**Gas Leakage Detector with Telegram Alerts**

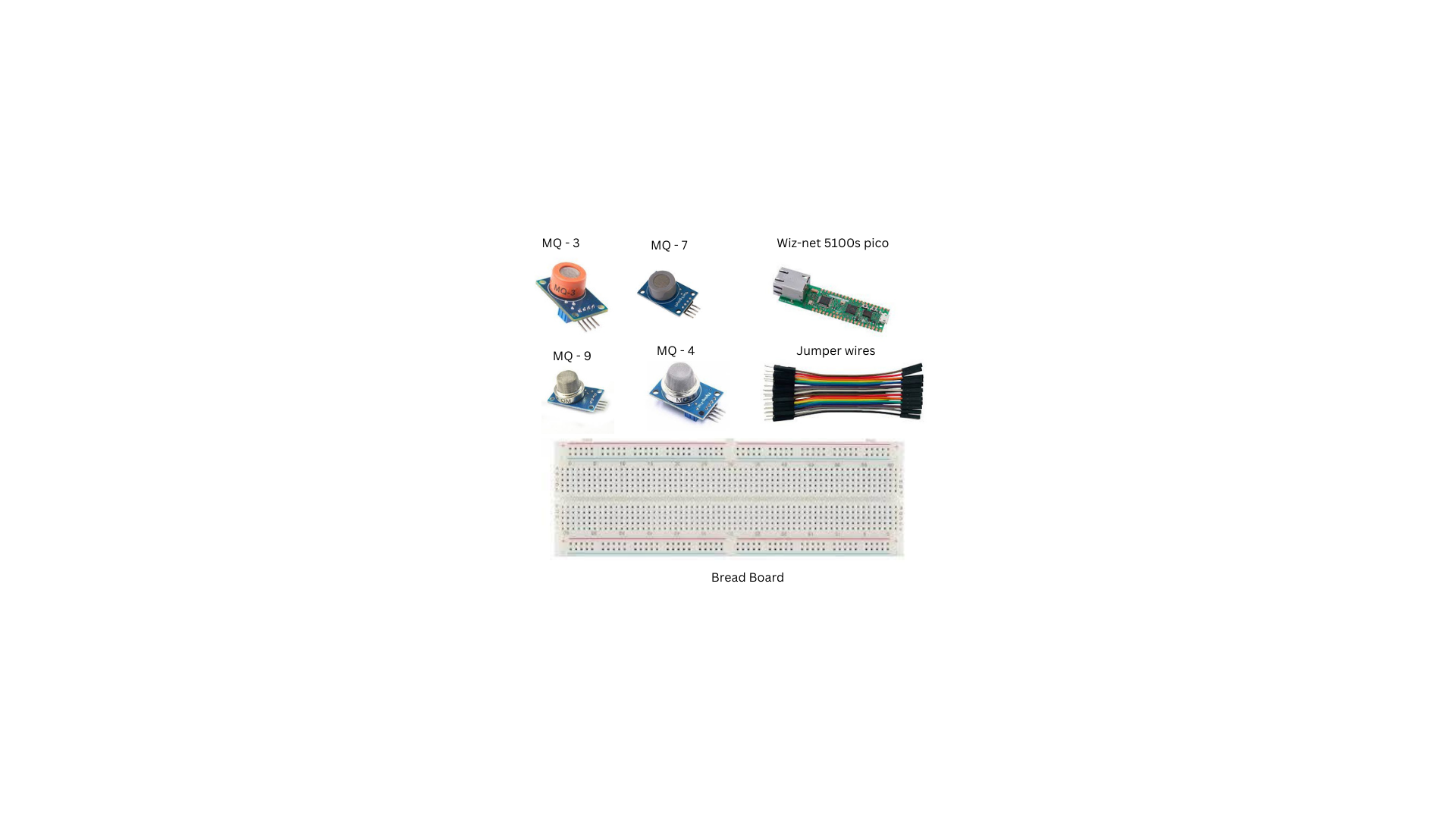
**Overview:**

The Gas Detection System with MQTT, InfluxDB, and Telegram is a project designed to monitor and analyze gas concentration levels in real-time. The system utilizes MQTT as the communication protocol to receive data from gas sensors and an MQTT broker. The system calculates the average gas concentration over a specified period using the obtained data and stores it in the InfluxDB database for further analysis. To ensure user notifications, the system integrates an automated Telegram bot, which provides real-time gas concentration updates and receives the average values. The bot sends alerts if the average concentration exceeds a predefined threshold, indicating a potential gas leak. The system's seamless integration of MQTT, InfluxDB, and Telegram guarantees timely alerts, efficient data storage, and informed decision-making, making it a powerful solution for gas detection and safety applications. MQTT is a lightweight publish/subscribe messaging tool, InfluxDB is a time-series database, and Telegram is a cloud-based instant messaging app. The integration of these technologies ensures timely alerts, efficient data storage, and informed decision-making.

