University Of Messina

Smart Agriculture: IoT Data Collection and Storage using MQTT

Pranav Varanganti | Armando Ruggeri |
07 JULY 2025

Agenda

- 1. Introduction
- 2.Database Structure
- 3. MQTT Architecture
- 4. Data Handling
- 5. Database Integrations & Processing Logic
- 6. Testing & Evaluation
- 7. Conclusion

Introduction

Modern agriculture is increasingly dependent on IoT sensors to monitor critical conditions such as temperature, soil moisture, and soil pH across large farms.
These sensors generate massive volumes of real-time data that need to be processed, stored, and analyzed efficiently to ensure healthy crop growth and maximize yields.

However, without timely data processing, farmers may face:

- Delayed detection of harmful conditions
- Crop damage and yield loss
- Inefficient farm management

This project aims to develop a smart data management system that uses MQTT and Python to handle this big data pipeline, enabling:

- Real-time data collection and processing
- Storage in appropriate databases
- Immediate alerts when sensor readings cross critical thresholds
- In this way, farmers can make data-driven decisions, respond quickly to potential threats, and achieve sustainable and profitable agriculture.

Problem Description

- Modern farms generate massive sensor data like Temperature, Moisture and Soil pH in real time. So, we have to store the large amount of data in suitable database according to its type and we have to do this in real time. Without real-time processing and delayed alerts can damage crops or reduce yield.
- **Challenges**:
  - High volume and high frequency of data.
  - Massive IoT sensor data requires real-time processing
  - Requires instant detection of abnormal conditions.
  - Different data types need different storage systems: structured (SQL), semi-structured (MongoDB), graph relationships (Neo4j).
- **Our Solution**
  - MQTT broker efficiently receives fast sensor data.
  - Python processes it immediately.
  - Data stored where it fits best: SQL for readings, MongoDB for alerts, Neo4j for relationships.

- This ensures the system is scalable, responsive, and tailored to the specific data storage needs of smart agriculture.

Project Objectives

- **Collect IoT sensor data using MQTT**
  - Use an MQTT broker (Eclipse Mosquitto) to receive continuous data streams from multiple sensors deployed in different zones in different farms.
  - Ensure reliable and lightweight data transfer, even with many sensors sending frequent readings.

- **Process data with Python in real time**
  - Use a Python MQTT client (Paho) to subscribe to topics and handle incoming data instantly.
  - Apply thresholds to detect abnormal conditions (e.g., high temperature, low soil moisture).
  - Trigger alerts when sensor readings cross defined limits.

- **Store data in multiple databases according to the type**
  - **MySQL:** Store structured sensor readings with timestamps, sensor IDs, and values.
  - **MongoDB:** Save semi-structured alert messages, including details about alert type(e.g., high temperature, low soil moisture), timestamps, and zones.
  - **Neo4j:** Map relationships between farms, zones, and sensors to visualize the network, and update min/max values of a specific sensor and alert counts for better insights.

Database Structure

- **MySQL database structure**:

  Firstly, we created the database named "**smart_agriculture**" then we created the tables named farms, zones, sensors and readings. The attributes of each tables are

  - farms - farm_id, farm_name, owner_name, contact_email and contact_phone
  - zones - zone_id,farm_id and crop_name
  - sensors - sensor_id, zone_id and sensor_type
  - readings - reading_id, sensor_id, value and timestamp

  We have five farms each farm have four zones and each zone have three sensors to generate the temperature, moisture and soil pH. This generated data will be stored in the readings table.

