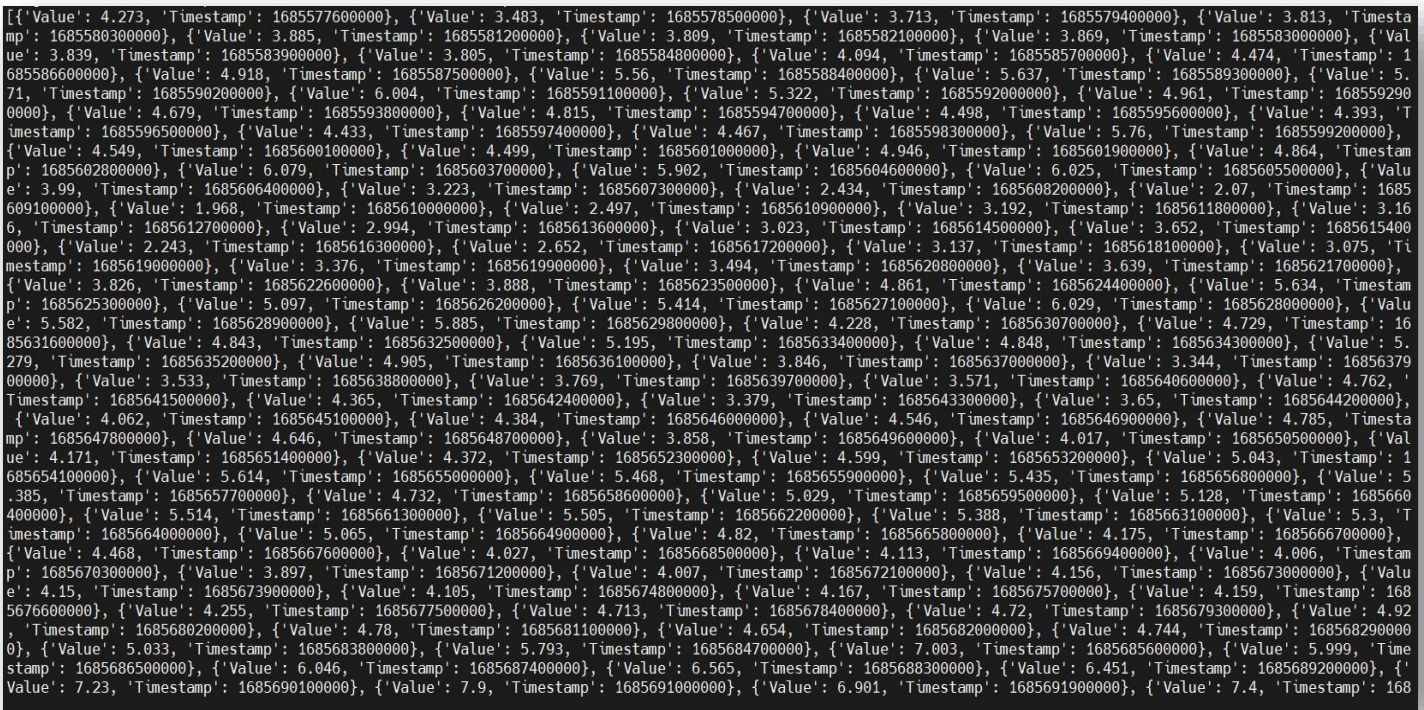

Figure 5: **Printing the metrics timestamp and value of PM2.5 data**

Task 2: Data preprocessing operator

For task 2 we need “paho-mqtt” and “pika” dependencies to be installed in edge virtual machine but as “paho-mqtt” has already been installed in task 1, we will just install pika by executing following command,

pip install pika

pika is a RabbitMQ client library with implementation of Advance Message Queuing Protocol (AMQP).

We have already downloaded EMQX docker image on edge virtual machine in task 1 which will be used to subscribe data published by task 1. As in task 2, after preprocessing, we have to send that data to cloud layer (task 3) with the help of RabbitMQ docker image, we need to pull and run that image on edge virtual machine.

We execute below command on edge VMs terminal and map the image on port 5672.

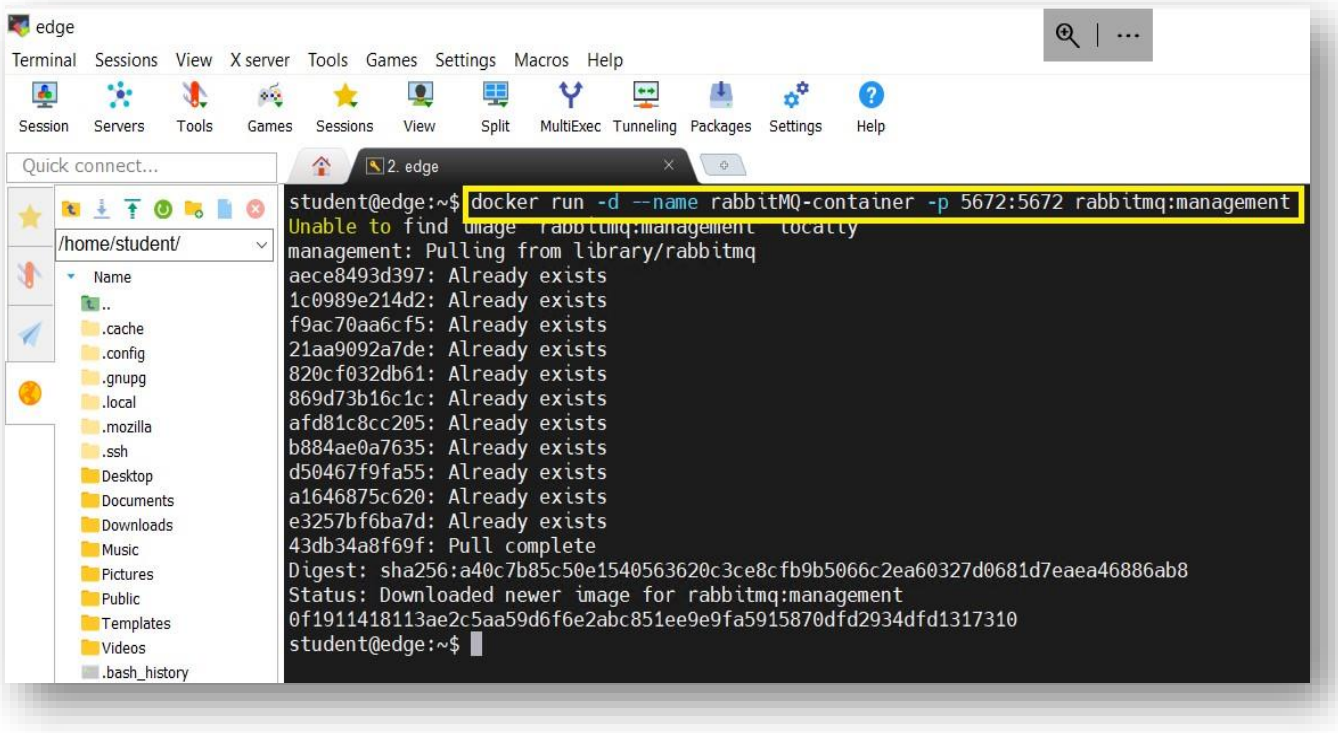

Figure 6**: Pulling and running RabbitMQ docker image on edge virtual machine**

RabbitMQ image is an open source message broker software which implements AMQP protocol and will be used for data communication/flow between edge layer and cloud layer.