**Things used in the project:**

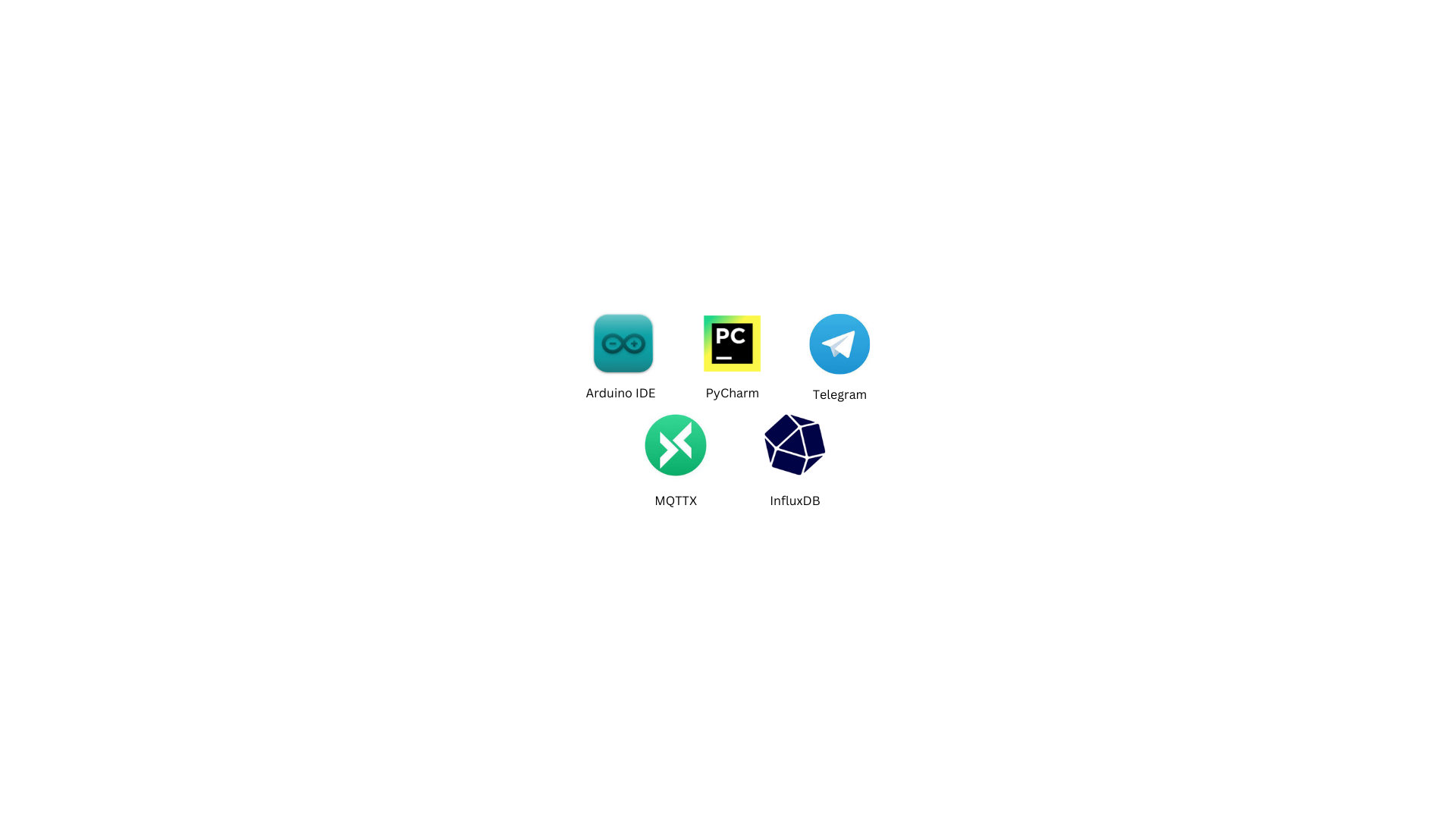
Hardware:

1. Wiz-net 5100s Pico Board
2. MQ-7 Gas sensor (Sensor for detecting Carbon Monoxide Gas)
3. MQ-9 Gas sensor (Sensor for detecting Carbon Monoxide, Coal Gas, Liquefied Gas)
4. MQ-3 Gas sensor (Sensor for detecting Alcohol Vapor)
5. MQ-4 Gas sensor (Sensor for detecting Methane (CNG) gas)
6. Jumper wires
7. Bread Board



Software apps:

1. Arduino IDE
2. MQTTX
3. PyCharm
4. Telegram API
5. InfluxDB

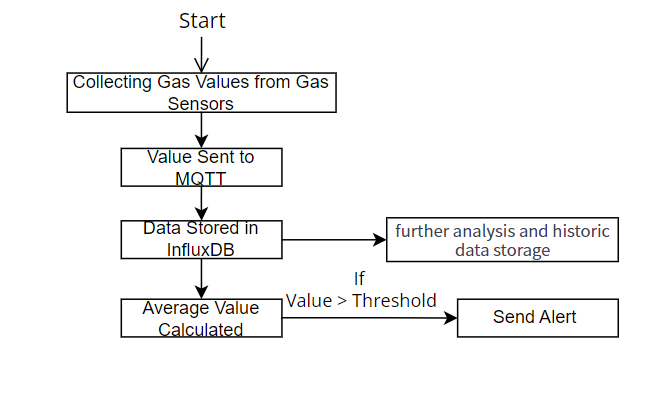


**Introduction:**

The Gas Detection System with MQTT, InfluxDB, and Telegram is a cutting-edge project designed to monitor and analyze gas concentration levels in real-time. It utilizes MQTT as the communication protocol to receive data from gas sensors and an MQTT broker. The system calculates the average gas concentration over a specified period using the obtained data and stores it in the InfluxDB database for further analysis.

To ensure user notifications, the system integrates an automated Telegram bot, which provides real-time gas concentration updates and receives the average values. The bot sends alerts if the average concentration exceeds a predefined threshold, indicating a potential gas leak. The system's seamless integration of MQTT, InfluxDB, and Telegram guarantees timely alerts, efficient data storage, and informed decision-making, making it a powerful solution for gas detection and safety applications.

**Flow Chart:**

****

* Gas sensors continuously collect gas concentration data.
* The data is sent to the MQTT Broker through MQTT.
* The MQTT Broker receives the data and stores it in the InfluxDB database.
* The Telegram bot sends real-time alerts to users if the gas concentration exceeds predefined threshold values.

