Bachelor's thesis

Information and Communications Technology

2024

Sérgio Apolinário da Costa

Smart Agriculture: Implementing IoT for Greenhouse Monitoring

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 67 pages

Sérgio Apolinário da Costa

Smart Agriculture: Implementing IoT for Greenhouse Monitoring

Modern agriculture demands sustainable and efficient farming methods to tackle climate change and resource scarcity. Greenhouse farming, in particular, faces inefficiencies in resource management and production. This thesis addresses these issues by developing an IoT-based system to monitor key environmental parameters in greenhouses. The goal was to use sensor data to optimise resource use and promote healthier plant growth.

To achieve this, a system was created using ESP32 devices to collect data from the greenhouse environment. These devices send the data to a central computer program that organises and stores the information in a database. This setup is flexible, allowing easy integration of new devices.

The system was tested over 45 days in a greenhouse, operating reliably without missing data or needing maintenance. The results demonstrated that the system met the necessary performance standards, showing its potential to improve greenhouse farming efficiency and sustainability significantly. This project provides a valuable solution for optimising resource utilisation and enhancing plant growth in controlled environments, addressing critical agricultural challenges.

Keywords:

ESP32, MQTT, AWS, BME680, BH1750, MongoDB, Angular, Greenhouse

Contents

Figures

Tables

List of abbreviations

ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
ARN	Amazon Resource Name
AWS	Amazon Web Service
BR	Basic Rate
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
EDR	Enhanced Data Rate
ECC	Elliptic Curve Cryptography
FTP	File Transfer Protocol
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
I2C	Inter-Integrated Circuit
MOX	Metal-oxide
MQTT	Message Queuing Telemetry Transport

PWM	Pulse Width Modulation
RAM	Random-Access Memory
RSA	Rivest-Shamir-Adleman
SDIO	Secure Digital Input Output
SHA	Secure Hash Algorithm
SoC	Systems on a Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SSID	Service Set Identifier
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UART	Universal Asynchronous Receiver / Transmitter
UI	User Interface
USB	Universal Serial Bus
VOC	Volatile Organic Compounds
VPN	Virtual Private Network
WPA	Wi-Fi Protected Access
WPS	Wi-Fi Protected Setup

1 Introduction

Finding better ways to farm sustainably and efficiently is crucial in today's farming. Challenges such as climate change and resource scarcity emphasise the need for innovative solutions that enhance productivity and protect the environment. Within this context, the conventional methods employed in greenhouse farming present inefficiencies in resource management and production maintenance.

For example, the approach to watering plants in some greenhouses lacks optimisation, leading to a waste of water resources. Traditional practices, such as watering crops for fixed durations, overlook critical factors such as soil moisture and temperature, which directly influence watering requirements. This mistake highlights the urgency of reevaluating agricultural practices to align with today’s environmental and resource challenges.

There has been a notable increase in studies in the past few years, particularly in Internet of Things (IoT) solutions dedicated to enhancing greenhouse farming practices. These studies underscore the growing interest and recognition of technology's significance in increasing agricultural sustainability and productivity.

This thesis aims to solve the challenges by developing a system integrating IoT technology to monitor critical environmental parameters within greenhouses. The primary objective is to utilise data collected from sensors deployed within the greenhouse environment to provide valuable insights for optimising resource utilisation, promoting healthier plant growth, and mitigating issues such as pest infestation and crop diseases.

This thesis project covers hardware and software components, including sensor deployment, data collection, analysis, and the development of user-friendly interfaces for accessing and interpreting the gathered data.

This thesis is structured into eight chapters, each describing the concepts used in this project.

The first chapter provides an introduction to the thesis's main idea, elucidating the complexity of the problem while giving the thesis overview.

Subsequently, the second chapter employs the MoSCoW prioritisation method to list all the project requirements.

The third chapter delineates the various systems architectures implemented on the journey to the final solution. It describes the theory behind each implementation, presents the functionality, and, in the end, explains why it did not emerge as the definitive solution for this endeavour.

In the fourth chapter, all the hardware needed for this project is listed, followed by a brief introduction, the specifications, and, if necessary, the pinout descriptions.

The fifth chapter catalogues all the software employed to make this project feasible, explains its usage, and provides all the software versions used.

The sixth chapter delves into the complexity of the implementation, providing a comprehensive guide to building a system capable of operationalising the project objectives.

The seventh chapter is dedicated to testing methodologies and the subsequent findings. This chapter also offers insights into the efficacy of the implemented solution.

Finally, the eighth chapter concludes by summarising the principal findings of the research and outlining the path for future developments.

2 Requirements

This chapter outlines all the requirements using the MoSCoW method. This is a renowned prioritisation technique for managing project requirements usually used by companies. It categorises requirements into four groups: must have, should have, could have, and will not have.

Must-have requirements are essential for the project, release, or product. They represent a non-negotiable need, a mandatory task that, if not completed, the project will not work or become useless.

Should have is the second category. These requirements are essential but not as critical as must-have requirements. They are necessary for the project's success but can be prioritised slightly lower.

Could-have requirements are less critical than should-have requirements. They are optional and have less impact on the project outcome if omitted.

Will not have list all the requirements not planned for implementation in the current phase. [1]

The Table 1 is a summary of the requirements for this project categorised according to the methodology described.

Table 1. MoSCoW priority table.

Must-have	Should have	Could have	Will not have
Robust Hardware architecture	Greenhouse integration	Interface for data visualisation	Autonomous greenhouse
Data logging	Data storage	Data analysis	
Server-ESP32 Communication			

2.1 Hardware architecture

Developing a robust hardware architecture is vital for the success of this project. If there are any flaws in this part, it can cause problems throughout the entire project. A robust hardware architecture should manage various challenges without breaking. It should tolerate hardware faults like power loss and be resilient against environmental factors like extreme temperatures or vibrations. Essentially, it is about ensuring that the hardware can perform reliably, minimising the risk of failures that could damage the entire project. The architecture will be provided in subsequent sections of this document.

2.2 Data logging

Data logging gathers sensor data and displays it on the console. This is crucial, and it serves as a diagnostic tool. If a sensor is improperly connected or the output reveals unexpected values, it provides valuable feedback. Consequently, this facilitates the identification of architectural issues.

2.3 Server-ESP32 Communication

Communication between the server and ESP32 devices is a crucial prerequisite. Establishing this connection enables remote access, such as gathering sensor data. This facilitates bidirectional communication, allowing ESP32 devices to send messages to the server and vice versa.

2.4 Greenhouse integration

It is crucial to store the hardware in the greenhouse to validate the sensors functionality and the data collection process. Testing for an extended period within the greenhouse environment ensures the correct system’s performance and can detect any potential issues.

2.5 Data storage

Using pre-existing data empowers to undertake research. Using data storage is essential for conducting studies, and having a trajectory of plants across time can be helpful.

2.6 Interface for data visualisation

Designing an interface for data visualisation enhances accessibility, enabling users to interpret data without effort instead of struggling with database entries. This avoids the need for users to delve into database functionality. Furthermore, this establishes an additional layer of security by limiting database access only to the server and the interface so that there is no necessity to distribute database credentials to all users for accessing the database.

2.7 Data analysis

Analysing data is optional, but it can undoubtedly facilitate accessing the collected data and provide a more comprehensive and readable interpretation of the data.

2.8 Autonomous greenhouse

Creating an autonomous greenhouse prototype is a compelling concept with the potential to improve agricultural practices. Integrating sensors and actuators would enable the greenhouse to respond dynamically to environmental data. For example, in response to high temperatures, the system could automatically open windows for ventilation, or if the soil moisture levels drop below a certain threshold, it could activate irrigation systems to ensure optimal plant growth. Implementing such a system would enhance efficiency and productivity.

3 System Architecture

Several prerequisites were considered to build the system architecture. The physical distance between the greenhouse, where the hardware system operates, and the server was significant. Additionally, the hardware required internet access, necessitating working on different networks. The system should be prepared to add new ESP32 devices and their sensors, minimising the effort required to integrate them. Moreover, the system needs to follow the best IoT practices. Furthermore, the communication between ESP32 and the server would be one-way, demanding preparation for this scenario.

This section outlines all the implementations and explains why certain ones were not included in the final version.

3.1 First implementation

A straightforward implementation is setting up the ESP32 devices to directly transmit the gathered data to the database, as represented in the Figure 1. In essence, the code deployed to the ESP32 devices can be set up to transmit data directly to a database.