Code Description

Import all the required libraries such as json, pika, datetime and paho-mqtt. Create the mqtt client object and connect to mqtt service in order to get the data from the topic which is published by task 1. In data preprocessing our first task is to filter out the outliers and print them on the screen. Outliers are the data points for which value is greater than 50. After removing all the outliers from the data, next task is to calculate the daily average value. In order to get average value of a particular day, I have made use of dictionaries. One dictionary will be storing the date and total of all the values for that particular date. Another dictionary to store the date and total number of values on that date and finally third dictionary will store the date and average value. Next, we are storing this data in a list with

timestamp and average value which will be sent by RabbitMQ producer to cloud (task 3). As we are sending this data to cloud layer, we have specified IP address of cloud virtual machine and 5672 port number which is mapped with RabbitMQ image while connecting to RabbitMQ service.

Code snippet for data preprocessor operator

datapreprocessor.py

```
import json
import pika
from datetime import datetime
from paho.mqtt import client as mqtt_client

if __name__ == '__main__':
    mqtt_ip = "192.168.0.102"

    mqtt_port = 1883
    topic = "CSC8112"

    # Create a mqtt client object
    client = mqtt_client.Client()

    # Callback function for MQTT connection
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT OK!", flush = True)
        else:
            print("Failed to connect, return code %d\n", rc, flush = True)


    # Connect to MQTT service
    client.on_connect = on_connect
    client.connect(mqtt_ip, mqtt_port)


    # Callback function will be triggered
    def on_message(client, userdata, msg):
        print(f"Printing PM2.5 data collected by emqx subscriber: {json.loads(msg.payload)}", flush = True) #type of json.loads(msg.payload): list
        emqxcollecteddata = json.loads(msg.payload)

        # Checking outliers and printing them on console
        outliers = []
        for d in emqxcollecteddata:
          if d['Value'] >= 50:
            outliers.append(d)
        print("**********************************************************************************************************************************************", flush = True)
        print("outliers :", outliers, flush = True)
        print("**********************************************************************************************************************************************", flush = True)

        # Filtering out outliers
        outliersremoved = [] # Data without outliers
        for d1 in emqxcollecteddata:
          if d1['Value'] < 50:
            outliersremoved.append(d1)


        # Getting average values logic below
        a = {}
        b = {}
        c = {}
        preprocessdata = []
        cnt = 2
        cnt1 = 1
        for entry in outliersremoved:
          timestamp = entry['Timestamp']
          value = entry['Value']
          date = datetime.fromtimestamp(timestamp / 1000).strftime('%Y-%m-%d')
          #print(date)
          if len(a) == 0:
            a[date] = value
          else:
            if date in a:
              a[date] = a[date] + value
            else:
              a[date] = value



        for entry1 in outliersremoved:
          timestamp = entry1['Timestamp']
          value = entry1['Value']
          date = datetime.fromtimestamp(timestamp / 1000).strftime('%Y-%m-%d')
          #print(date)
          if len(a) == 0:
            b[date] = 1
          else:
            if date in b:
              b[date] = b[date] + 1
            else:
              b[date] = 1

        #print(b)
```

```
        #print(b)

        for d in a:
          c[d] = a[d] / b[d]

        #print(c)

        for i in c:
          preprocessdata.append({'Timestamp': i, 'Value': c[i]})

        print("Printing daily average value data: ",preprocessdata, flush = True)

        rabbitmq_ip = "192.168.0.100"
        rabbitmq_port = 5672
        # Queue name
        rabbitmq_queque = "CSC8112test"
        # Connect to RabbitMQ service
        connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbitmq_ip, port=rabbitmq_port))
        channel = connection.channel()
        # Declare a queue
        channel.queue_declare(queue=rabbitmq_queque)
        # Produce message
        channel.basic_publish(exchange='',
                              routing_key=rabbitmq_queque,
                              body=json.dumps(preprocessdata))
        connection.close()









    # Subscribe MQTT topic
    client.subscribe(topic)
    client.on_message = on_message

    # Start a thread to monitor message from publisher
    client.loop_forever()

```

Figure 7: **Task 2 python code**

Another part in task 2 is to build a docker image of data preprocessing operator. For these we need to create three files namely docker file, yaml file and requirements text file.

Docker File

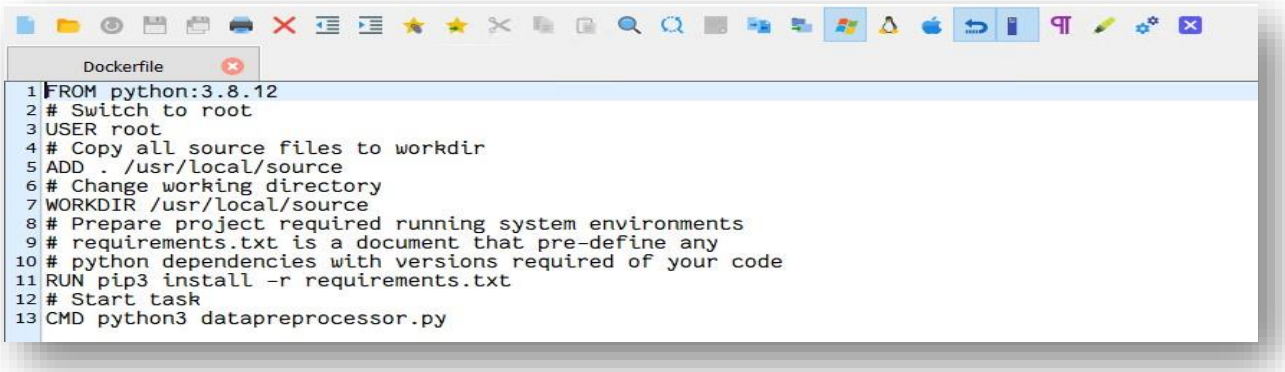

Figure 8: **Docker File**

Docker file is used to build an image of our data preprocessor (task 2) code. In this file we specify that the language used is python, then we switch to the root and copy all the files from the current directory to source directory and set it as a working directory. Next it installs all the required dependencies for running the code that are mentioned in requirements text file and finally the command to run our data preprocessor (task 2) code.