**Getting Started:**

* 1. **The configuration of MQTT in Arduino IDE:**
* **Install PubSubClient Library:**
  + Open the Arduino IDE.
  + Navigate to "Sketch" > "Include Library" > "Manage Libraries..."
  + In the Library Manager, search for "PubSubClient" and click the "Install" button to install the library.
* **Define Constants for MQTT Broker:** In this part of the code, you need to define specific constants required to connect to the MQTT broker. These constants include the broker's IP address (e.g., 172.16.0.100), the port number (e.g., 1883), and the MQTT topics (i.e., GS-1, GS-2, GS-3, GS-4). The MQTT topics act as unique identifiers for different gas sensors, allowing the Arduino board to publish data to and receive data from the broker.
* **Create EthernetClient and PubSubClient Objects**: The code creates objects for the Ethernet client (ethClient) and the PubSubClient (client). The Ethernet client is responsible for handling the network communication, while the PubSubClient facilitates the MQTT communication with the broker. These objects are essential for establishing the connection to the MQTT broker and handling the communication.
* **Connect to the MQTT Broker**: To establish a connection with the MQTT broker, the reconnect() function is called. This function continuously attempts to reconnect until a successful connection is made. The successful connection enables the Arduino board to interact with the MQTT broker, sending and receiving data.
* **Publish Data to MQTT Broker:** If your Arduino board is collecting gas concentration data or any other sensor data, you can use the publish() method to send that data to the MQTT broker. By publishing data to specific MQTT topics, other devices or systems subscribed to those topics can receive and process the data. This functionality allows real-time monitoring and data sharing across devices.
* **Loop and Maintain Connection**: In the Arduino sketch's loop() function, you should call the loop() method of the MQTT client (PubSubClient). This allows the Arduino board to maintain the connection with the MQTT broker and handle incoming messages. The loop() function is essential for keeping the communication active and ensuring the board continues to publish data and receive any updates from the MQTT broker.
  1. **The Configuration of MQTT in PyCharm:**
* **Install necessary library:**
* Open your PyCharm project: Launch PyCharm and open your Python project.
* Open the Terminal: Go to "View" in the menu bar, then click on "Tool Windows," and finally select "Terminal."
* Install paho-mqtt: In the terminal window, type the following command and press Enter: **pip! Install paho-mqtt**

* **MQTT Broker Setup:** The code sets up the MQTT broker settings with the mqtt\_broker\_url(broker.emqx.io), mqtt\_port(1883), and mqtt\_keepalive(60). These settings define the address of the MQTT broker (domain name or IP address), the port number for the MQTT communication, and the keepalive interval for the MQTT connection.
* **MQTT Client Setup**: The code creates an MQTT client object using the mqtt.Client() function. The MQTT client is the main interface through which the Arduino board communicates with the MQTT broker. The on\_connect and on\_message callback functions are assigned to the client to handle the connection and incoming messages, respectively.
* **MQTT Connection:** The mqtt\_client.connect line establishes a connection with the MQTT broker using the specified broker URL, port, and keepalive interval. The MQTT client will attempt to connect to the broker, and if successful, it will call the on\_connect callback function.
* **MQTT Callbacks:** The on\_connect callback function is defined to be called when the MQTT client successfully connects to the broker. In this function, the code subscribes to the desired MQTT topics for each gas sensor specified in the gas\_sensors list. Subscribing to a topic allows the Arduino board to receive gas concentration data from the corresponding sensors.
* **MQTT Publishing** : The client.publish(topic, payload) method is used to publish data to the MQTT broker. Inside the loop() function, the code reads sensor values (in this case, gas concentration values) from different sensors and publishes them to their respective MQTT topics ("GS-1", "GS-2", "GS-3", "GS-4"). Each sensor value is converted to a string format (payload) before being published to the broker.
* **MQTT Loop:** The mqtt\_client.loop\_start() line starts a background loop for handling incoming MQTT messages. It allows the Arduino board to continuously listen for incoming messages from the broker while still executing other tasks in the main loop.
  1. **Configuration of Telegram-Bot in Python:**
* **Install necessary library:**
  + Open your PyCharm project: Launch PyCharm and open your Python project.
  + Open the Terminal: Go to "View" in the menu bar, then click on "Tool Windows," and finally select "Terminal."
  + Install requests: In the terminal window, type the following command and press Enter: **pip! Install requests**
  + Install Telegram: In the terminal window, type the following command and press Enter: **pip! Install telegram**
  + The requests library is a popular Python library used for making HTTP requests and interacting with web services, including APIs like the Telegram Bot API.
* **Telegram Setup:** The code defines a function called send\_telegram\_notification(message) responsible for sending Telegram notifications. To use this function, you need to obtain a Telegram Bot Token from the Telegram BotFather and replace the value of telegram\_bot\_token(6535063345:AAHun\_...........) with your actual bot token. Additionally, you need to specify the telegram\_chat\_id(1063……..) to determine the recipient of the notifications. The telegram\_chat\_id represents the ID of the user or group that will receive the notifications.
* **Telegram API URL:** The telegram\_api\_url(https://api.telegram.org/bot) is set to the Telegram Bot API endpoint for sending messages. The code constructs the API URL using the bot token obtained earlier and the sendMessage method. This URL will be used to make an HTTP POST request to the Telegram Bot API in order to send messages.
* **Sending Telegram Messages:** The send\_telegram\_notification(message) function takes a message as input and constructs the necessary data for the Telegram API request. It then sends an HTTP POST request to the Telegram API using the requests library to send the message to the specified telegram\_chat\_id.
* **Usage in Code:** The function send\_telegram\_notification(message) is called at various points in the code to send Telegram messages. For example, if the average gas concentration for a specific gas sensor exceeds a threshold, the function is called to send a notification indicating the detected "LEAKAGE."

1. **Configuration of Influxdb in Python:**

* **Install necessary library:**
* Open your PyCharm project: Launch PyCharm and open your Python project.
* Open the Terminal: Go to "View" in the menu bar, then click on "Tool Windows," and finally select "Terminal."
* Install influxdb\_client: In the terminal window, type the following command and press Enter: **pip! Install influxdb\_client**
* InfluxDB Initialization: The code defines a class called InfluxDB, which serves as a wrapper to interact with the InfluxDB database. The constructor \_\_init\_\_(self, bucket="AIoT") initializes the InfluxDB instance with the necessary configuration. It sets the InfluxDB URL, token, organization, and the default bucket for data storage.

self.url= "<http://10.21.70.16:8086>"

self.token="4SCyOmTPUhSgUiUSf0A8Fn31keTgErGXCoxo…………”

and self.org=" Christ"