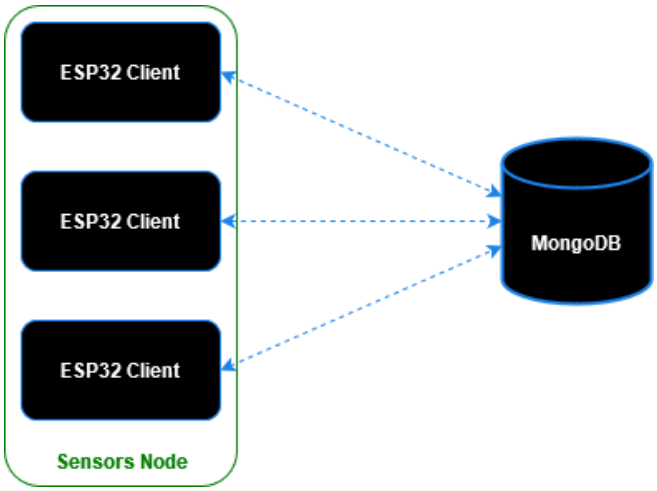


Figure 1. System schematic for the first implementation.

While this solution is very straightforward to execute, it overlooks essential prerequisites.

Exposing the database directly to the internet for ESP32 device connection increases security risks. Without robust security measures such as authentication and encryption, the database becomes vulnerable to unauthorised access and attacks. Additionally, depending on the data volume transmitted from ESP32 devices and the amount of traffic, there is a risk of increasing loading time and potential performance issues or even provoking database downtime during peak usage.

Removing the server escalates the data integrity risk and raises concerns about potential data loss or corruption if messages fail to reach the database. This approach also complicates the management of database scalability and performance as the system grows.

The primary issue relies on the project's adaptability. If changes are required to restructure parts of the system in the future, all the work will need to be redone, and alternative implementations will need to be considered. With this approach, it is impossible to establish bidirectional communication or even send data to the ESP32 devices.

3.2 Second implementation

This implementation consists of a series of sensor nodes with multiple ESP32 devices gathering data from respective sensors. On the other hand, a Raspberry Pi is used as a server. The ESP32 device transmits data in a JSON message to the Raspberry Pi using the MQTT protocol. The MQTT protocol is where a client, in this case, an ESP32 device, publishes data to a broker; in this example, mosquitto is used. The broker will publish the data to any client subscribed to that topic. A topic is essentially a way to categorise the data sent; for example, this project could use a topic such as ’/greenhouse/smart1’, where ’smart1’ is used as the ESP32 ID. [2]

In addition to utilising Mosquitto as the MQTT message broker service, this implementation incorporates Node-RED. Node-RED subscribes to desired topics, and when receives data by mosquitto, Node-RED is notified and subsequently processes the JSON message and then publishes it to the database.

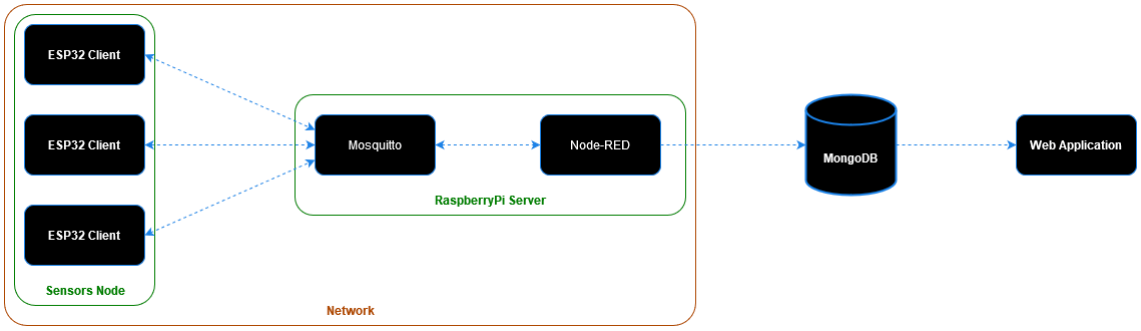


Figure 2. System schematic for the second implementation.

While this setup seems efficient, it is missing a prerequisite: the network. The developed implementation works only on local networks. To solve this problem, mosquito could be configured to accept external network clients, potentially turn off the Raspberry Pi firewall, and configure port forwarding on the router to accept MQTT requests. However, this option was discarded since access to the router configuration was denied, and implementing these changes would significantly compromise security by exposing the system to potential attackers.

3.3 Third implementation

This implementation solves the challenge of accessing different networks and uses much of the previous scheme functionality. Utilising Ngrok on the Raspberry Pi provides a straightforward solution for accessing the server across networks. Ngrok is a cross-platform application that creates secure tunnels to localhost machine, allowing developers to expose their local development server to the Internet with minimal effort. [3].

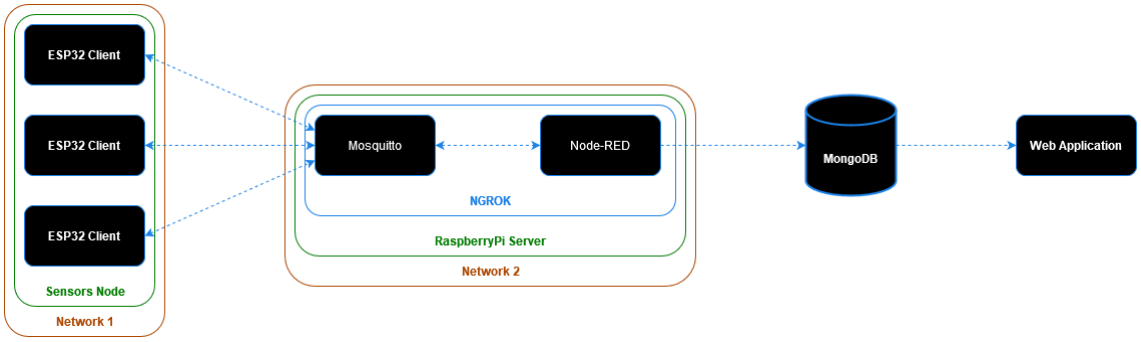


Figure 3. System schematic for the third implementation.

To use Ngrok, there is the need to install it and authenticate it. The authentication token can be obtained from the Ngrok webpage after login. The token is then stored in the default configuration file for future use.

Figure 4. Ngrok authentication token.

Tunnels can be created by executing the specified command. To initiate a tunnel for MQTT requests, execute the command '**ngrok tcp 1883**', which results in the following tunnel configuration. [4].

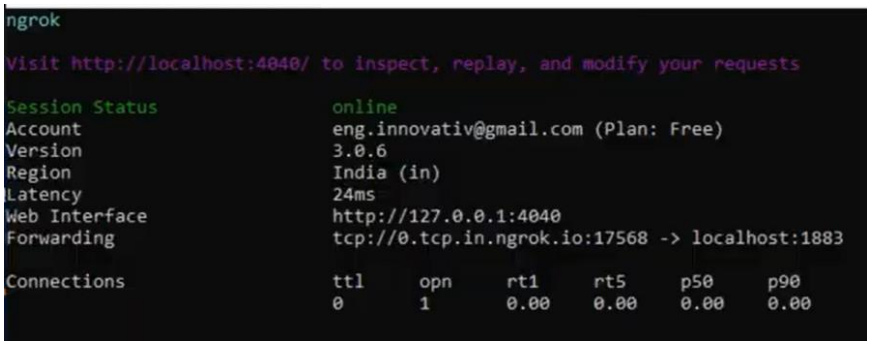


Figure 5. Ngrok tunnel configuration

The host IP and port number are necessary to perform an MQTT request to the server. Taking the image above as an example, the host IP would be ’0.tcp.in.ngrok.io’, and the port number would be ’17568’ [4].

Ngrok has established a tunnel to the local server, enabling external devices to interact. This implementation facilitates the transmission and reception of MQTT messages from any location, even when the devices are in different networks relative to the Raspberry Pi.

Although Ngrok provides a way to expose local services to the internet, it presents security risks despite encrypting traffic. When transmitting data, caution is necessary to prevent the exposure of sensitive information. Additionally, the Ngrok free tier has limitations on usage, such as concurrent connections and tunnel duration, potentially leading to rate limit issues. Furthermore, while Ngrok is suitable for development and testing, it may not offer the scalability and reliability required for production environments when dealing with high traffic volumes or new client additions.