Docker-compose file

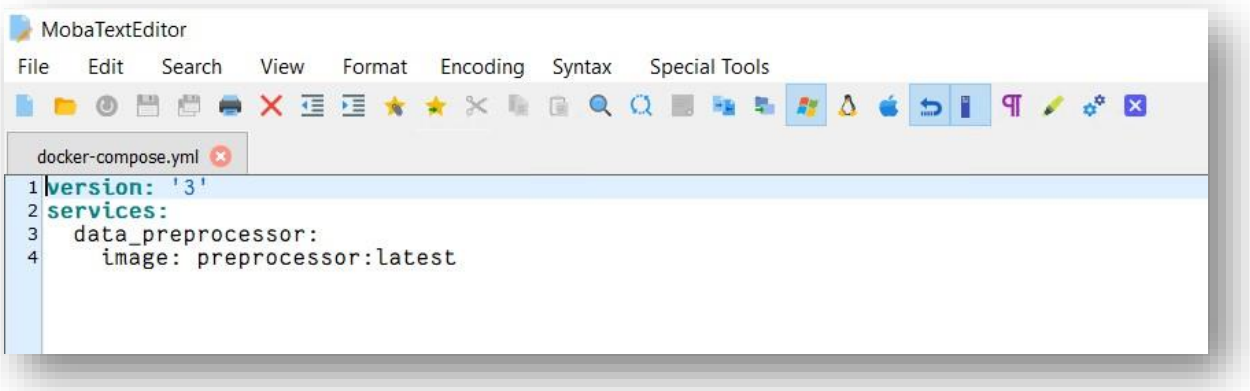

Figure 9: **docker-compose file**

Docker compose file contains the configuration in which we define the name and image of the service that we want to run as a part of our application.

Requirements text file

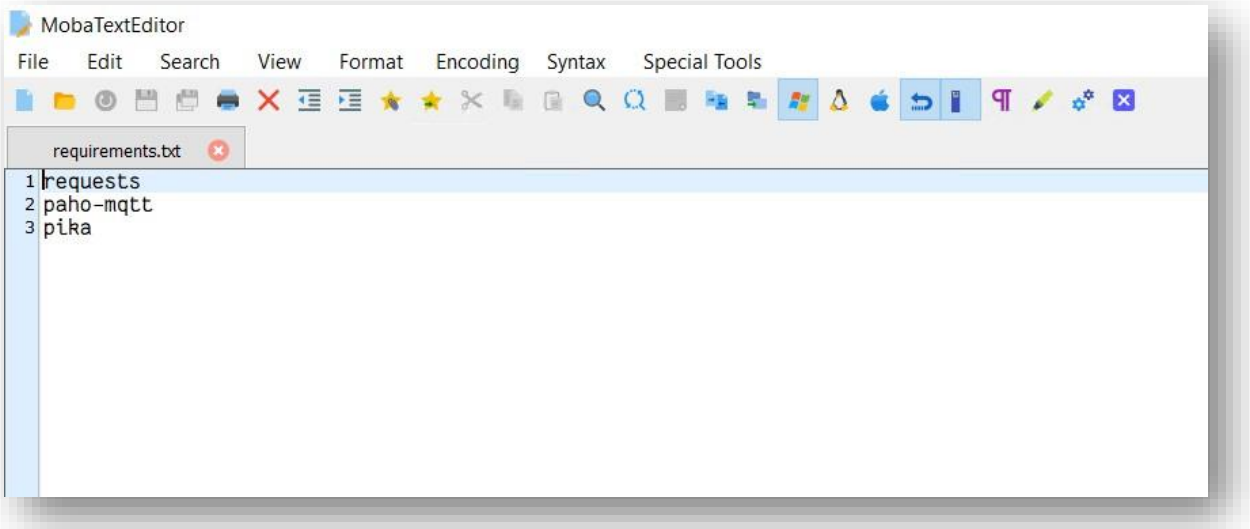

Figure 10: **requirements text file**

Requirements text file contains all the dependencies that we want to install as a part of running our application.

Now we build the image of our data preprocessor (task 2) operator.