* **Writing Data to InfluxDB:** The method write\_data\_to\_influxdb(self, measure, field\_name, value) is responsible for writing gas concentration data to the InfluxDB database. It takes the measure, field\_name, and value as parameters. The measure = "\_measurement1" , the field\_name = "gas\_concentration\_" + sensor["name"].
* **InfluxDB Query:** The method query(self, measure, start) is used to query data from the InfluxDB database. It takes the measure and start time as parameters. The method constructs a Flux query to fetch data from the specified measurement (measure) starting from the provided time (start). The query result is returned as tables.
* **Calculating Average:** The method query\_mean(self, measure, start) is similar to the query() method but includes a mean() function to calculate the average value for the queried data. It helps in calculating the average gas concentration over a specified period.
* **Usage in Code:** In the on\_message() function, the received gas concentration data is written to InfluxDB using the write\_data\_to\_influxdb() method. Additionally, in the main loop, the query\_mean() method is used to fetch the average gas concentration for each gas sensor over the last 2 minutes ("-2m"), and the process\_gas\_sensor() function processes the data and stores it in InfluxDB.

**Methodology:**

* **Overview of Methodology**

****

The Gas Detection System with MQTT, InfluxDB, and Telegram involves several essential phases to efficiently gather real-time gas concentration data, provide timely alerts, and store the collected information for further analysis and monitoring.

**Setup Phase:**

* In this initial phase, we create the necessary infrastructure to facilitate communication and data storage.
* We start by setting up a Telegram bot, an InfluxDB server, and an MQTT broker. These components will play crucial roles in the functioning of the Gas Detection System.
* The Telegram bot will be used to send notifications and alerts to users based on the gas concentration data.
* The InfluxDB server will serve as the database to store the gas concentration data received from the MQTT client.
* The MQTT broker facilitates communication between the MQTT client and the gas sensor, enabling the transmission of real-time gas concentration data.

**Library Installation Phase:**

* Once the infrastructure is in place, the next step is to install the required Python libraries using pip. These libraries will enable seamless interactions between the MQTT client, the InfluxDB server, and the Telegram API.
* Necessary libraries might include 'paho-mqtt' for MQTT communication, 'influxdb' for interacting with InfluxDB, and 'python-telegram-bot' for interacting with the Telegram API.

**MQTT Client Configuration Phase:**

* After installing the required libraries, the MQTT client is configured to subscribe to the gas concentration topic. It establishes a connection with the MQTT broker and waits for incoming messages containing real-time gas concentration data from the gas sensor.
* The client ensures that the environment is properly configured, including providing the appropriate credentials and settings to connect to the MQTT broker.

**Real-Time Gas Concentration Data Retrieval Phase:**

* The Gas Detection System employs an MQ-7, MQ-9, MQ-3, MQ-4 Gas Sensors connected to the Wiznet W5100S Pico Board to provide the MQTT client with real-time gas concentration data.
* The gas sensors communicates gas concentration information through analog voltage values.
* Upon receiving messages from the sensor, the MQTT client processes the incoming data and stores it for further analysis.

**Calculation of Average Gas Concentration Phase:**

* To assess the gas concentration over a specific period, such as 2 minutes, the Gas Detection System implements a method to gather received data for that duration.
* The system computes the average gas concentration based on the collected data, providing a more comprehensive understanding of the overall gas levels during that period.
* The average gas concentration is then saved in a list for further use.

**Telegram Alerting Phase:**

* The Gas Detection System utilizes the Telegram API to send alerts and notifications to users based on the estimated average gas concentration.
* If the average concentration exceeds a specified threshold, the system sends a Telegram message warning users of a potential gas leak, prompting them to take appropriate actions.
* Conversely, if the average concentration falls below the threshold, the system sends a message indicating no leakage and providing reassurance.

**Data Storage in InfluxDB Phase:**

* The final step involves ensuring that all gas concentration data received is stored in the InfluxDB database for further analysis and monitoring.
* The system utilizes the InfluxDB client library to implement a method that efficiently writes the received data to the InfluxDB server.
* By storing data in InfluxDB, users can perform data analysis, generate visualizations, and gain insights into gas concentration trends over time.

**Code Explanation:**

* **Arduino Code:**

|  |
| --- |
| #include <SPI.h>  #include <Ethernet.h>  #include <PubSubClient.h> |

* These are the three libraries that are used in this code. The SPI.h library is used to communicate with the gas sensors. The Ethernet.h library is used to connect to the internet. The PubSubClient.h library is used to publish and subscribe to MQTT topics.

|  |
| --- |
| byte mac[] = { 0xDE, 0xED, 0xBA, 30, 0xFE, 0xED };  IPAddress ip(172, 16, 0, 100);  IPAddress server(44, 195, 202, 69); |

* The code is defining some variables that will be used later in the code. The mac[] variable is an array of bytes that represents the MAC address of the Ethernet shield. The Ip variable is an object of the IPAddress class that represents the IP address of the Arduino board. The server variable is also an object of the IP Address class that represents the IP address of the MQTT broker.

|  |
| --- |
| const int Sensor1Pin = 35;  const int Sensor2Pin = 34;  const int Sensor3Pin = 32;  const int Sensor4Pin = 31; |

* The Sensor1Pin, Sensor2Pin, Sensor3Pin, and Sensor4Pin constants represent the pins on which the gas sensors are connected.
  + Sensor1 - MQ-7 Gas sensor (Sensor for detecting Carbon Monoxide Gas)
  + Sensor2 - MQ-9 Gas sensor (Sensor for detecting Carbon Monoxide, Coal Gas, Liquefied Gas)
  + Sensor3 - MQ-3 Gas sensor (Sensor for detecting Alcohol Vapor)
  + Sensor4 - MQ-4 Gas sensor (Sensor for detecting Methane (CNG) gas)

|  |
| --- |
| EthernetClient ethClient;  PubSubClient client(ethClient); |

* It define an object of the EthernetClient class and an object of the PubSubClient class. The ethClient object is used for communicating with the Ethernet shield and the client object is used for publishing and subscribing to MQTT topics

|  |
| --- |
| void callback(char\* topic, byte\* payload, unsigned int length) {  String message = "";  for (int i = 0; i < length; i++) {  message += (char)payload[i];  }  } |