3.4 Fourth implementation

Implementing a VPN at each network point instead of relying on Ngrok is an alternative. Selecting a suitable VPN provider, such as Tailscale, is essential to fulfil this requirement.

Setting up the network on the Raspberry Pi is more accessible; simply install the Tailscale application and initiate it, and the network will be up and running.

When dealing with ESP32 devices, the setup process tends to be more time-consuming. This is because ESP32 devices do not support the installation of external software. In such cases, setting up a ’subnet router’ becomes necessary. The subnet routers act as a gateway, relaying traffic from the Tailscale network onto the physical subnet. Another device is required to configure this subnet router: a machine capable of running Ubuntu or another

operating system or a router specifically designed to support at least port forwarding and VPN functionalities. [5]

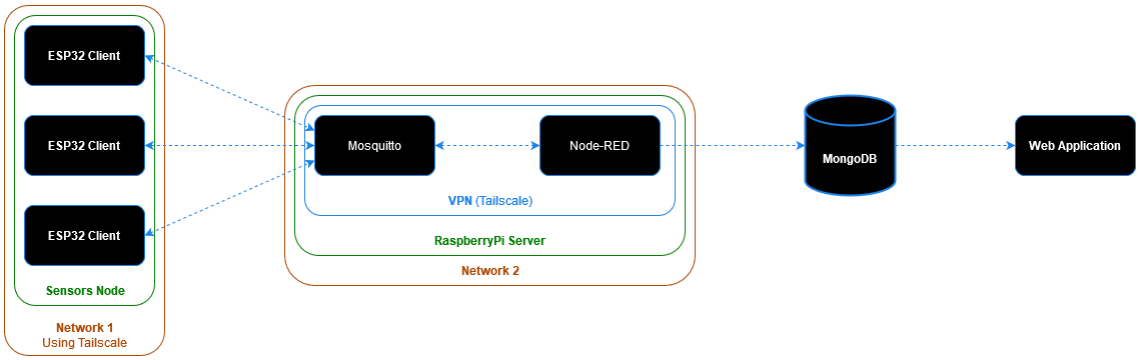


Figure 6. System schematic for the fourth implementation.

This solution was not implemented in the final solution because a router specific to the greenhouse would have needed to be bought, which would have required power to charge the router. Another issue with this solution was that if new ESP32 devices need to be added, Tailscale is not scalable.

This solution was not implemented as the final one mainly because it required purchasing a dedicated greenhouse router and power. Additionally, scalability issues appeared with adding new ESP32 devices using Tailscale.

3.5 Final implementation

This section presents the final architecture implemented, fulfilling all the initial prerequisites.

The ESP32 devices will be in a sensor node, as with the previous implementations, each with access to a Wi-Fi hotspot. The server will utilise Node-RED to subscribe to the MQTT broker, translate the incoming messages, and then publish the data in the database. This system uses Amazon Web Services (AWS), which offers Internet of Things (IoT) services and solutions to connect and manage billions of devices. AWS facilitates collecting, storing, and analysing IoT data across various domains. Specifically, AWS IoT Core

supports MQTT requests, enabling the definition of clients representing the ESP32 devices and functioning as a broker for the server. [6]

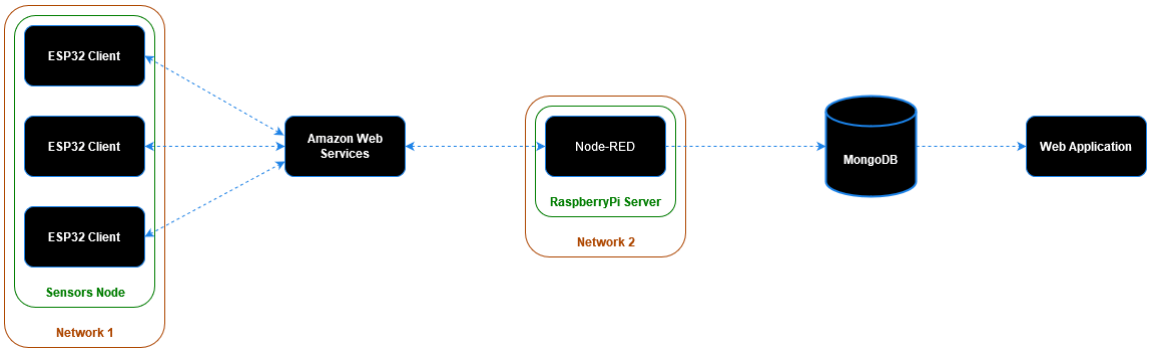


Figure 7. System schematic for the final implementation.

Designing this architecture enhances flexibility and scalability for integrating new devices. Within this solution, adding ESP32 devices across different networks becomes easily achievable. Furthermore, AWS IoT Core offers robust security measures, device authentication, encryption, and access control. AWS is more flexible than the other implementations, supporting various communication protocols, including MQTT, HTTP, and WebSockets. Additionally, AWS integrates with other AWS services such as AWS Lambda, Amazon S3, Amazon DynamoDB, Amazon Kinesis, and others.

4 Hardware

This section outlines all the hardware employed in the execution of this project.

4.1 ESP32

ESP32 is a series of low-power systems on a chip microcontroller (SoC) that provides connectivity to Wi-Fi and Bluetooth (in some models). These devices were developed by Espressif Systems, which has a range of affordable modules. The ESP32 microcontrollers are the successors of the ESP8266, which are more advanced and have more features.

Since the initial ESP32 model was released, various variants have been introduced and announced, forming the ESP32 microcontroller family. While these chips vary in CPU specifications and functionalities, they are unified by a standard software development kit and broad code compatibility.

In this project, the **ESP32 NodeMCU development board** was used. This variant has ultra-low power consumption, a lightweight design, and a compact size. This board offers versatility with PWM, I2C, SPI, UART, 1-wire, and 1-analog pin functionalities. [7]

Figure 8. ESP32 NodeMCU development board. [7]

4.1.1 Specifications

Key specifications of this microcontroller include:

- **Frequency range:** 2.4 – 2.5 GHz,
- **Bluetooth protocol:** Bluetooth v4.2 BR/EDR,
- **Memory:** 4MB Flash, 520KB SRAM,
- **Wireless form:** On-board PCB Antenna,
- **IO Capability:** UART, I2C, SPI, PWM, SDIO, GPIO, ADC, DAC,
- **Electrical characteristics:** 3.3V operated, 15mA output current per GPIO pin and 80mA average working current,
- **Operating temperature:** -40ºC to 125ºC,
- **Security type:** WPA / WPA2 / WPA2-Enterprise / WPS,
- **Encryption type:** AES / RSA / ECC / SHA. [7]

4.1.2 Pinout

This microcontroller has 38 pins, including three ground pins, a single 3.3V pin, and a single 5V pin. Below is provided an image with a comprehensive pinout diagram:

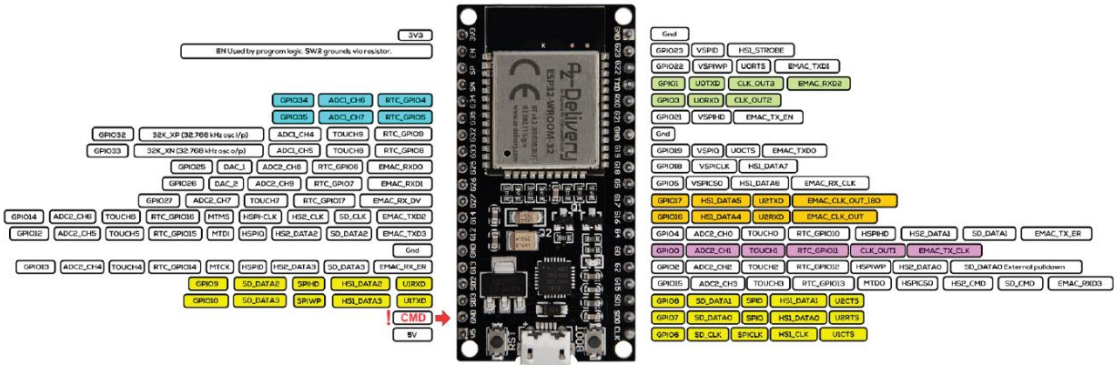


Figure 9. ESP32 NodeMCU development board pinout. [7]

4.2 Raspberry Pi

The Raspberry Pi Foundation created the Raspberry Pi in collaboration with Broadcom. It is a collection of compact single-board computers. This tiny board has components common in ordinary computers, including a CPU, GPU, memory, and a WI-FI module, all boosted by an operating system. The board hosts a 40-pin GPIO interface, facilitating the integration of sensors and actuators. [8]

The Raspberry Pi 4 Model B was selected as the primary hardware platform in this thesis project. The integrated Wi-Fi module, a cost-effective and user-friendly solution, drove the decision.