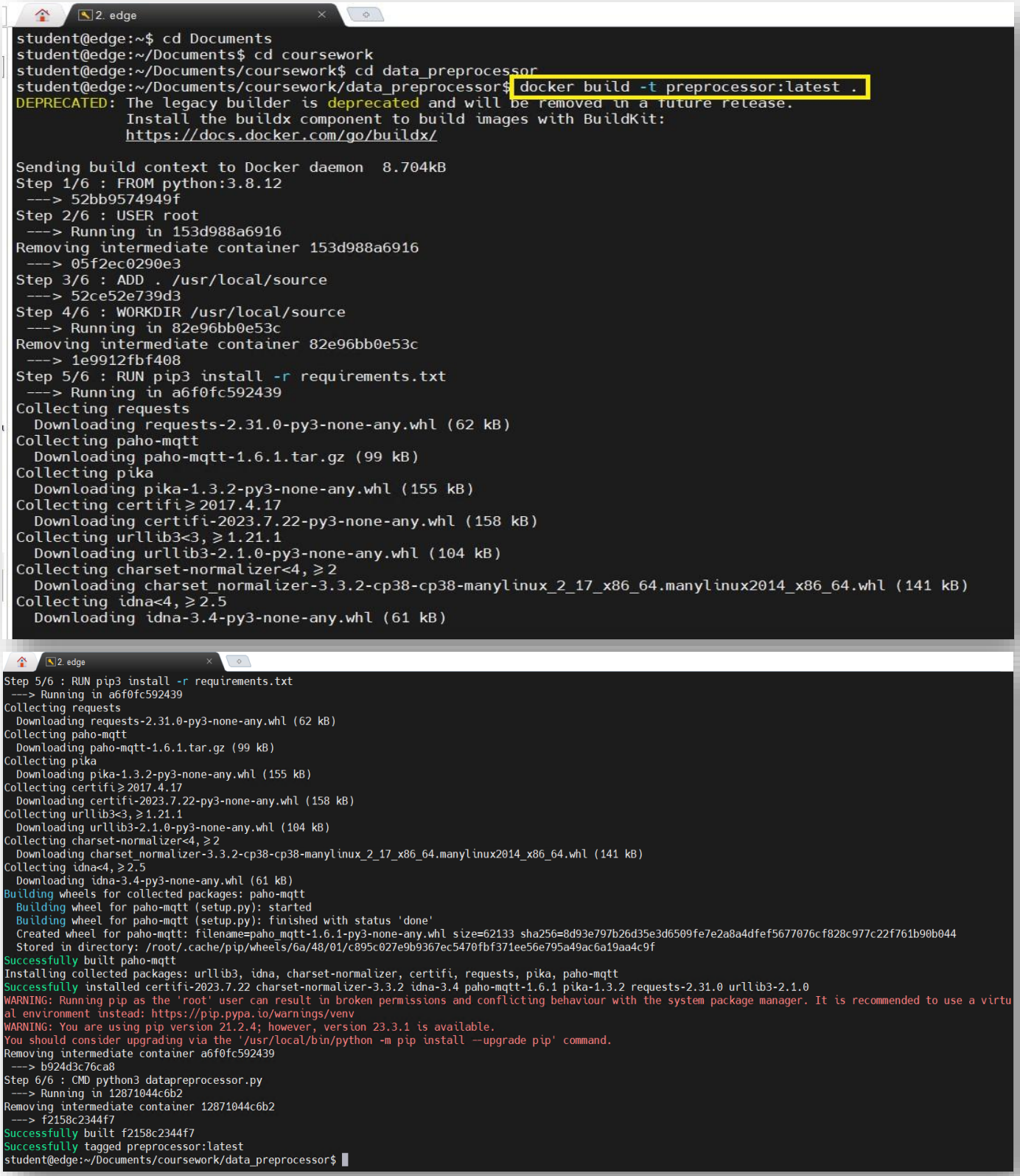

Figure 11: **Building an image of "data preprocessing operator" (task 2) python code**

Output of Task 2 – data preprocessing operator (Running as a docker image)

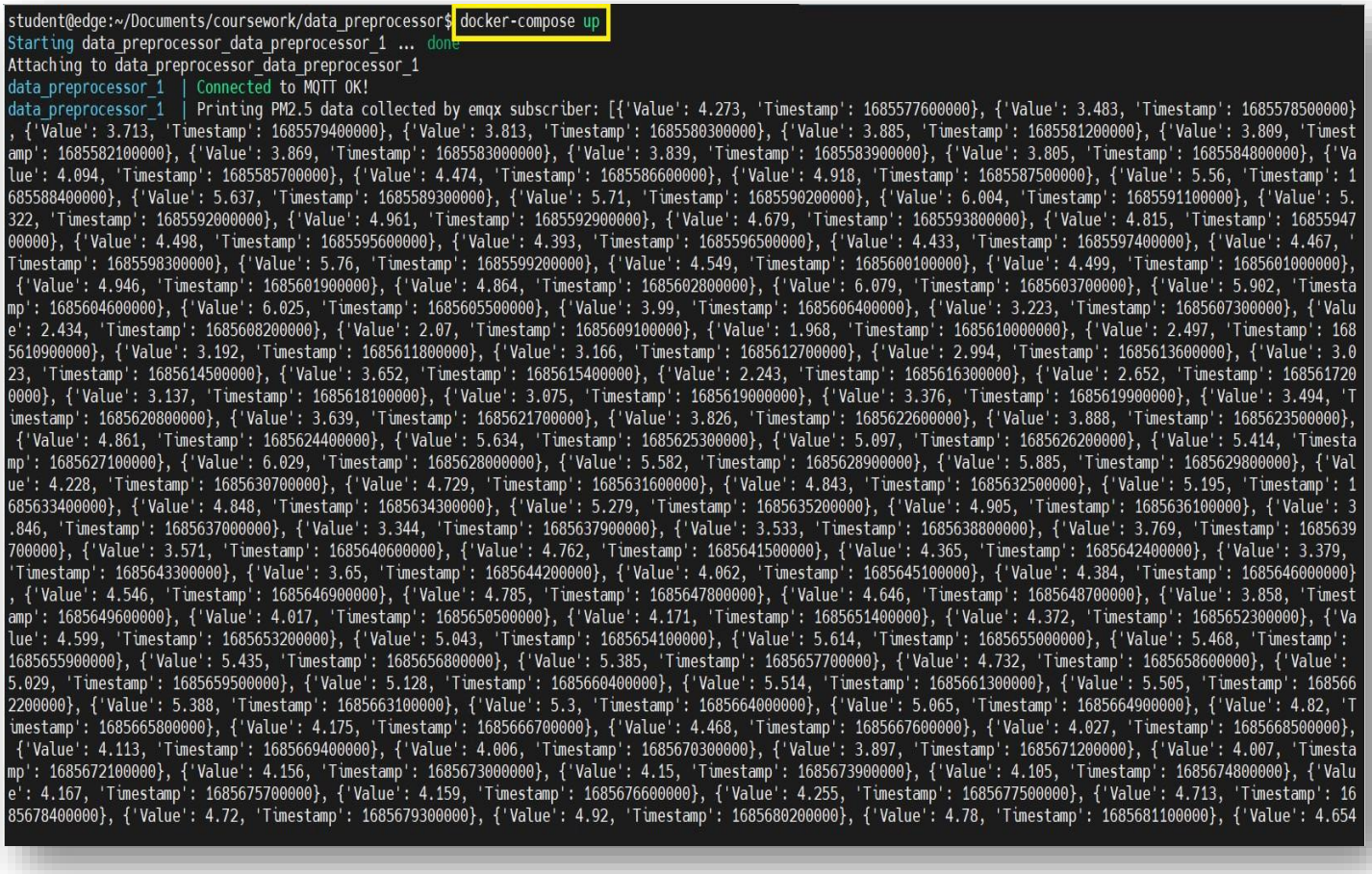

Figure 12: **Printing PM2.5 data collected by EMQX subscriber**

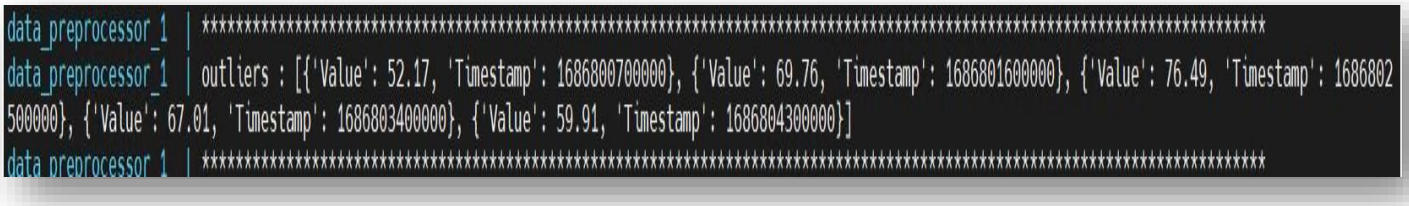

Figure 13: **Printing outliers data**

Figure 14: **Printing daily average value data**

Task 3: Data prediction and visualization

This task requires installation of dependencies such as pika, prophet, matplotlib, plotly and pandas on cloud virtual machine. Commands to install these dependencies are as below,

pip install pika – RabbitMQ client library for python.

pip install prophet – Python library which is used for forecasting timeseries based data.

pip install matplotlib – Python library used for visualization purpose.

pip install - - upgrade plotly – Python plotting library.

pip install pandas – Python library which is used for data manipulation and data analysis.