* It defines a function called callback() which will be called when a message is received on a subscribed topic. This function reads the message payload and stores it in a string variable.

|  |
| --- |
| void reconnect() {  // Loop until we're reconnected  while (!client.connected()) {  Serial.print("Attempting MQTT connection...");  // Attempt to connect  if (client.connect("arduinoClient30")) {  Serial.println("connected");  // Once connected, publish an announcement...  client.publish("Out", "hello world");  // ... and resubscribe  client.subscribe("gas\_concentration");  } else {  Serial.print("failed, rc=");  Serial.print(client.state());  Serial.println(" try again in 10 seconds");  // Wait 5 seconds before retrying  delay(10000);  }  }  } |

* It defines a function called reconnect() which will be called when the client loses its connection to the MQTT broker. This function attempts to reconnect to the broker until it is successful.

|  |
| --- |
| void setup()  {  Serial.begin(57600);  Ethernet.init(17);  client.setServer(server, 1883);  client.setCallback(callback);  Ethernet.begin(mac);  delay(1500);  pinMode(Sensor1Pin, INPUT);  pinMode(Sensor2Pin, INPUT);  pinMode(Sensor3Pin, INPUT);  pinMode(Sensor4Pin, INPUT);  } |

* It defines a function called setup() which will be called once when the Arduino board is powered on or reset. This function initializes serial communication, initializes Ethernet, sets up pins for gas sensors as input pins and sets up objects for EthernetClient and PubSubClient classes.

|  |
| --- |
| void loop()  {  if (!client.connected()) {  reconnect();  }  client.loop();  // Read sensor values  float Sensor1Value = readSensor1();  float Sensor2Value = readSensor2();  float Sensor3Value = readSensor3();  float Sensor4Value = readSensor4();  // Publish the sensor values to their respective MQTT topics  char Sensor1Topic[] = "GS-1"; #GS-1 Is the MQTT topic for sensor MQ-7  char Sensor2Topic[] = "GS-2"; #GS-2 Is the MQTT topic for sensor MQ-9  char Sensor3Topic[] = "GS-3"; #GS-3 Is the MQTT topic for sensor MQ-3  char Sensor4Topic[] = "GS-4"; #GS-4 Is the MQTT topic for sensor MQ-4  char payload[10];  snprintf(payload, sizeof(payload), "%f", Sensor1Value);  client.publish(Sensor1Topic, payload);  snprintf(payload, sizeof(payload), "%f", Sensor2Value);  client.publish(Sensor2Topic, payload);  snprintf(payload, sizeof(payload), "%f", Sensor3Value);  client.publish(Sensor3Topic, payload);  snprintf(payload, sizeof(payload), "%f", Sensor4Value);  client.publish(Sensor4Topic, payload);  // Adjust the interval between readings and publications  delay(10000);  }  // Function to read gas sensor value  float readSensor1() {  // Read the analog value from the gas sensor using analogRead function  int Sensor1Value = analogRead(Sensor1Pin);  // Display the raw analog value on the Serial Monitor  Serial.print("Sensor 1 Value: ");  Serial.println(Sensor1Value);  return Sensor1Value; // Return the raw analog value without mapping or scaling  }  // Function to read sensor 2 value  float readSensor2() {  // Read the analog value from the second sensor using analogRead function  int Sensor2Value = analogRead(Sensor2Pin);  // Display the raw analog value on the Serial Monitor  Serial.print("Sensor 2 Value: ");  Serial.println(Sensor2Value);  return Sensor2Value; // Return the raw analog value without mapping or scaling  }  // Function to read sensor 3 value  float readSensor3() {  // Read the analog value from the third sensor using analogRead function  int Sensor3Value = analogRead(Sensor3Pin);  // Display the raw analog value on the Serial Monitor  Serial.print("Sensor 3 Value: ");  Serial.println(Sensor3Value);  return Sensor3Value; // Return the raw analog value without mapping or scaling  }  // Function to read sensor 4 value  float readSensor4() {  // Read the analog value from the third sensor using analogRead function  int Sensor4Value = analogRead(Sensor4Pin);  // Display the raw analog value on the Serial Monitor  Serial.print("Sensor 4 Value: ");  Serial.println(Sensor4Value);  return Sensor4Value; // Return the raw analog value without mapping or scaling  } |

* The loop() function checks if there is a connection to MQTT broker and if not, it tries to reconnect. It reads values from gas sensors and publishes them to MQTT broker. It checks if the client is connected to the MQTT broker. If it is not connected, the reconnect() function is called which attempts to reconnect to the broker until it is successful. The client.loop() function processes any incoming messages from the MQTT broker. The function reads values from gas sensors using functions readSensor1(), readSensor2(), readSensor3(), and readSensor4(). These functions read the analog value from each sensor using the analogRead() function and return the raw analog value without mapping or scaling. The function publishes the sensor values to their respective MQTT topics using the snprintf() and client.publish() functions. The last line of the function sets an interval of 10 seconds between readings and publications using the delay() function.
* Sensor1 - MQ-7 Gas sensor (Sensor for detecting Carbon Monoxide Gas)
* Sensor2 - MQ-9 Gas sensor (Sensor for detecting Carbon Monoxide, Coal Gas, Liquefied Gas)
* Sensor3 - MQ-3 Gas sensor (Sensor for detecting Alcohol Vapor)
* Sensor4 - MQ-4 Gas sensor (Sensor for detecting Methane (CNG) gas)
* **Python Code:**

The code begins by importing necessary libraries to enable communication, logging, data storage, and other functionalities. We will be analyzing the code and its functionality.

|  |
| --- |
| import paho.mqtt.client as mqtt  import logging  import requests  import time  from influxdb\_client import Point, WritePrecision  from influxdb\_client.client import influxdb\_client  from influxdb\_client.client.write\_api import SYNCHRONOUS  from influxdb\_client import InfluxDBClient |