Database Structure

- **MongoDB database structure**:

  In MongoDB we have the database named "**smart_agriculture**" and then we have the collection named alerts.
- Document structure:

  Each alert document contains the following fields:
  - sensor_id — ID of the sensor that triggered the alert
  - zone_id — ID of the zone where the sensor is located
  - sensor_type — Type of sensor: **Temperature**, **Moisture**, or **Soil pH**
  - alert_type — Indicates whether it is a **high** or **low** condition alert (e.g., high_temperature)
  - value — The sensor reading that triggered the alert
  - unit — Unit of the reading (°C for temperature, % for moisture, pH for soil pH)
  - timestamp — The date and time when the alert was generated


```
_id: ObjectId('6862a9542e997d722474ff69')
sensor_id : "SENSOR_002"
zone_id : "ZONE_01"
sensor_type : "Moisture"
alert_type : "low_moisture"
value : 1.98
unit : "%"
timestamp : 2025-06-30T17:12:18.000+00:00
```

Database Structure

- **Neo4j database structure**:

  In Neo4j we have the database named "**smart_agriculture**" and then we have the nodes Farm, Sensors and Zone. Each node have the attributes

  - Farm - farm_id, farm_name, owner_name
  - Sensors – sensor_id, sensor_type, max_value, max_value_timestamp, min_value and min_value_timestamp
  - Zone – zone_id, crop_name

  Relationships:

  1.HAS_ZONE:
  - Connects **Farm → Zone**
  - Each farm contains four zones.

  2.COLLECTS_DATA_OF:
  - Connects **Sensor → Zone**
  - Attributes:
    - high_alert_count — Number of times the sensor triggered a high alert for that zone
    - low_alert_count — Number of times the sensor triggered a low alert for that zone
    - last_updated — Timestamp of the most recent update for this relationship

Data Flow Diagram

- MQTT Broker -> Python Publisher -> Python subscriber -> Updating Databases

- MQTT Broker : running the downloaded MQTT broker
- Python Publisher : generates the readings for each sensor
- Python Subscriber : receives the generated data
- Updating Databases: stores the received data in the databases

MQTT Server Setup

- Installed Mosquitto broker from the link https://mosquitto.org/download/
- Now we setup the server using python. The python code mqtt_publisher.py will publish the data through mqtt.
- In mqtt_publisher.py first we connect with the mqtt server using the localhost port number 1883 and we setted the topic as “smart_agriculture/sensors”. So, at this topic the server will publish the data of the each sensor we have.
- We know that we have 5 farms, 4 zones in each farm and 3 sensors in each zone. So, I stored the sensors data in list which have each sensor’s data in a dictionary. The sensors data in the dictionary consists sensor_id, sensor_type and zone_id.
- Now we generate the values of each sensor according to its type using the function generate_values which takes sensor_type as input. It generates the temperature between -5 to 50 degree Celsius, moisture between 0 to 100 percentage and soil_pH from 3.5 to 9.0.
- In the main function we will connect to the mqtt server and start the loop with the exception of keyboard interruption in which we will generate the value for each sensor according to it’s type.
- Finally the generated data will be published to the topic “smart_agriculture/sensors” in the Json format.

MQTT Server Setup

- **Json format of the published data :**
  - sensor_id : The Id of the sensor for which reading is generated
  - sensor_type : The type of the sensor for which reading is generated
  - zone_id : The zone id for which the reading is generated
  - value : The value generated
  - timestamp : The time when the value is generated

After running the mosquito broker and the mqtt_publisher.py python file the data will be published like this

```
PS D:\notes\yr_2_semeaster_2\database_mod_B\project> python mqtt_publisher.py
D:\notes\yr_2_semeaster_2\database_mod_B\project\mqtt_publisher.py:31: DeprecationWarning: Callback API version 1 is dep
recated, update to latest version
  client = mqtt.Client()
Published: {'sensor_id': 'SENSOR_001', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_01', 'value': 31.01, 'timestamp':
1751658143}
Published: {'sensor_id': 'SENSOR_002', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_01', 'value': 38.9, 'timestamp': 1751
658143}
Published: {'sensor_id': 'SENSOR_003', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_01', 'value': 5.06, 'timestamp': 17516
58143}
```

Receiving The Data

- The Python script mqtt_subscriber.py receives sensor data published by the MQTT publisher.
- First, we define the on_connect function, which subscribes to the MQTT topic where sensor data is published.
- Next, the on_message function is defined to handle incoming messages. It decodes the JSON payload into a Python dictionary and stores it in the variable data. This function then calls process_sensor_data from handlers.py.
- The process_sensor_data function is responsible for storing the received data into the three databases (MySQL, MongoDB, and Neo4j), which were set up earlier in the project.
- In the main function, an MQTT client instance is created, and the on_connect and on_message callback functions are set.
- The client then connects to the MQTT broker and enters a loop (loop_forever) that keeps it listening for incoming messages indefinitely.
- Whenever a message is received, the on_message callback processes and stores the data accordingly.