As we need to consume data sent by RabbitMQ producer in task 2, we need to pull and run RabbitMQ docker image in cloud virtual machine.

We execute below command on cloud virtual machine to pull and run RabbitMQ image.

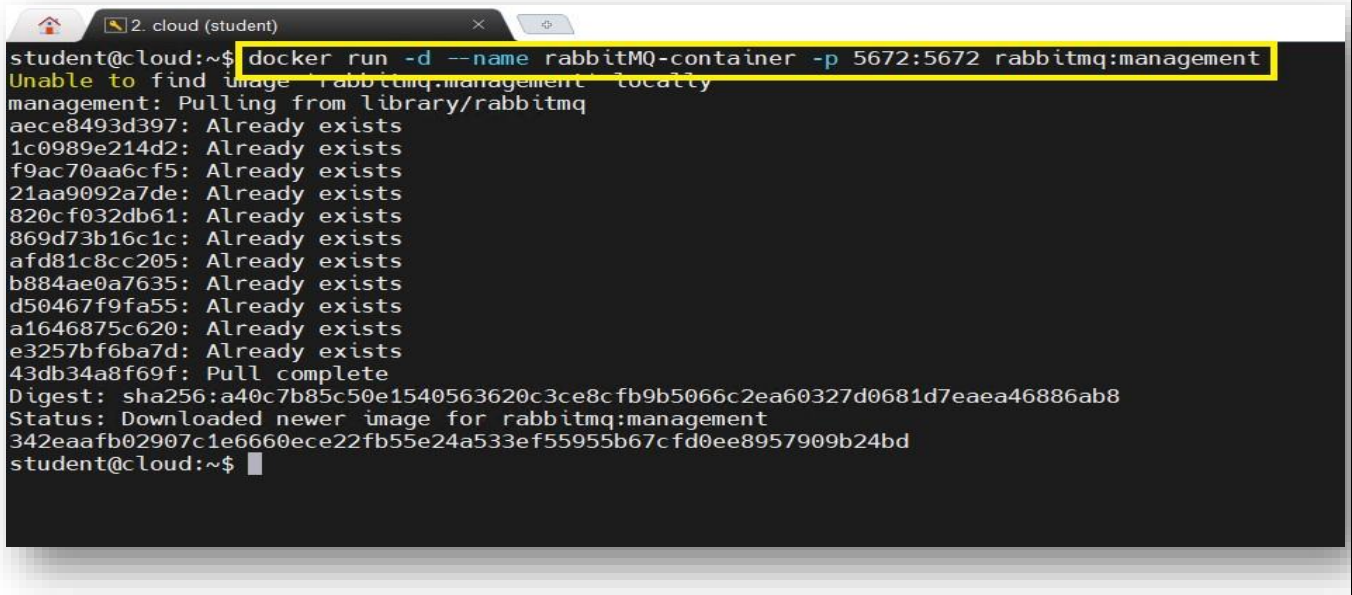

Figure 15: **Pulling and running RabbitMQ image on cloud virtual machine**

Code Description

In task 3 first we are importing all the required libraries such as json, pika, matplotlib, pandas and datetime. Then we are connecting to the RabbitMQ service in order to consume the data that is sent by RabbitMQ producer in task2. Once we receive the data in cloud layer, first we have to reformat the timestamp to date time format and print the reformatted timestamp on console.

There are two functions written with the name “visualize” and “predict” which are used for visualizing average PM2.5 daily data and to predict the trend of PM2.5 for next 15 days respectively. We pass the data with reformatted timestamp to both of these functions as a parameter. For machine learning model to predict the data for next 15 days, it makes use of a predefined ml_engine code. This predefined ml_engine code is downloaded and stored in the same directory where we are storing the python code file for task 3.

Code snippet for task 3

datapredictor.py

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

if __name__ == '__main__':
  rabbitmq_ip = "192.168.0.100"
  rabbitmq_port = 5672
  # Queue name
  rabbitmq_queque = "CSC8112test"

  # Feeding this time converted data to matplotlib for visualization
  def visualize(timeconvertdata):
    data_df = pd.DataFrame(timeconvertdata)

    # Initialize a canvas
    plt.figure(figsize=(80, 40), dpi=60)
    # Plot data into canvas
    plt.plot(data_df["Timestamp"], data_df["Value"], color="#FF3B1D", marker='.', linestyle="-")
    plt.title("Example data for demonstration")
    plt.xlabel("DateTime")
    plt.ylabel("Value")
    plt.xticks(rotation=90)
    # Save as file
    plt.savefig("figure1.png")
    # Directly display
    #plt.show()

  # supplying the data to ML engine in order to train and predict values for next 15 days
  def predict(timeconvertdata):

    data_df = pd.DataFrame(timeconvertdata)

    # Create ML engine predictor object
    predictor = MLPredictor(data_df)
    # Train ML model
    predictor.train()
    # Do prediction
    forecast = predictor.predict()
    # Get canvas
    fig = predictor.plot_result(forecast)
    fig.savefig("prediction.png")
    #fig.show()

  def callback(ch, method, properties, body):

    print(f"Printing PM2.5 average value data collected by RabbitMQ consumer: {json.loads(body)}")

    consumeddata = json.loads(body)

    # convert this consumeddata in required timestamp format

    timeconvertdata = []

    for i in consumeddata:
      date = datetime.strptime(i['Timestamp'], '%Y-%m-%d').strftime('%Y-%m-%d %H:%M:%S')
      timeconvertdata.append({'Timestamp': date, 'Value': i['Value']})

    # printing PM2.5 data with reformatted timestamp
    print("**********************************************************************************************************************************************")
    print("PM2.5 data with reformatted timestamp: ", timeconvertdata)
    print("**********************************************************************************************************************************************")

    visualize(timeconvertdata)
    predict(timeconvertdata)