* paho.mqtt.client: This library provides an MQTT client implementation for Python, allowing the system to interact with the MQTT broker and subscribe to topics.
* logging: The logging module is used to set up logging for potential errors or informative messages.
* requests: This library is used for making HTTP requests, but it is not used in the code snippet provided.
* time: The time module is used for time-related functions and is likely utilized for timing or delays in the program execution.
* influxdb\_client: This library enables interaction with the InfluxDB database.
* Point, WritePrecision: These are classes from the influxdb\_client library that are used to define data points and specify the precision of data writes to the InfluxDB database.
* InfluxDBClient: This class allows the code to connect to the InfluxDB database.

|  |
| --- |
| logging.basicConfig(level=logging.INFO) |

* The code sets up logging to provide insights into any potential errors or relevant information during the program execution. The logging level is set to INFO, which means that informational messages and higher-level messages (e.g., WARNING, ERROR) will be logged. This helps in debugging and monitoring the system.

|  |
| --- |
| mqtt\_broker\_url = 'broker.emqx.io'  mqtt\_port = 1883  mqtt\_keepalive = 60 |

* mqtt\_broker\_url: This variable holds the domain name or IP address of the MQTT broker that the Gas Detection System will connect to.
* mqtt\_port: This variable specifies the port number on which the MQTT broker is running. The default MQTT port is 1883.
* mqtt\_keepalive: This variable defines the maximum time interval (in seconds) that the system will allow between communications with the broker. If no messages are sent or received within this interval, the system will send a "ping" message to keep the connection alive.

|  |
| --- |
| def on\_connect(client, userdata, flags, rc):  if rc == 0:  print("Connected to MQTT broker")  for sensor in gas\_sensors:  client.subscribe(sensor["topic"]) # Subscribe to the desired MQTT topic for each gas sensor  else:  print(f"Connection failed with code {rc}") |

* This function is called when the MQTT client successfully connects to the MQTT broker. It subscribes to the desired MQTT topic for each gas sensor listed in the gas\_sensors list. If the connection fails, it prints an appropriate error message.

|  |
| --- |
| def send\_telegram\_notification(message):  # Telegram Bot token and chat ID  telegram\_bot\_token = 'Your\_Telegram\_Bot\_Token'  telegram\_chat\_id = 'Your\_Telegram\_Chat\_ID'  # Telegram API URL  telegram\_api\_url = f"https://api.telegram.org/bot{telegram\_bot\_token}/sendMessage"  # Data to send in the POST request  data = {  "chat\_id": telegram\_chat\_id,  "text": message,  }  try:  response = requests.post(telegram\_api\_url, data=data)  response.raise\_for\_status() # Raise an exception if the response status is not 2xx (indicating success)  print("Message sent to Telegram successfully")  except requests.exceptions.RequestException as e:  print("Error sending message to Telegram:", e) |

* This function sends a notification message to Telegram using the Telegram Bot API. It takes the message as input, along with the Telegram bot token and chat ID. The function constructs the API URL and sends the message using a POST request to the Telegram API. It also handles exceptions in case there is an error in sending the message.

|  |
| --- |
| def on\_message(client, userdata, msg):  global send\_mqtt\_messages  # Decode the payload of the MQTT message  payload = msg.payload.decode('utf-8')  print("Received MQTT message:", payload)  # Find the corresponding sensor based on the topic  for sensor in gas\_sensors:  if msg.topic == sensor["topic"]:  # Write the received value to InfluxDB with the unique field name  field\_name = "gas\_concentration\_" + sensor["name"]  ts.write\_data\_to\_influxdb(sensor["measure"], field\_name, float(payload))  # Call the function to perform some action after writing data to InfluxDB  ts.after\_writing\_to\_influxdb()  # Set the flag to True after writing to InfluxDB  send\_mqtt\_messages = True  break |

* This function is called whenever a new MQTT message is received from the broker. It decodes the payload of the message and prints the received MQTT message. It then finds the corresponding gas sensor based on the topic and processes the received data.
* If a sensor's average gas concentration value exceeds 30, it sends a Telegram notification with the sensor name and the average value, indicating a gas leak. It also writes the average value to InfluxDB with a unique field name for each sensor.

|  |
| --- |
| def process\_gas\_sensor(sensor):  # Calculate the average of the received values  if len(sensor["mqtt\_values"]) > 0:  average = sum(sensor["mqtt\_values"]) / len(sensor["mqtt\_values"])  field\_name = "gas\_concentration\_" + sensor["name"] # Generate unique field name for each sensor  if average > 30:  send\_telegram\_notification(f"{sensor['name']} - Average value: {average:.2f} LEAKAGE DETECTED!!!")  print(f"{sensor['name']} - Average value: {average:.2f}")  # Write the average value to InfluxDB with the unique field name  ts.write\_data\_to\_influxdb(sensor["measure"], field\_name, average)  else:  print(f"{sensor['name']} - No Leakage")  # Clear the list for the next interval  sensor["mqtt\_values"].clear() |

* This function processes data for a specific gas sensor. It calculates the average of the received values and checks if it exceeds 30 (indicating a potential gas leak). If it does, it calls the send\_telegram\_notification function to send a warning message to Telegram with the sensor name and average value. It also writes the average value to InfluxDB with a unique field name for each sensor.
* The function clears the list of received values for the next interval.

|  |
| --- |
| # Create MQTT client  mqtt\_client = mqtt.Client()  # Set callback functions  mqtt\_client.on\_connect = on\_connect  mqtt\_client.on\_message = on\_message  # Connect to MQTT broker  mqtt\_client.connect(mqtt\_broker\_url, mqtt\_port, mqtt\_keepalive)  # Start MQTT loop in the background  mqtt\_client.loop\_start() |

* The code then creates an MQTT client and sets its callback functions. The on\_connect function is called when the client successfully connects to the MQTT broker, and the on\_message function is called when a new message is received. It then connects the MQTT client to the MQTT broker and starts the MQTT loop in the background to handle incoming messages.