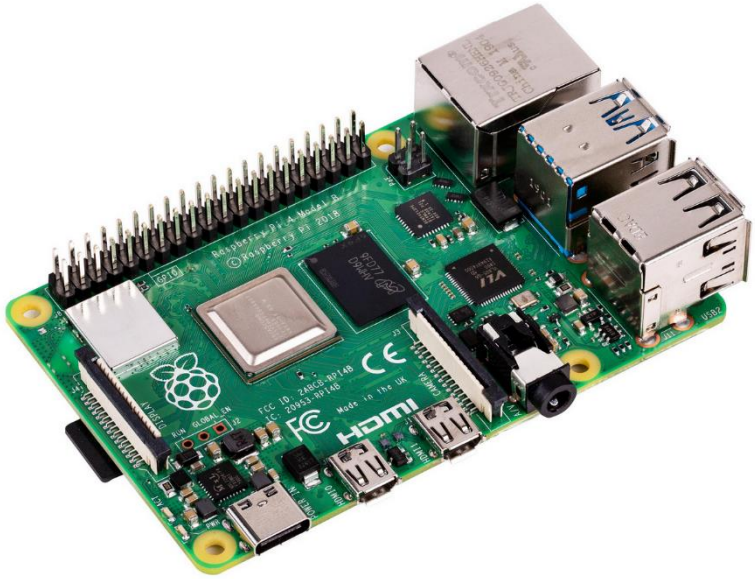


Figure 10. Raspberry Pi 4 Model B.

This board has a more powerful processor, features a USB-C power supply, supports dual 4K displays via two micro-HDMI ports, and includes USB 2 and USB 3 ports and an Ethernet port. Additionally, users have the flexibility to select the desired RAM capacity. This project utilises the 4 GB variant, though options range from 1 GB to 2 GB and 8 GB.

4.3 BME680 Sensor

The BME680 is a multifaceted digital sensor that combines gas, pressure, humidity, and temperature sensing capabilities within its compact design. The gas sensor can detect various gases, including volatile organic compounds (VOC).

This sensor contains a Metal-oxide (MOX) sensor designed to detect VOCs present in the atmosphere. It comprises a metal-oxide surface, a sensing chip for conductivity measurements, and a heater. MOX sensors operate by absorbing oxygen molecules onto their sensitive layers.

In the beginning, the BME680 furnishes resistance values, which then vary in direct relation to variations in VOC concentrations. Elevated levels of VOCs align with decreased resistance, while lower concentrations result in increased resistance levels. [9]

4.3.1 Specifications

The key specifications of this sensor include:

- **Operating voltage:** 1.7 V – 3.6 V,
- **Operating temperature:** -40ºC – 85ºC
- **Communication interface:** I2C / SPI
- **Temperature sensor accuracy:** +/- 1ºC
- **Humidity sensor accuracy:** +/- 3%
- **Pressure sensor accuracy:** +/- 1hPa
- **Temperature sensor operation range:** -40ºC to 85ºC
- **Humidity sensor operation range:** 0% to 100%
- **Pressure sensor operation range:** 300hPa to 1100hPa [9]

4.3.2 Pinout

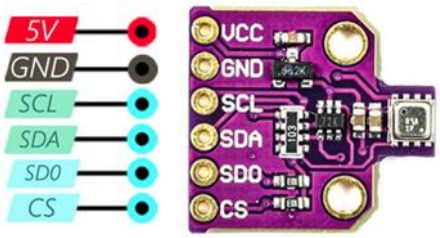


Figure 11. BME680 pinout.

The BME680 sensor features a set of six pins, each playing a vital role in its operation:

- **VCC:** power positive pin, supplies the necessary electrical energy to operate the sensor,
- **GND:** power negative pin, ensures proper grounding for the sensor, contributing to its stable performance,
- **SCL:** serves a crucial role in facilitating communication via the I2C/SPI protocol by acting as the clock line,
- **SDA:** essential for data transmission in I2C/SPI communication, SDA functions as the data line, enabling the exchange of information,
- **SDO:** operates in SPI mode; this pin serves as the data output line, transmitting sensor data to connected devices,
- **CS:** operates in SPI mode, and this pin enables the signal for SPI devices, facilitating communication between the sensor and other peripherals.

4.4 Capacitive Soil Moisture Sensor

The analog capacitive soil moisture sensor measures soil moisture levels by capacitive sensing rather than resistive sensing, like other types of moisture sensors. It is made of a corrosion-resistant material, giving it a long service life.

The Capacitive Soil Moisture Sensor measures changes in capacitance as it is inserted into the soil. The sensor's capacitance alters based on the soil's moisture levels, affecting its charging time. [10]

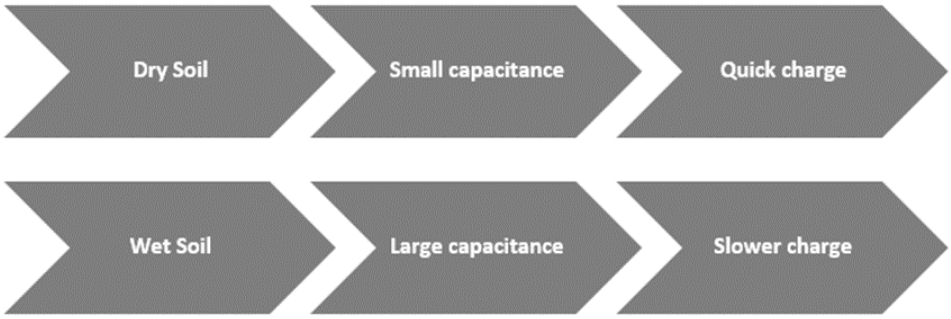


Figure 12. Sensor capacitance VS soil moisture level.

The sensor uses a circuit with a 555 timer configured as an oscillator. Its square waves are fed into an RC integrator, where the soil probe is the capacitor. The integrator produces a triangular wave, rectified and smoothed to create a DC output proportional to soil moisture. Dry soil charges the capacitor quickly, resulting in a higher output voltage, while wet soil charges it slowly, leading to a lower output voltage. [10]

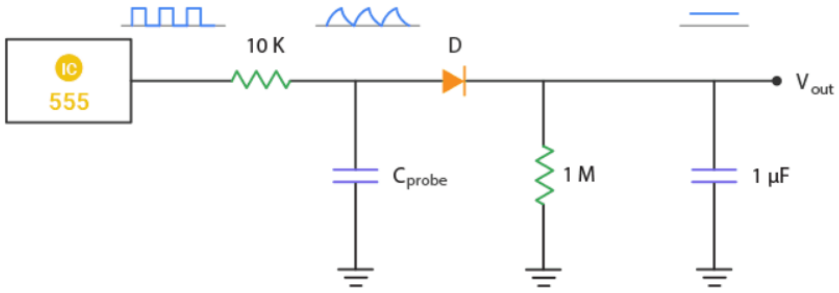


Figure 13. Capacitive soil moisture sensor diagram. [10]

4.4.1 Specifications

The key specifications of this sensor include:

- **Operating voltage:** 3.3V to 5.5V,
- **Operating current:** < 5mA
- **Output voltage at 5V:** 1.5V to 3V.

4.4.2 Pinout

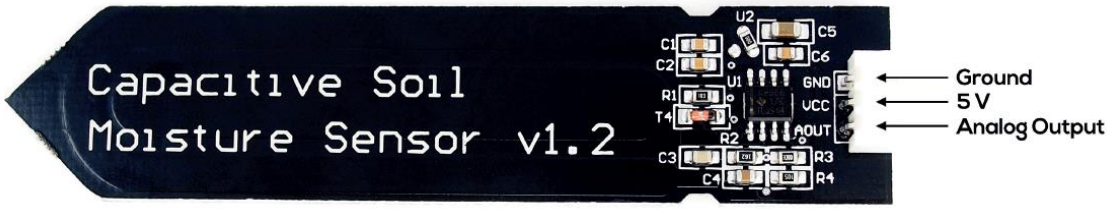


Figure 14. Capacitive soil moisture pinout

The capacitive soil moisture sensor features a set of three pins, each playing a vital role in its operation:

- **VCC:** power positive pin, supplies the necessary electrical energy to operate the sensor,
- **GND:** power negative pin, ensures proper grounding for the sensor, contributing to its stable performance,
- **AOUT:** gives an analogue output that is proportional to the amount of moisture in the soil.