    # Connect to RabbitMQ service with timeout 1min
  connection = pika.BlockingConnection(
        pika.ConnectionParameters(host=rabbitmq_ip, port=rabbitmq_port, socket_timeout=60))

  channel = connection.channel()
  # Declare a queue
  channel.queue_declare(queue=rabbitmq_queque)

  channel.basic_consume(queue=rabbitmq_queque,
                        auto_ack=True,
                        on_message_callback=callback)

  channel.start_consuming()
```

Figure 16: Task 3 python code

Code snippet for ml_engine

```python
'''
    Author: Rui Sun
    Date: 2022-09-25

    Predict timeseries data

    Official guide book of Prophet: https://facebook.github.io/prophet/docs/quick_start.html#python-api
'''

from prophet import Prophet

class MLPredictor(object):
    '''
    Example usage method:

        from ml_engine import MLPredictor

        predictor = MLPredictor(pm25_df)
        predictor.train()
        forecast = predictor.predict()


        fig = predictor.plot_result(forecast)
        fig.savefig(os.path.join("Your target dir path", "Your target file name))

    '''

    def __init__(self, data_df):
        '''
        :param data_df: Dataframe type dataset
        '''
        self.__train_data = self.__convert_col_name(data_df)
        self.__trainer = Prophet(changepoint_prior_scale=12)

    def train(self):
        self.__trainer.fit(self.__train_data)

    def __convert_col_name(self, data_df):
        data_df.rename(columns={"Timestamp": "ds", "Value": "y"}, inplace=True)
        print(f"After rename columns \n{data_df.columns}")
        return data_df

    def __make_future(self, periods=15):
        future = self.__trainer.make_future_dataframe(periods=periods)
        return future
```

```python
    def predict(self):
        future = self.__make_future()
        forecast = self.__trainer.predict(future)
        return forecast

    def plot_result(self, forecast):
        fig = self.__trainer.plot(forecast, figsize=(15, 6))
        return fig
```

Figure 17: ml_engine code

Output of Task 3: data prediction and visualization

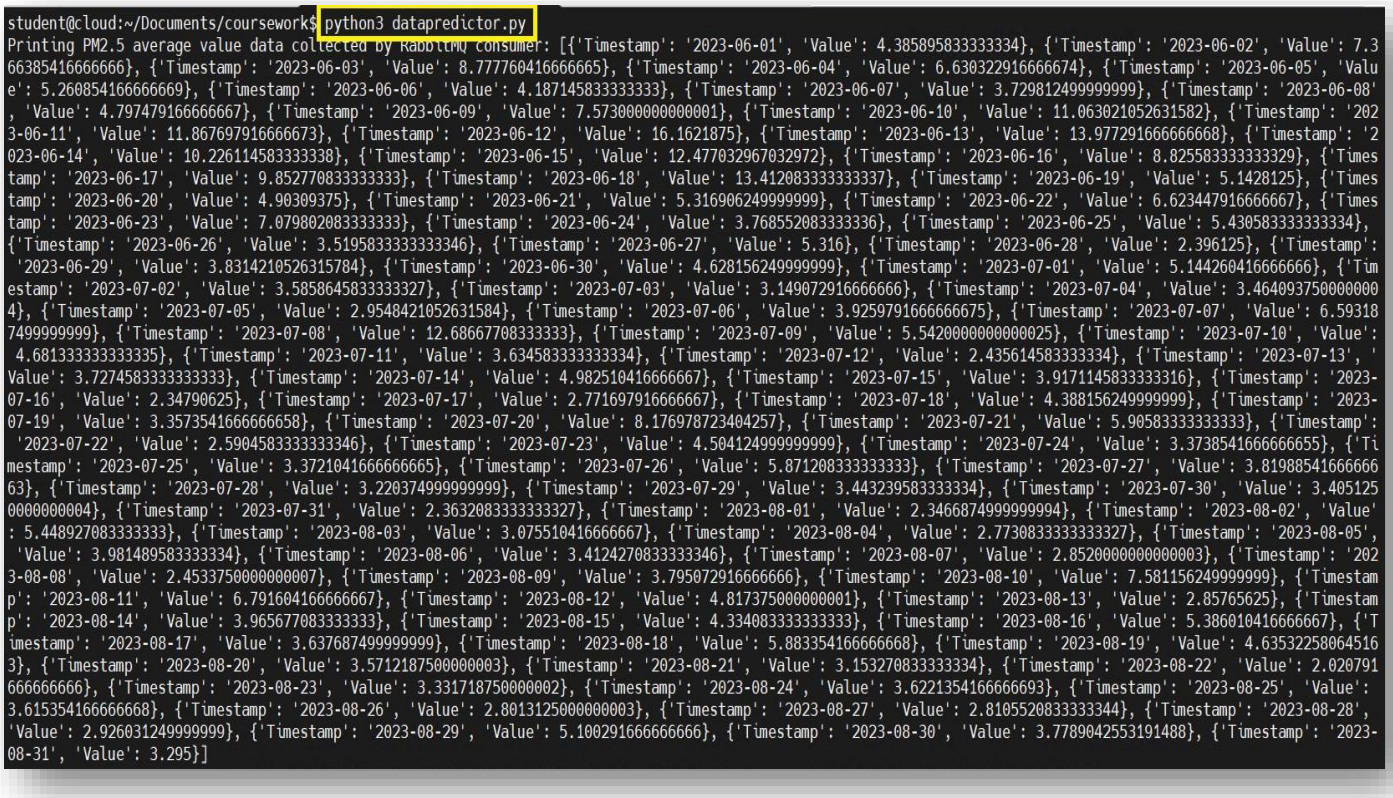

Figure 18: Printing average PM2.5 data collected by RabbitMQ consumer

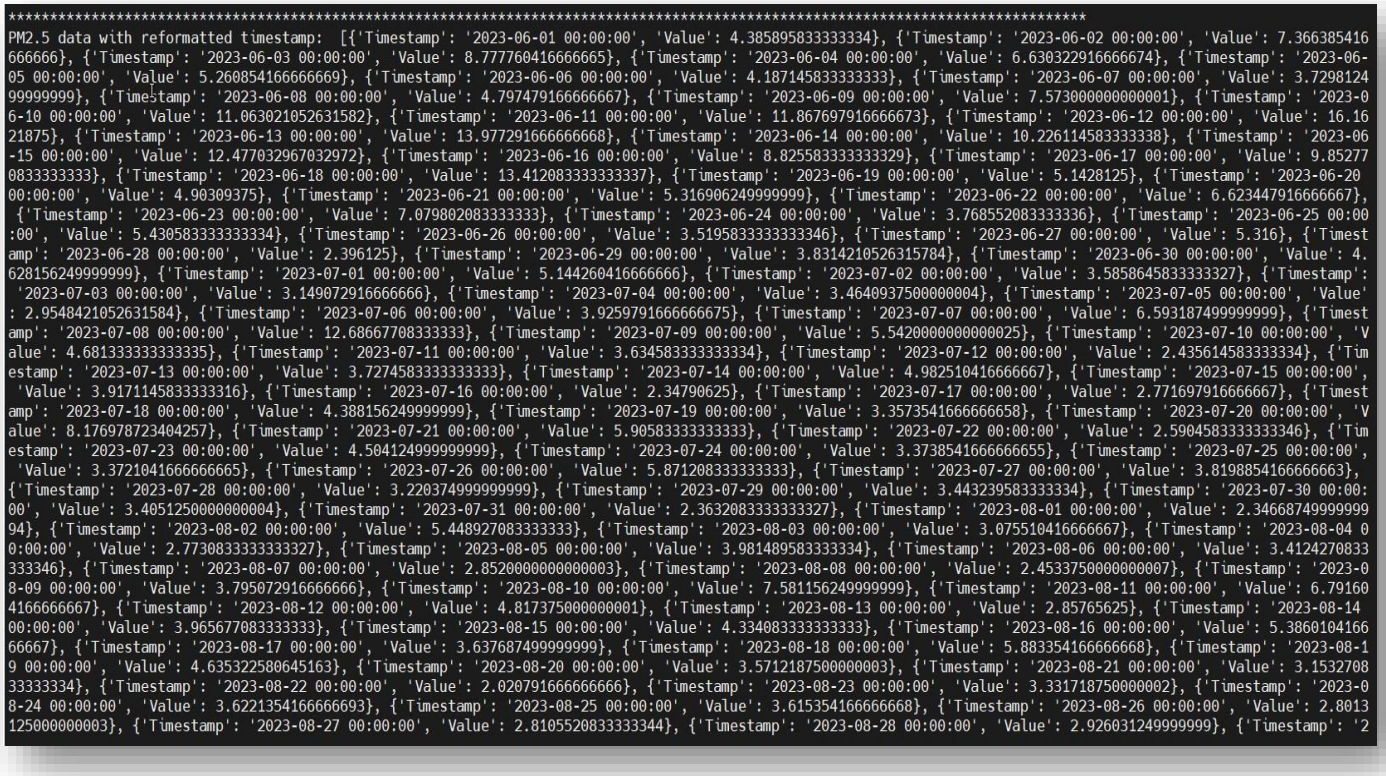

Figure 19: Printing data with reformatted timestamp