Data Handling

- Whenever the data is received by the MQTT Subscriber, it triggers the on_message function. Inside this function, the process_sensor_data function from handlers.py is called, which is responsible for storing the received data into the three different databases.
- To handle the data, first we need to connect with the databases. So, in the “handlers.py” at the staring of the code we connect with MySQL using the host, root, password and the database name.
- Then we will connect with the MongoDB using the host, database name and the collection name.
- Then we will connect with the Neo4j using the host, database name and the password.
- Now we connected with all the three databases to store the generated data.
- We also define threshold values for each sensor type to detect abnormal conditions:
  - **Temperature:** 0°C – 45°C
  - **Moisture:** 6% – 75%
  - **Soil pH:** 5.0 – 7.5
- If the generated sensor value is not in the range of its threshold then it will generate an alert.
- Now lets see when and how the data will be stored.

MySQL Integration

- In MySQL we are storing all the generated sensor values in a table called readings.
- We defined a function called “save_reading_mysql” which takes sensor_id, value and timestamp as input.
- In this function we will we execute a querry which is

```
querry = """INSERT INTO readings (sensor_id, value, timestamp)
VALUES (%s, %s, FROM_UNIXTIME(%s)) """
cursor.execute(querry, (sensor_id, value, timestamp))
```

- This querry will insert all the sensor generated values into the readings table which is in our database smart_agriculture.
- This way, every reading is safely stored in a structured and queryable format.

MongoDB Integration

- In MongoDB we will store all the alerts in a collection called alerts.
- We defined a function called “save_alert_mongodb” which takes sensor_id, zone_id, sensor_type, alert_type, value, timestamp as input.
- In the function first we will determine the units for appropriate sensor_type (e.g., % for moisture).
- Then we will write a piece of code which will add an alert document to the alerts collection.

```
alert_doc = {
    "sensor_id": sensor_id,
    "zone_id": zone_id,
    "sensor_type": sensor_type,
    "alert_type": alert_type,
    "value": value,
    "units": unit,
    "timestamp": datetime.fromtimestamp(timestamp).isoformat()
}
alerts_collection.insert_one(alert_doc)
```

- This above code will insert the data if the generated sensor data is an alert condition.

Neo4j Integration

- We defined a function called “update_mongodb” which takes sensor_id, zone_id, sensor_type, value, timestamp, alert_type1 as input.
- In Neo4j we have to do two things
  - first to create (if not existed) or update the attributes max_value, max_value_timestamp, min_value and min_value_timestamp of the node sensors.
  - Second to create(if not existed) or update the attributes high_alert_count, low_alert_count and last_updated of the relationship COLLECTS_DATA_OF between sensors and zone.
- Updating sensor attributes:
  - To update the attributes first we need to match the sensor using sensor_id and then we have to check the attribute is null or not. If it is null we have set the attribute with value generated by the sensor. If its not null then we have to check the condition and update the attribute’s value.
  - This will be done by the code given below

```
session.run("""
    MATCH (s:Sensors {sensor_id: $sensor_id})
    SET
        s.min_value = CASE WHEN s.min_value IS NULL OR $value < s.min_value THEN $value ELSE s.min_value END,
        s.min_value_timestamp = CASE WHEN s.min_value IS NULL OR $value < s.min_value THEN datetime($timestamp) ELSE s.min_value_timestamp END,
        s.max_value = CASE WHEN s.max_value IS NULL OR $value > s.max_value THEN $value ELSE s.max_value END,
        s.max_value_timestamp = CASE WHEN s.max_value IS NULL OR $value > s.max_value THEN datetime($timestamp) ELSE s.max_value_timestamp END
""", sensor_id=sensor_id, value=value, timestamp=datetime.fromtimestamp(timestamp).isoformat())
```