4.5 BH1750 Sensor

The BH1750 is a 16-bit ambient light sensor that communicates via I2C protocol. It outputs luminosity measurements in lux. It offers precise luminosity measurements in lux units. With a wide range, it can detect light levels as low as 0 lux and as high as 65535 lux. [11]

This sensor is available in different breakout board formats.

The BH1750 serves multiple purposes, such as determining day or night conditions, dynamically adjusting LED brightness based on ambient light levels, detecting the status of illuminated LEDs and much more. [11]

4.5.1 Specifications

The key specifications of this sensor include:

- **Operating voltage:** 3.3V to 5V,
- **Operating temperature:** -40ºC – 85ºC,
- **Accuracy:** +/- 20%,
- **Interface:** I2C,
- **Range:** 0 – 65535 lux [11]

4.5.2 Pinout

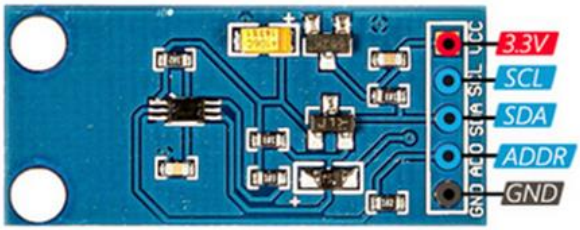


Figure 15. BH1750 pinout.

The BH1750 sensor features a set of five pins, each playing a vital role in its operation:

- **VCC:** power positive pin, supplies the necessary electrical energy to operate the sensor,
- **GND:** power negative pin, ensures proper grounding for the sensor, contributing to its stable performance,
- **SCL:** this is the serial clock pin used for synchronising data transmission between the BH1750 sensor and the microcontroller,
- **DAT:** is for Data, where the sensor sends/receives data to and from the microcontroller,
- **ADDR:** sets the sensor's I2C address, allowing multiple sensors to be connected on the same bus without conflicts**.**

4.6 Mobile phone

Another project requirement is a mobile phone or a router to provide internet connectivity. This component enables communication between the ESP32 devices and the server. In this context, an old mobile phone was used as a hotspot, providing internet access to the ESP32 devices.

4.7 Breadboard and jumper wires

For the implementation, a breadboard and jumper wires were essential to connect the microcontroller to the sensors efficiently and without soldering. The breadboard provided a flexible and reusable platform for building and testing the circuit, allowing quick modifications and troubleshooting. Jumper wires established reliable connections between the various components, ensuring both the integrity of the connections and the ease of making adjustments as needed.

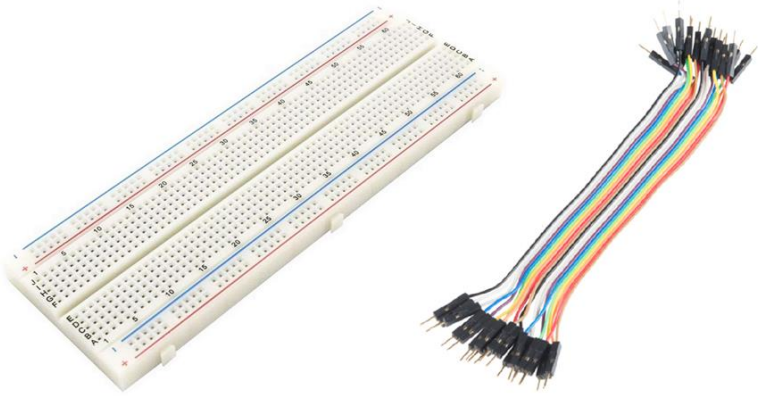


Figure 16. Breadboard and jumper wires.

This setup not only streamlined the prototyping process but also minimised the potential for errors and damage that could occur with soldered connections.

5 Software

This section outlines all the software used in the execution of this project.

5.1 Arduino IDE

This project used the Arduino IDE, a crucial tool in embedded systems and physical computing. This platform provides a software environment adjusted for programming Arduino microcontrollers. Its intuitive interface and extensive library support facilitate and accelerate the prototyping and development process.

Figure 17. Arduino logotype. [12]

The version of Arduino IDE utilised for this project was 2.3.0, publicly released on February 7, 2024. This platform offers multiple features, including code verification, compiling and automatic board detection via port recognition. Additionally, it features a sketchbook for storing the code files and a section for board management for browsing and installing board packages. A board package essentially has the instructions for compiling the code to the board. Moreover, it also features a section for library management to browse and manage the thousands of available libraries for Arduino and other microcontrollers. [12]

This IDE has a serial monitor, which facilitates viewing data streaming from the board by using, for example, the ’Serial.print()’ command, which can be used as a debugging tool. Furthermore, the Arduino IDE includes a dedicated debugging section for testing and debugging of programs. While offering more features, these represent the core functionalities of the Arduino IDE utilised in this project. [12]

5.2 AWS IoT Core

AWS IoT Core is a crucial service offered by Amazon Web Services (AWS) designed to facilitate secure and scalable communication between connected devices and the cloud. It is the foundational layer for building IoT applications, offering a robust infrastructure that handles device connectivity, message routing, and device management. AWS IoT Core empowers developers to efficiently connect and manage many devices while ensuring data privacy, integrity, and reliability through various features such as Device Gateway, Rules Engine, and Device Shadow. [6]

Figure 18. AWS logotype. [6]

The MQTT features of AWS IoT Core offer a range of capabilities, including support for message persistence and topic-based filtering. By using these features, IoT developers can arrange efficient and reliable communication between devices and cloud services while accommodating diverse use cases and deployment scenarios.

5.3 MongoDB

MongoDB is a document-oriented NoSQL database with high performance, scalability, and flexibility for handling unstructured and semi-structured data. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, allowing for easier integration with modern application development frameworks. Its schema-less design enables developers to iterate quickly and adapt to changing data requirements without predefined schemas. [13]

One of MongoDB's key features is its ability to horizontally scale across multiple nodes, making it suitable for handling large volumes of data and high-traffic applications. By distributing data across a cluster of servers, MongoDB ensures fault tolerance and high availability, reducing the risk of downtime and data loss.

Figure 19. MongoDB logotype.

Another aspect of MongoDB is its rich query language and robust aggregation framework, enabling developers to perform complex data manipulations and analytics tasks efficiently. Furthermore, MongoDB offers strong support for indexing, allowing developers to optimise query performance and ensure fast access to data.

5.4 Node-RED

Node-RED is an open-source flow-based development tool for visual programming in IoT applications. IBM Emerging Technology Services developed Node-RED, offering a browser-based flow editor that enables users to wire together devices, APIs, and online services. It utilises a visual interface where users can drag and drop nodes and connect them to create flows, eliminating the need for traditional programming. This visual approach simplifies the development process, making it accessible to many users. [14]

One of Node-RED's key features is its extensive library of pre-built nodes, which covers a wide range of functionalities, from reading sensor data to interacting with web services. These nodes can be easily installed through the Node-RED interface, allowing developers to integrate various devices and services into

their applications quickly. Additionally, Node-RED supports the creation of custom nodes using JavaScript.

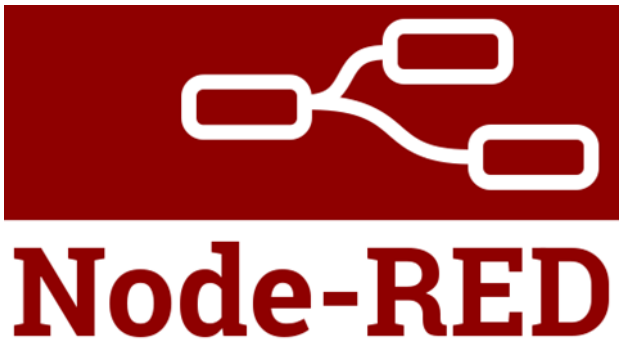


Figure 20. Node-RED logotype.

Node-RED is flexible and easy to use, making it a valuable tool for rapid prototyping and development of IoT solutions. Furthermore, it supports event-driven programming and integration with popular IoT platforms like MQTT, making it well-suited for building real-time, reactive applications.