```
****************************************************************************************************************************************************
After rename columns
Index(['ds', 'y'], dtype='object')
12:18:28 - cmdstanpy - INFO - Chain [1] start processing
12:18:28 - cmdstanpy - INFO - Chain [1] done processing
```

Figure 20: plotting graphs successfully

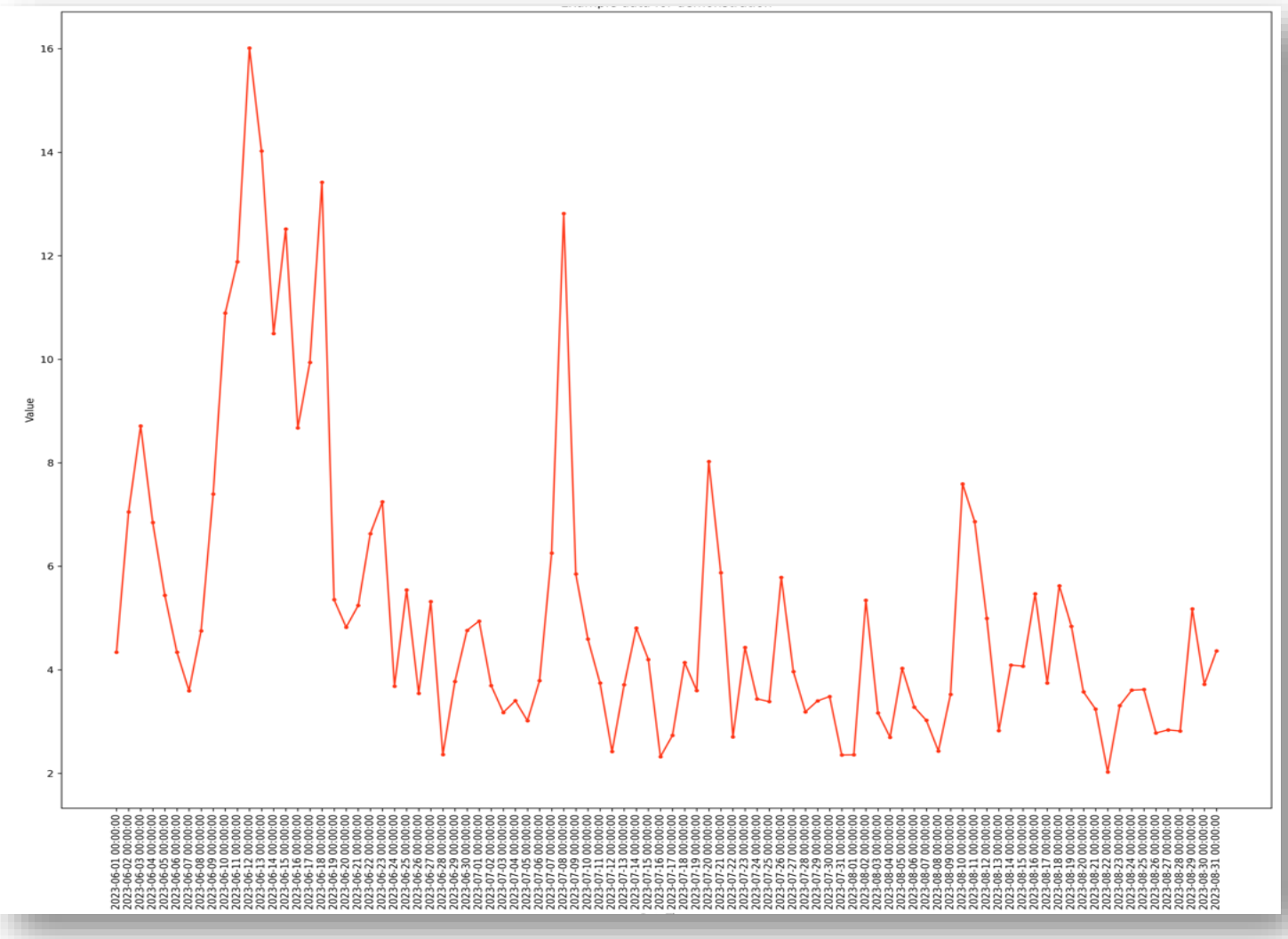

Figure 21: Data Visualization line chart

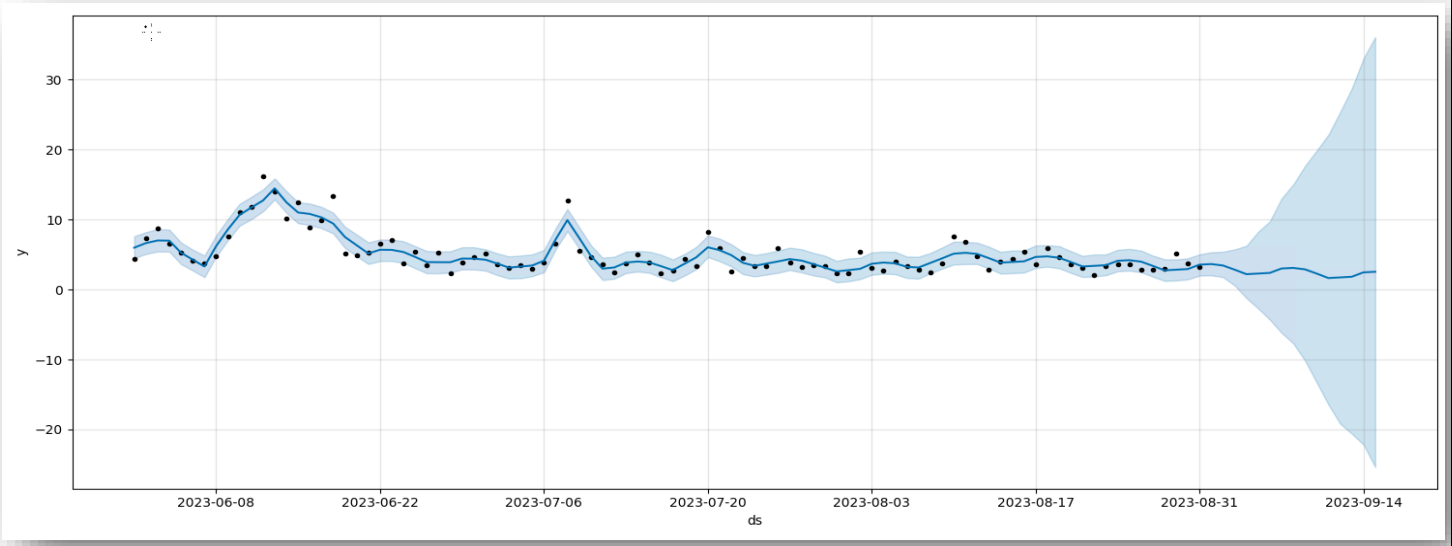

Figure 22: Data prediction graph

Screenshot of running services in docker environment

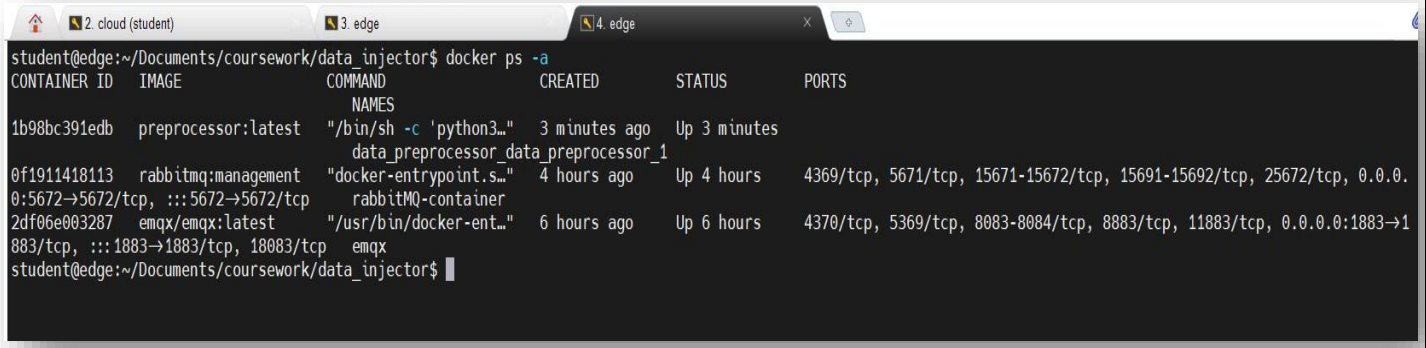

Figure 23: Docker services running on edge virtual machine

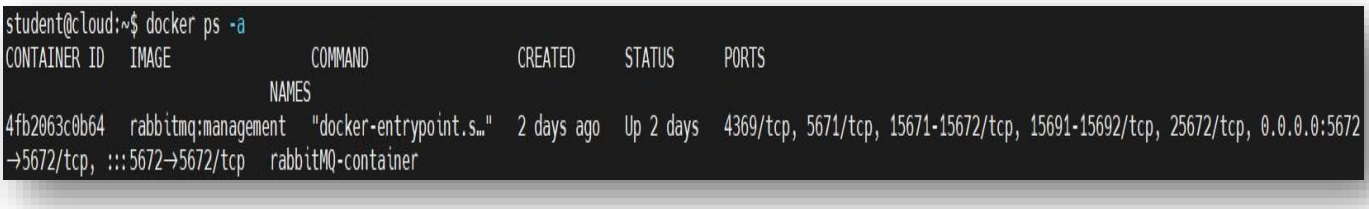

Figure 24: Docker services running on cloud virtual machine

Analytical discussion of results and related conclusion

Looking at the data visualization graph it is observed that for few days in month of June, 2023 the value of PM2.5 was considerably high as compared to rest of the days in selected duration. Also, in month of July we can see that there was a sudden spike in value of PM2.5 between 7th to 9th July, 2023 and then it became normal later. This might have happened due to some one-off activity or some external factors.

Looking at the forecast of first 15 days for the month of September, we can expect stable values of PM2.5 data with no sudden spikes. Overall if we compare starting from June month then we can see a declining trend in values of PM2.5 data over a period of time.

Finally, to conclude, this was an engaging project providing hands on experience and knowledge about collecting and processing data fetched from IOT sensors. Data flow between IOT, edge and cloud layer using MQTT and AMQP protocols. Using docker to pull and run required images and creating docker images of our own code and utilizing it further. Preprocessing and cleaning the raw data before passing it for visualization and prediction using machine learning model.