Neo4j Integration

- Updating relationship attributes:
  - First we need to check the generated sensor value is an alert or not. If its an alert we will check if it's a high alert or low alert (e.g., high_temperature, low_moisture etc).
- If its a high alert condition first we need to connect the sensors and zone with relationship COLLECTS_DATA_OF. Then if the attribute high_alert_count is null we need to set its value as 1. If it exists we have to set its value as high_alert_count+1. At last we need to set the latest updated timestamp.
- If its a low alert condition and if the attribute low_alert_count is null we need to set its value as 1. If it exists we have to set its value as low_alert_count+1. At last we need to set the latest updated timestamp.
- After doing this all the sensor attributes and relationship attributes we will be updated.
- This will be done by the code given in the next page

Neo4j Integration

- Code to update sensor attributes and relationship attributes :

```
if alert_type:
    if (alert_type == "high_temperature" or alert_type == "high_moisture" or alert_type == "high_soil_ph"):
        querry ="""
            MATCH (s:Sensors {sensor_id: $sensor_id})-[r:COLLECTS_DATA_OF]->(z:Zone {zone_id: $zone_id})

            SET
                r.high_alert_count = CASE WHEN r.high_alert_count IS NULL  THEN 1 ELSE r.high_alert_count+1 END,
                r.last_updated = datetime($timestamp)
        """
    elif (alert_type == "low_temperature" or alert_type == "low_moisture" or alert_type == "low_soil_ph"):
        querry ="""
            MATCH (s:Sensors {sensor_id: $sensor_id})-[r:COLLECTS_DATA_OF]->(z:Zone {zone_id: $zone_id})

            SET
                r.low_alert_count = CASE WHEN r.low_alert_count IS NULL  THEN 1 ELSE r.low_alert_count+1 END,
                r.last_updated = datetime($timestamp)
        """
    session.run(querry, sensor_id=sensor_id, zone_id=zone_id, alert_type=alert_type, timestamp=datetime.fromtimestamp(timestamp).isoformat())
```

Data Handling

- Now last step in our handlers.py is to integrate all the previously defined functions in a function called process_sensor_data which takes the sensor generated data(as a dictionary) as input.
- Now we use some local variable to store the sensor_id,sensor_type etc.
- Then we call the save_reading_mysql function with sensor_id, value and timestamp as inputs. This will save the received data to readings table in the database.
- Now we have to determine if the generated value is an alert or not. If its an alert what is the type high or low alert. This will be necessary for MongoDB and Neo4j. We will do this using the threshold values we defined before and can be done with this code.

```
alert_type = None
if sensor_type in thresholds:
    if value > thresholds[sensor_type]["high"]:
        alert_type = f"high_{sensor_type.lower()}"
    elif value < thresholds[sensor_type]["low"]:
        alert_type = f"low_{sensor_type.lower()}"
```

Data Handling

- If alert type is not null, we will call the save_alert_mongodb with necessary input values. It will insert the alert document to the alerts collection.
- At last we will call the update_neo4j function with necessary input values to update the attributes of Sensors and the relationship COLLECTS_DATA_OF.
- This is whole architecture of our project and next we will test the project by running it.