5.5 Angular

Angular is a robust, TypeScript-based web application framework within the Node.js. It is primarily developed and maintained by the Angular Team at Google, alongside a community of individual developers and corporate contributors. This collaboration ensures continuous improvement, innovation, and support for the framework. [15]

Figure 21. Angular logotype.

Angular represents a complete architectural rewrite from its predecessor, AngularJS, aiming to address the limitations and challenges encountered in the original framework. This new iteration introduces a more modular, scalable, and

efficient approach to building web applications, leveraging modern web standards and development practices. [15]

In this project, Angular was utilised to develop a web application.

5.6 Angular Material

In this project, Angular Material was utilised to streamline and expedite the front-end development of the web application. Angular Material, a UI component library for Angular developed by the Angular team at Google, offers a comprehensive collection of reusable and accessible UI components based on the Material Design specifications. This library includes various pre-built components, such as buttons, cards, dialogues, and grids, which support the creation of responsive layouts and adaptive user interfaces. It also allows easy customisation with themes to align with the application's branding. Angular Material significantly reduces development time while ensuring a polished and professional user interface.

Figure 22. Angular Material logotype.

6 Implementation

This section delves into the implementation details. It surrounds the hardware circuit building, incorporating all used sensors and the software installation and configuration. This section also delves into board programming, including code development. Furthermore, the setup process for essential applications such as AWS IoT Core and the database is reported. Also, software integration with hardware is described in more detail, alongside the installation and configuration of the server environment.

6.1 Circuit

Below, in the Figure 23, the final hardware circuit schema is shown, highlighting the integration of sensors and the ESP32 board. Using a single ESP32 microcontroller, this implementation integrates several sensors, including one BH1750 for gathering light intensity, one BME680 for environmental data, and one soil moisture sensor for assessing soil conditions.

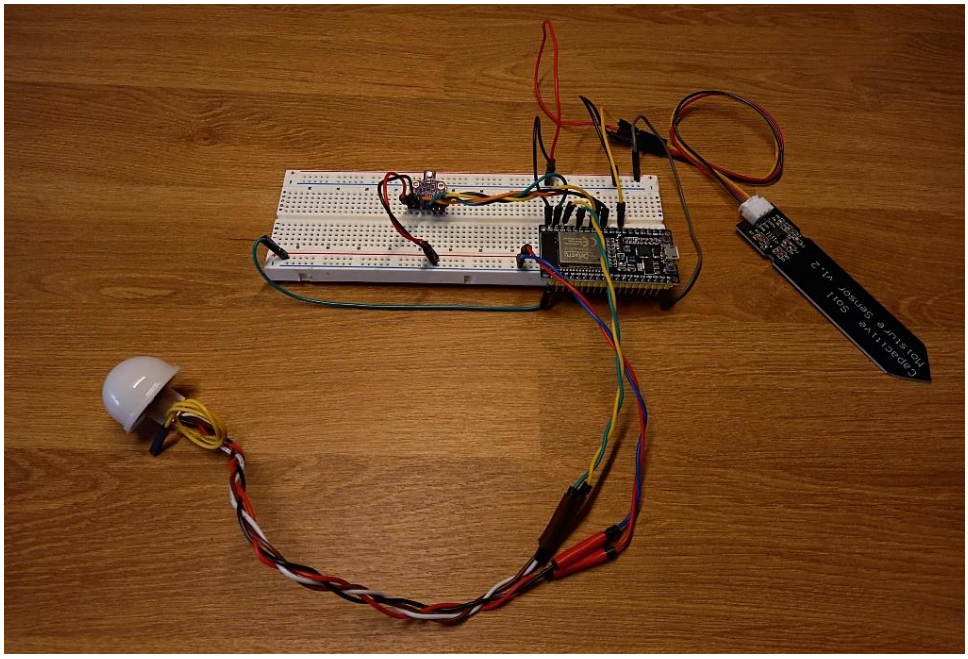


Figure 23. Hardware circuit schema.

6.1.1 Integration of the BH1750 sensor with ESP32

In the domain of illuminance measurement, the BH1750 sensor is a crucial instrument for quantifying light intensity in lux. Lux is a unit within the International System of Units designated for luminous flux per unit area. This sensor operates within the lux range of 0 to 65535; these sensors offer a comprehensive insight into ambient light levels. In contrast, lux values ranging from 10000 to 25000 are the amount of brilliance similar to full daylight without direct exposure to the sun.

Within the context of this project, the BH1750 sensor uses four of its five pins: the VCC, GND, SCL, and DAT pins. The omission of the ADDR pin derives from its exclusive purpose in configuring the sensor's I2C address, facilitating the integration of multiple sensors on a singular bus without facing conflicts. As this implementation relies on a single BH1750 sensor with an ESP32 microcontroller, the ADDR pin remains redundant and thus unutilised.

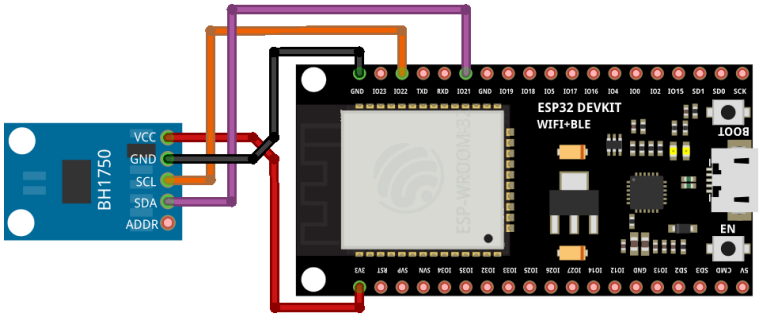


Figure 24. BH1750 sensor and ESP32 schematic.

In this implementation, the BH1750 sensor connects with the ESP32 microcontroller. Through the I2C communication protocol, the pin SCL connects to GPIO22, while the sensor's DAT pin connects with the ESP32 microcontroller GPIO21 pin. Ensuring power supply, the VCC connection is established with the 3v3 pin, while ground connectivity is maintained with the ground pin. This configuration, as illustrated in the Figure 24, facilitates the integration and data exchange between the components.

6.1.2 Integration of the BME680 sensor with ESP32

The BME680 sensor can measure environmental parameters, including gas, pressure, humidity, and temperature. It features six pins: VCC, GND, SCL, SDA, SDO, and CS. The BME680 can be integrated with the ESP32 through SPI or I2C communication protocols.

In the I2C setup, the SDO and CS pins can be omitted in the connection process. By connecting the SCL and SDA pins to GPIO22 and GPIO21 on the ESP32 microcontroller, these two pins suffice for I2C communication, leaving the other pins unutilised. However, the final implementation uses the SPI configuration, reserving the I2C pins for connecting the BH1750 sensor. [9]

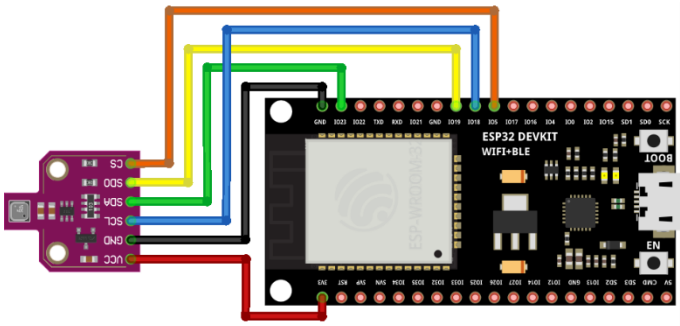


Figure 25. BME680 sensor and ESP32 schematic (SPI).

The BME680 sensor connected to the ESP32 microcontroller via SPI communication is shown above. Each pin is meticulously connected to its corresponding GPIO pin on the microcontroller: SCL to GPIO18, SDA to GPIO23, SDO to GPIO19, and CS to GPIO5. To ensure proper functionality, the VCC pin needs to be connected to the 3v3 pin, while the ground needs to be connected to the ground pin on the ESP32 microcontroller.

6.1.3 Integration of the capacitive soil moisture sensor with ESP32

Connecting the capacitive soil moisture sensor to the ESP32 is straightforward, facilitated by its three pins: GND, VCC, and AOUT. The AOUT pin is the analog output, providing the soil moisture value. To establish the connection, the VCC pin needs to be connected to the ESP32's 5V pin, ground to ground, and AOUT to GPIO4.