|  |
| --- |
| class InfluxDB:  def \_\_init\_\_(self, bucket="AIoT"):  # InfluxDB configurations  self.token = "Your\_InfluxDB\_Token"  self.org = "Your\_InfluxDB\_Organization"  self.url = "http://10.21.70.16:8086" # Replace with your InfluxDB server URL  self.bucket = bucket  self.client = influxdb\_client.InfluxDBClient(url=self.url, token=self.token, org=self.org)  self.write\_api = None  self.delete\_api = None  self.query\_api = None |

* The InfluxDB class is designed to manage interactions with the InfluxDB database. It is initialized with a bucket parameter, which specifies the name of the InfluxDB bucket to which data will be written. The class sets up the necessary configurations to connect to the InfluxDB server using the provided URL, token, and organization.
* token: The authentication token required to access the InfluxDB server.
* org: The organization to which the InfluxDB belongs.
* url: The URL of the InfluxDB server.
* bucket: The name of the InfluxDB bucket where data will be written.
* client: An instance of the InfluxDBClient class, which connects to the InfluxDB server using the provided configurations.
* write\_api, delete\_api, query\_api: These attributes store instances of the respective APIs (write, delete, and query) for InfluxDB. They are initialized to None and will be created when needed.

|  |
| --- |
| def delete(self, measure, start, stop):  if self.delete\_api is None:  self.delete\_api = self.client.delete\_api()  # start = "1970-01-01T00:00:00Z"  # stop = "2100-01-01T00:00:00Z"  return self.delete\_api.delete(start, stop, "\_measurement=" + measure, bucket=self.bucket, org=self.org)  def write\_data\_to\_influxdb(self, measure, field\_name, value):  # Write the received gas concentration data to InfluxDB  if self.write\_api is None:  self.write\_api = self.client.write\_api(write\_options=SYNCHRONOUS)  point = (Point(measure).field(field\_name, value))  try:  self.write\_api.write(bucket=self.bucket, org=self.org, record=point)  print("Data point written to InfluxDB:", point.to\_line\_protocol())  except Exception as e:  print("Error writing to InfluxDB:", e)  def after\_writing\_to\_influxdb(self):  print("Data written to InfluxDB. Perform desired action here.")  def query(self, measure, start):  if self.query\_api is None:  self.query\_api = self.client.query\_api()  query = f"""from(bucket: "{self.bucket}")  |> range(start: {start})  |> filter(fn: (r) => r.\_measurement == "{measure}")"""  tables = self.query\_api.query(query, org=self.org)  return tables  def query\_mean(self, measure, start):  if self.query\_api is None:  self.query\_api = self.client.query\_api()  query = f"""from(bucket: "{self.bucket}")  |> range(start: {start})  |> filter(fn: (r) => r.\_measurement == "{measure}")  |> mean()"""  tables = self.query\_api.query(query, org=self.org)  return tables |

* delete(self, measure, start, stop): This method is used to delete data from the InfluxDB database. It takes the measure (measurement name), start (start time of data to delete), and stop (end time of data to delete) as inputs. It returns the result of the delete operation.
* write\_data\_to\_influxdb(self, measure, field\_name, value): This method writes a data point to the InfluxDB database. It takes measure (measurement name), field\_name (the field name for the gas concentration value), and value (the actual gas concentration value) as inputs. It writes the data point to InfluxDB using the synchronous write API. The Point class from the influxdb\_client library is used to define the data point.
* after\_writing\_to\_influxdb(self): This method is called after writing data to InfluxDB. It is a placeholder for performing any desired action or additional processing after the data has been successfully written to the database. The method currently prints a message indicating that the data has been written.
* query(self, measure, start): This method is used to query data from the InfluxDB database based on the specified measure (measurement name) and start (start time of the data range to query). It constructs an InfluxQL query to fetch data from the specified bucket and organization, and returns the query result in the form of tables.
* query\_mean(self, measure, start): Similar to the query method, this method performs a query to calculate the mean value of the specified measure within the specified data range (based on start). It returns the query result in the form of tables.

|  |
| --- |
| gas\_sensors = [  {  "name": "Sensor1",  "measure": "MQ-7",  "topic": "GS-1",  "mqtt\_values": []  },  {  "name": "Sensor2",  "measure": "MQ-9",  "topic": "GS-2",  "mqtt\_values": []  },  {  "name": "Sensor3",  "measure": "MQ-3",  "topic": "GS-3",  "mqtt\_values": []  },  {  "name": "Sensor4",  "measure": "MQ-4",  "topic": "GS-4",  "mqtt\_values": []  }  ] |

* The code defines a list gas\_sensors that holds information about different gas sensors and their settings. Each sensor is represented as a dictionary with the following properties:
* name: A descriptive name for the gas sensor.
* measure: The measurement name that will be used in InfluxDB to store data from this sensor.
* topic: The MQTT topic to which the sensor is subscribed to receive gas concentration data.
* mqtt\_values: A list that stores received MQTT values from the sensor. This list will be used to calculate the average gas concentration for each sensor.

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  ts = InfluxDB(bucket="AIoT") |

* The program initializes an InfluxDB object ts with the specified bucket parameter as "AIoT." This object will be used to interact with the InfluxDB database and perform data operations.

|  |
| --- |
| try:  while True:  # Set the flag to False at the start of each iteration  send\_mqtt\_messages = False  # Query data for the last 2 minutes for each sensor  for sensor in gas\_sensors:  tables = ts.query\_mean(sensor["measure"], "-2m")  # Calculate the average of query results for each sensor  for table in tables:  for record in table.records:  value = record.get\_value()  sensor["mqtt\_values"].append(value)  # Process data for the current sensor  process\_gas\_sensor(sensor)  # Wait for 2 minutes before calculating the next average  time.sleep(120)  except KeyboardInterrupt:  # This block will execute when Ctrl+C is pressed  print("Script terminated by user.")  # Optionally, you can send a final message to Telegram notifying the script termination  send\_telegram\_notification("Script terminated by user.") |

* The program initializes an InfluxDB object ts with the specified bucket parameter as "AIoT." This object will be used to interact with the InfluxDB database and perform data operations.
* The main loop runs indefinitely and continuously monitors the gas concentration data from the gas sensors. It performs the following steps in each iteration:
  + - * + Querying Data and Calculating Average:

Within the loop, for each gas sensor in the gas\_sensors list, the program queries the InfluxDB database for the average gas concentration over the last 2 minutes. It uses the ts.query\_mean method from the InfluxDB object to perform the query. The query result is a list of tables, and the program extracts the average value from each table and stores it in the corresponding sensor's mqtt\_values list.