Testing & Simulations

- First we need to run the mqtt mosquitto broker.
- Then run the mqtt_subscriber.py in the terminal.

```
PS D:\Notes\yr_2_semeaster_2\database_mod_B\project> python mqtt_subscriber.py
D:\Notes\yr_2_semeaster_2\database_mod_B\project\mqtt_subscriber.py:19: DeprecationWarning: Callback API version 1 is de
precated, update to latest version
  client = mqtt.Client()
Connected with result code 0
Received: {'sensor_id': 'SENSOR_001', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_01', 'value': 22.09, 'timestamp': 1
751815704}
Received: {'sensor_id': 'SENSOR_002', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_01', 'value': 90.19, 'timestamp': 1751
815705}
Received: {'sensor_id': 'SENSOR_003', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_01', 'value': 7.08, 'timestamp': 175181
5705}
Received: {'sensor_id': 'SENSOR_004', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_02', 'value': 45.6, 'timestamp': 17
51815705}
Received: {'sensor_id': 'SENSOR_005', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_02', 'value': 99.37, 'timestamp': 1751
815705}
Received: {'sensor_id': 'SENSOR_006', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_02', 'value': 8.95, 'timestamp': 175181
5705}
Received: {'sensor_id': 'SENSOR_007', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_03', 'value': -0.41, 'timestamp': 1
751815705}
Received: {'sensor_id': 'SENSOR_008', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_03', 'value': 78.53, 'timestamp': 1751
815705}
Received: {'sensor_id': 'SENSOR_009', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_03', 'value': 8.36, 'timestamp': 175181
5705}
```

Testing & Simulations

- Then run the mqtt_publisher.py in another tab of the terminal.

```
PS D:\Notes\yr_2_semeaster_2\database_mod_B\project> python mqtt_publisher.py
D:\Notes\yr_2_semeaster_2\database_mod_B\project\mqtt_publisher.py:31: DeprecationWarning: Callback API version 1 is dep
recated, update to latest version
  client = mqtt.Client()
Published: {'sensor_id': 'SENSOR_001', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_01', 'value': 22.09, 'timestamp':
1751815704}
Published: {'sensor_id': 'SENSOR_002', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_01', 'value': 90.19, 'timestamp': 175
1815705}
Published: {'sensor_id': 'SENSOR_003', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_01', 'value': 7.08, 'timestamp': 17518
15705}
Published: {'sensor_id': 'SENSOR_004', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_02', 'value': 45.6, 'timestamp': 1
751815705}
Published: {'sensor_id': 'SENSOR_005', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_02', 'value': 99.37, 'timestamp': 175
1815705}
Published: {'sensor_id': 'SENSOR_006', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_02', 'value': 8.95, 'timestamp': 17518
15705}
Published: {'sensor_id': 'SENSOR_007', 'sensor_type': 'Temperature', 'zone_id': 'ZONE_03', 'value': -0.41, 'timestamp':
1751815705}
Published: {'sensor_id': 'SENSOR_008', 'sensor_type': 'Moisture', 'zone_id': 'ZONE_03', 'value': 78.53, 'timestamp': 175
1815705}
Published: {'sensor_id': 'SENSOR_009', 'sensor_type': 'Soil_pH', 'zone_id': 'ZONE_03', 'value': 8.36, 'timestamp': 17518
15705}
```

Testing & Simulations

- Data stored in MySQL :