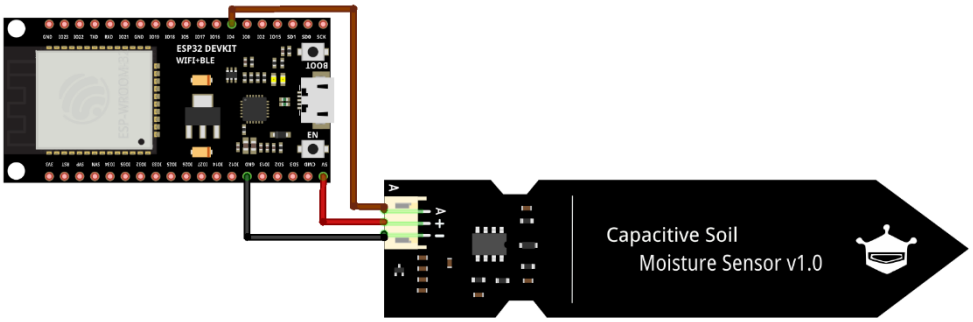


Figure 26. Capacitive soil moisture sensor and ESP32 schematic.

6.1.4 Final implementation

Below is the final integration, representing the culmination of the three previously documented implementations.

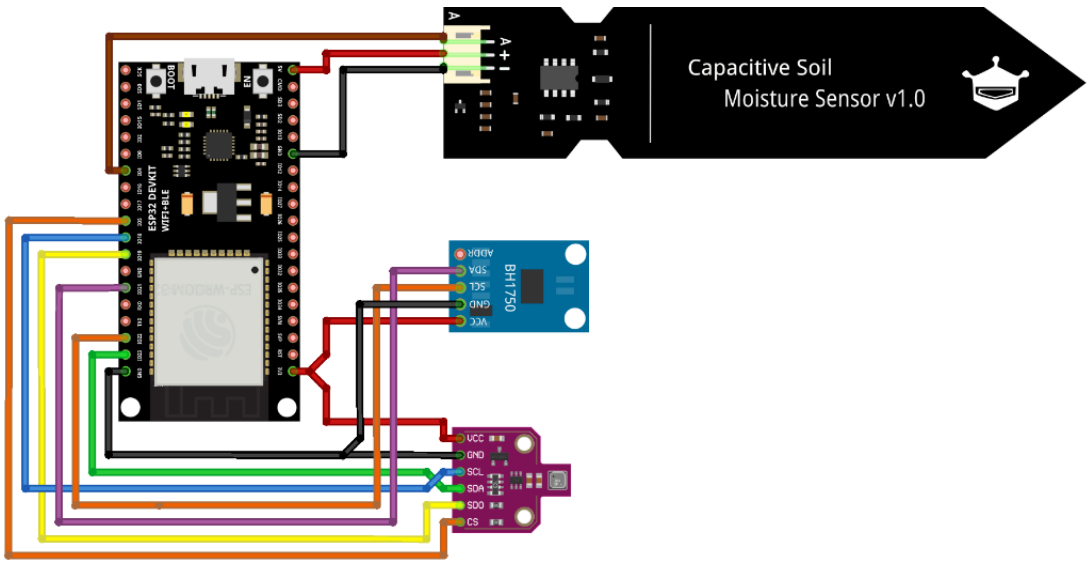


Figure 27. Final implementation schematic.

6.2 Setting up Arduino IDE

Setting up the Arduino IDE begins with downloading the executable file from their official website. Once downloaded, the installation process is straightforward, comprising just a few simple steps. Following these instructions ensures a smooth installation experience.

6.2.1 ESP32 Board Installation

To integrate the ESP32 board into the Arduino IDE, it's essential to follow a few steps. Initially, the IDE does not include the ESP32 board by default. Therefore, users need to add it manually. This can be accomplished by accessing the ’File’ option in the menu bar and selecting ’Preferences’ from the submenu. This action triggers the opening of a new dialogue box, similar to the Figure 28.

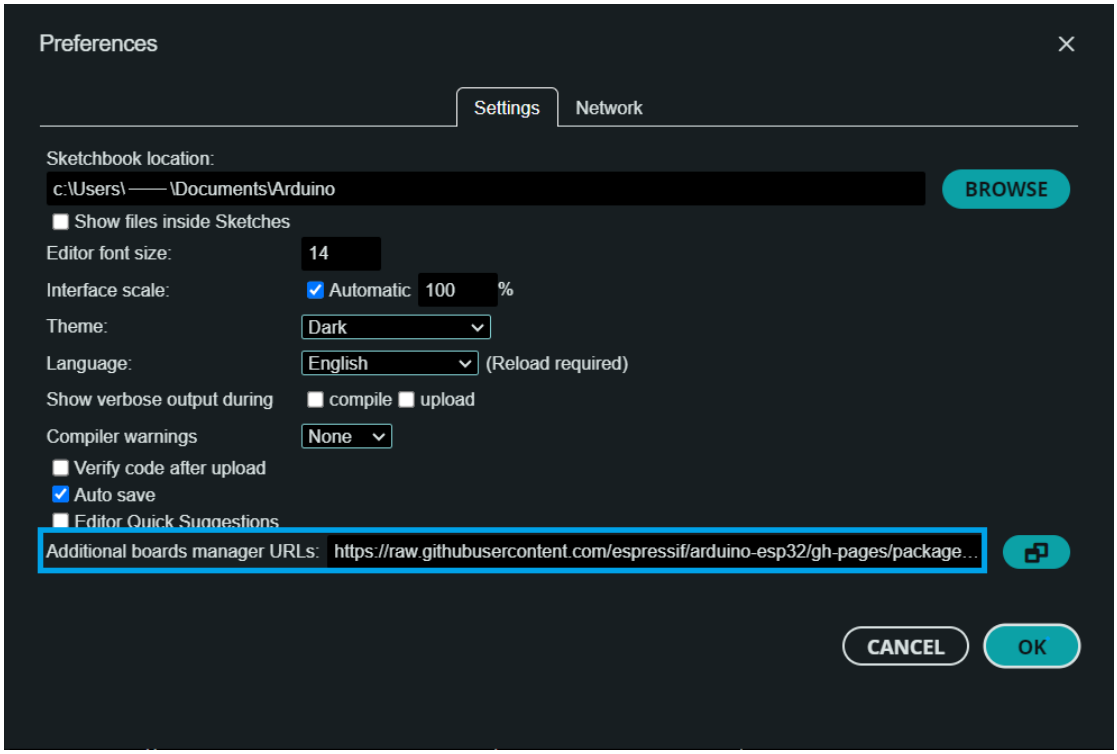


Figure 28. Arduino IDE preferences dialog.

To enhance functionality, incorporate the URL for the additional board's manager on the preferences dialogue, which is https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json. If multiple URLs are required, they can be delimitated by commas.

Subsequently, it is crucial to install the ESP32 board, and this can be done by navigating to the ‘Tools’ menu, selecting ‘Board’, and then ‘Board Manager’. Upon selecting the ‘Board Manager’, a new dialogue window will appear. Within this dialogue, it is necessary to search for ‘esp3f2’ and proceed to install the ‘ESP32’ package developed by Espressif Systems. The version utilised for this project was 2.0.16, as shown below.

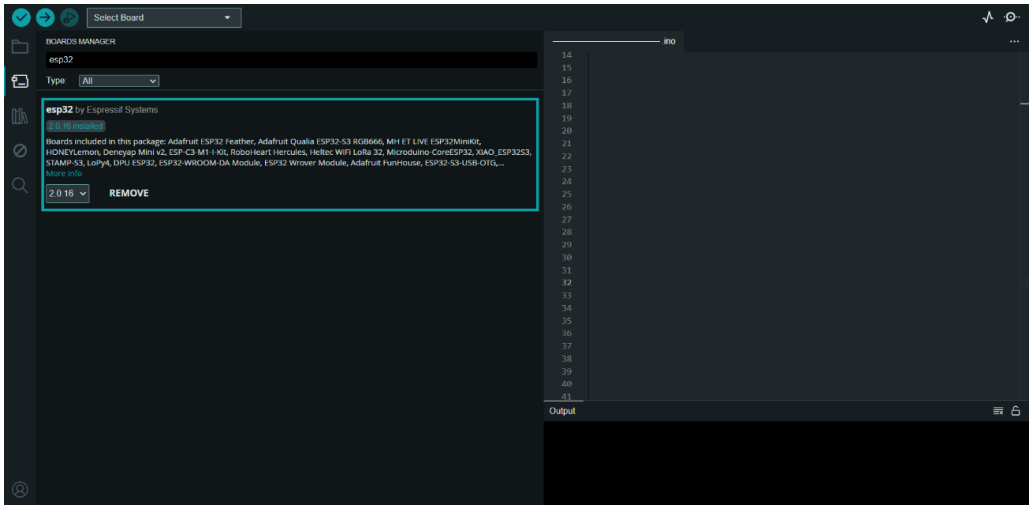


Figure 29. Arduino IDE boards manager menu.