* + - * + Processing Gas sensors:

After calculating the average for each sensor, the program calls the process\_gas\_sensor function for the current sensor. This function calculates the overall average of the received values and compares it to a threshold of 30. If the average exceeds 30, it sends a Telegram notification to warn about a potential gas leak. It also writes the average value to the InfluxDB database using the ts.write\_data\_to\_influxdb method.

* + - * + Sleep and Repeat:

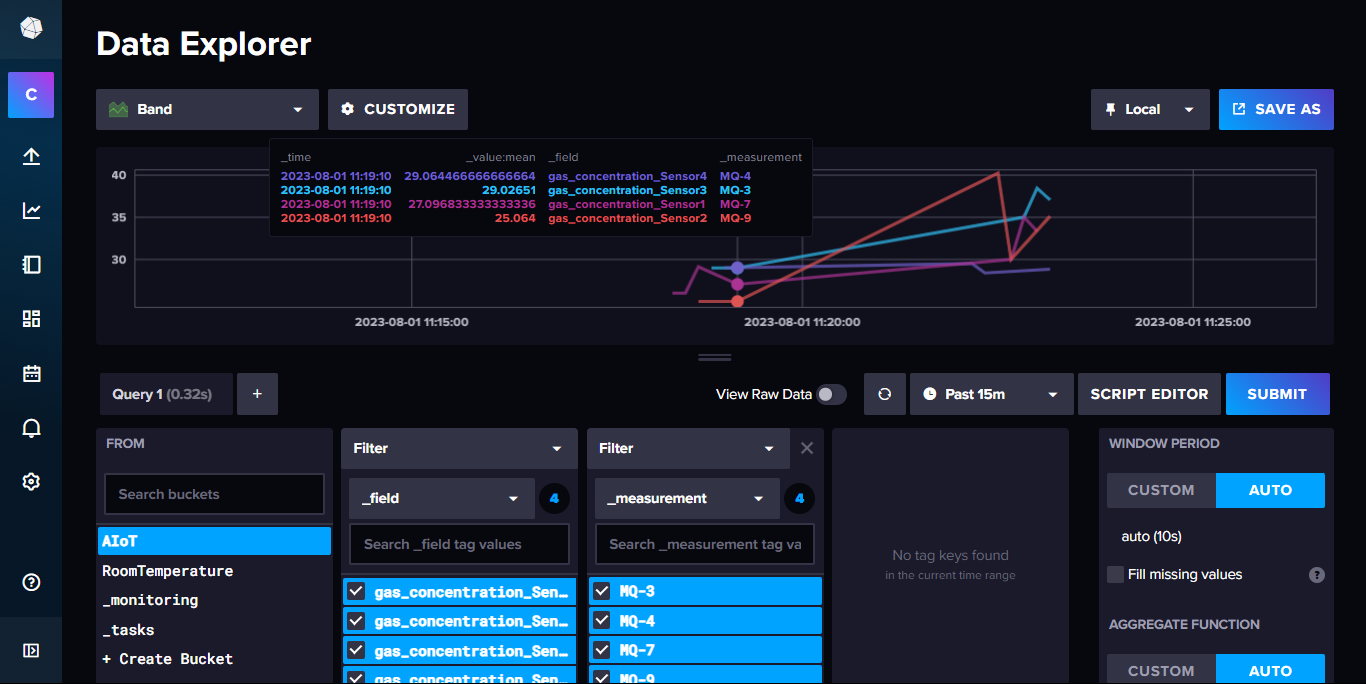
After processing data for all sensors, the program waits for 2 minutes (120 seconds) using time.sleep(120) before starting the next iteration. This allows the program to collect new data points for the next 2-minutes interval and calculate the averages again.

* + - * + Keyboard Interupt handling:

If the user presses Ctrl+F2, the program will catch the KeyboardInterrupt exception and execute the code in the except KeyboardInterrupt block. This will print a message indicating that the script has been terminated by the user and, optionally, send a final message to Telegram notifying the script termination.

**Output:**

* + **Data Visualization:**

****

* **X-Axis:** Represents time or timestamps, showing the duration over which, the gas concentration data was recorded.
* **Y-Axis**: Represents the gas concentration values recorded by the sensors.
* **Lines:** Multiple lines on the graph, each representing a different gas sensor’s reading. Each line represents the gas concentration values recorded by a specific sensor at different points in time.
* **Legend:** A legend indicating the colour used for each gas sensor on the graph. It helps identify which line represent which sensor.
* **Trend Analysis:** Observing the graph over time, you may be able to identify trends or patterns in the gas concentration levels recorded by each sensor, such as spikes, fluctuations, or consistent levels.
  + **Leakage Alerts:**



* **Telegram Alerts:** These are notifications sent via the Telegram messaging platform to inform users about the gas concentration readings from different sensors.
* **Gas Concentration Readings:** The messages contain gas concentration readings for each sensor. Each message begins with "SensorX" (where X is the sensor number) to indicate which sensor's data is being reported.
* **Average Value:** The messages include the average gas concentration value for each sensor over a specific period. The average value gives an overall indication of the gas levels recorded by that particular sensor during the monitoring period. If the average value exceeds the predefined threshold of 30, indicating a potential gas leak or elevated gas levels.
* **LEAKAGE DETECTED!!!:** The presence of this text indicates that the average gas concentration value has exceeded a predefined threshold or limit, suggesting a potential gas leak or elevated gas levels. The "LEAKAGE DETECTED!!!" alert warns users about the hazardous condition.
* **Script Terminated by User:** The appearance of this text suggests that the gas concentration monitoring script or system execution was manually terminated or stopped by a user after the detected leakage was repaired or was taken care of.