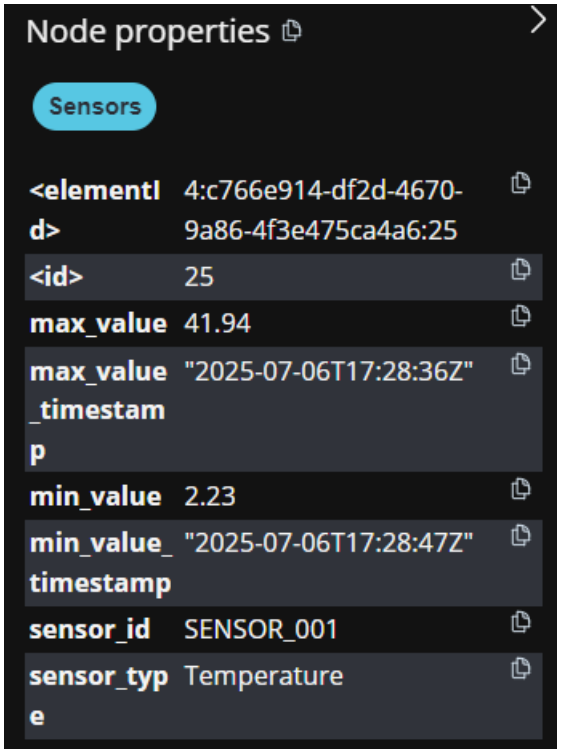
reading_id	sensor_id	value	timestamp
848	SENSOR_001	22.09	2025-07-06 17:28:24
849	SENSOR_002	90.19	2025-07-06 17:28:25
850	SENSOR_003	7.08	2025-07-06 17:28:25
851	SENSOR_004	45.6	2025-07-06 17:28:25
852	SENSOR_005	99.37	2025-07-06 17:28:25
853	SENSOR_006	8.95	2025-07-06 17:28:25
854	SENSOR_007	-0.41	2025-07-06 17:28:25
855	SENSOR_008	78.53	2025-07-06 17:28:25
856	SENSOR_009	8.36	2025-07-06 17:28:25

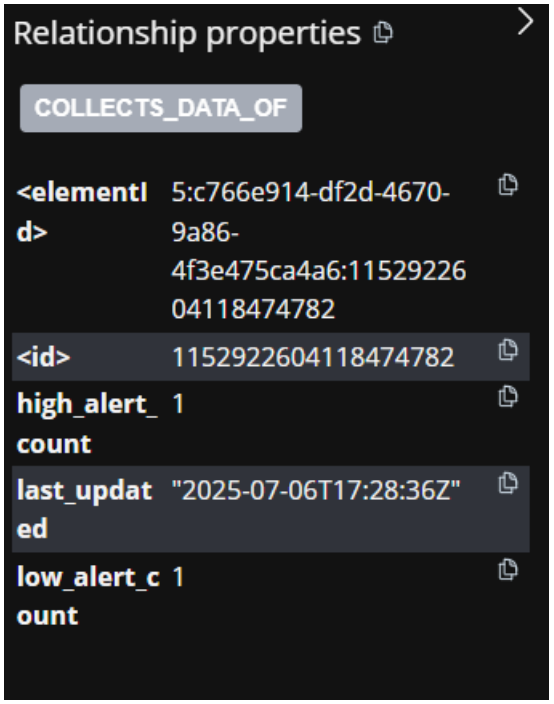
- Data stored in MongoDB:

```
> db.alerts.find({},{sensor_id:1,alert_type:1, _id:0})
< {
    sensor_id: 'SENSOR_002',
    alert_type: 'high_moisture'
  }
  {
    sensor_id: 'SENSOR_004',
    alert_type: 'high_temperature'
  }
  {
    sensor_id: 'SENSOR_005',
    alert_type: 'high_moisture'
  }
  {
    sensor_id: 'SENSOR_006',
    alert_type: 'high_soil_ph'
  }
```

Testing & Simulations

- Data updated in Neo4j :





Conclusion

- In this project, we successfully designed and implemented a Smart Agriculture system that leverages IoT sensors, MQTT messaging, and multi-database storage to monitor critical environmental parameters such as temperature, soil moisture, and soil pH across multiple farms and zones.
- By integrating:
  - **MQTT** for real-time, and efficient data transfer between sensors and the processing pipeline.
  - **MySQL** to store all sensor readings in a structured and query-friendly format.
  - **MongoDB** to maintain rich, flexible JSON-like documents for detailed alert information.
  - **Neo4j** to model complex relationships and generate valuable insights through graph-based queries, including min/max tracking and alert relationships.
- we demonstrated how combining multiple data storage technologies enhances performance, flexibility, and analysis capabilities for a modern precision agriculture system.
  - The project also shows how:
  - Real-time data ingestion can be handled reliably using MQTT brokers and Python clients.
  - Alerts can be automatically generated when sensor values exceed defined thresholds.
  - All the collected data can be visualized and queried efficiently for improved decision-making and farm management.