Once the installation is complete, accessing the ‘Tools’ menu and navigating to ‘Board’ and further to ‘ESP32’ reveals a comprehensive list of all available ESP32 boards for programming. In this case, it was utilised the 'ESP32 Dev Module'.

6.2.2 ESP32 Libraries installation

In order to program the ESP32 microcontroller, installing additional libraries was crucial. In the Arduino IDE, this process is straightforward. This can be done by navigating to the library manager menu, searching for the desired library name, and then installing the desired library. The Figure 30**Error! Reference source not found.** below illustrates this process.

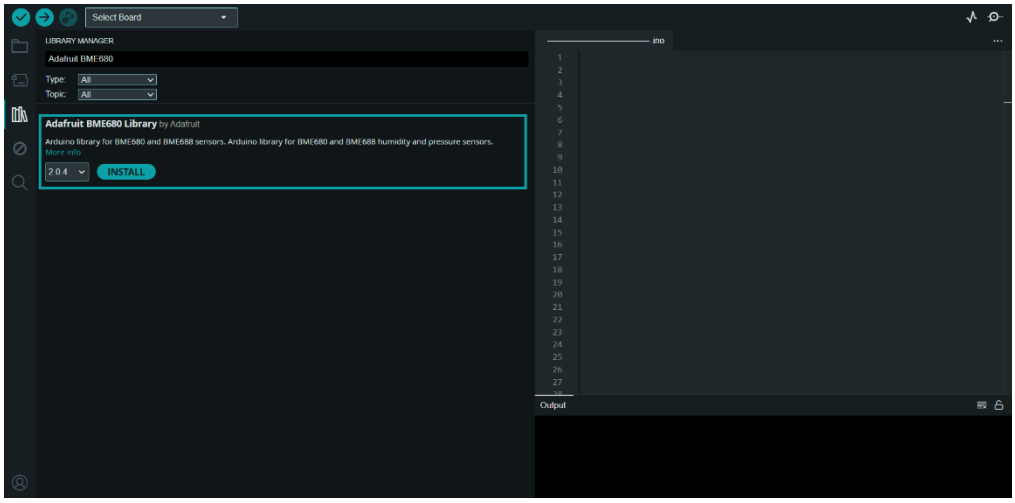


Figure 30. Arduino IDE library manager menu.

The **Adafruit BME680 Library**, developed by **Adafruit**, was installed to interface with the BME680 sensor. This library offers a straightforward API that simplifies the sensor's initialisation, configuration of its settings, and data retrieval. Using this library makes integrating the BME680 sensor into various projects easier. The version installed and used was 2.0.4. [15]

The **Adafruit Unified Sensor** library, developed by **Adafruit**, was installed. This library offers a standardised interface for interacting with various sensor types. It provides a common API across different sensor drivers, simplifying the integration of multiple sensors into projects. This allows developers to avoid learning different interfaces for each sensor. The version installed and utilised was 1.1.14. [16]

The **BH1750** library, developed by Christopher Laws, was also installed. This library is specifically designed to interface with the BH1750 ambient light sensor. The version installed and utilised was 1.3.0. [17]

The **ArduinoJson** library, developed by **Benoit Blanchon**, was installed. This powerful and efficient JSON library is specifically designed for embedded systems, particularly platforms like Arduino. It offers an easy-to-use API for parsing, creating, and manipulating JSON documents. Optimised for low memory usage and high performance, ArduinoJson is ideal for microcontrollers with limited resources. The version installed and utilised was 7.0.4. [18]

The last library to install was the PubSubClient, created by Nick O'Leary. This popular and lightweight library facilitates MQTT communication on Arduino and compatible boards. The version installed and utilised was 2.8. [19]

6.2.3 Programming the ESP32 board

The code for gathering sensor data is straightforward. The most complex aspect is configuring the board to connect and communicate with the AWS service. The board primarily needs to know two things: where to publish the collected data and where to subscribe to AWS messages. Additionally, the ESP32 requires the endpoint and port for the AWS service, the certificate CA, the device certificate, and the private key. These five parameters are crucial for secure communication and must remain confidential. The board also needs the thing name, Wi-Fi SSID, and Wi-Fi password to establish a connection.

All the project code is available in this GitHub repository: https://github.com/serac01/smart-agriculture. The repository does not include sensitive information, but a sample file with placeholders for the necessary attributes is provided for security reasons.

6.3 Setting up the Raspberry

This project utilised the Raspberry Pi 4 model B with 4GB. Initially, the microSD card must be configured, requiring the download of the Raspberry Pi Imager from the official website. Upon opening the application, the device should be selected; in this case, the Raspberry Pi 4 option was chosen. Subsequently, the operating system should be chosen; in this project, the ’Ubuntu desktop 24.04 LTS (64-BIT)’ was used. The final step involves choosing the storage for writing the operating system. [20]

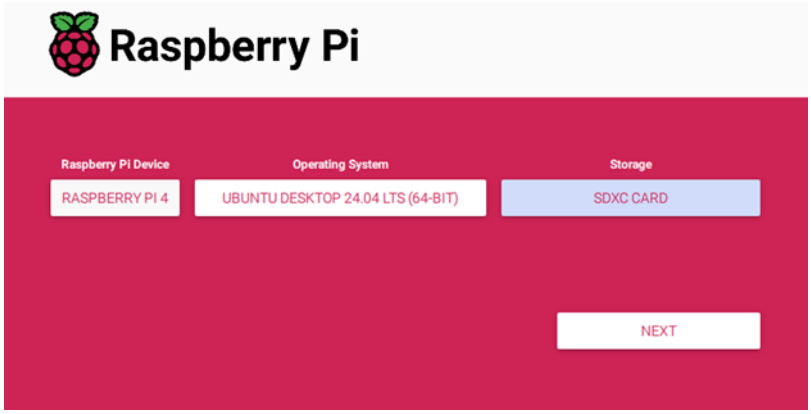


Figure 31. Raspberry Pi Imager settings.

After these selections, the operating system is written onto the microSD card. The Raspberry Pi can be powered on upon completion of the writing process. The microSD card is inserted first, followed by connecting the Raspberry Pi to power, connecting the monitor via micro-HDMI, and connecting the keyboard and mouse.

Upon powering the Raspberry Pi, the operating system installation can be initiated. This process is straightforward. Initially, the system language needs to be selected, followed by the keyboard layout and, subsequently, the current location. The next step is the user information, including the name, computer name, username, and password. After the user's configuration, the system goes to installation, which requires some time. Upon completion of the installation process, the setup is finalised.

6.4 Setting up the AWS IoT Core

The AWS IoT Core was designed to facilitate secure and scalable communication between connected devices and the cloud. Accessing this feature requires the creation of an Amazon Web Services account. The AWS IoT Core can be accessed by navigating to the AWS management console's main page, where the option for AWS IoT Core is selected from the Services menu at the top.

In the thesis, utilising AWS IoT Core to execute MQTT requests will streamline communication between the ESP32 devices and the server, represented by the Raspberry Pi. The MQTT functionalities provided by AWS encompass a variety of capabilities, such as message persistence and topic-based filtering. [6]

6.4.1 Creating a Thing

To initiate the system's setup on the AWS IoT Core, creating a **thing** is imperative. An IoT ’thing’ is a virtual representation and repository of the physical device within the cloud infrastructure. To access the 'thing' option, it is needed to go under the manage tab in the left menu. Creating the 'thing' begins with selecting the quantity of 'things' to be generated. In this project, only one 'thing' was created. Following this, the specification of 'thing' properties ensues, commencing with the assignment of the thing name. In this case, ’*SMART_AGRICULTURE’* was designated as the name, with the remaining configurations left as default. The subsequent step demands the configuration of the device certificate. For this project, it was chosen to auto-generate a new certificate following AWS recommendations. Following this, a policy should be attached to the certificate. Upon the association of the policy, the certificates and keys will be available for download. It is crucial that the device certificate, along with the public and private keys, as well as the Root CA certificates, be securely saved on the computer.

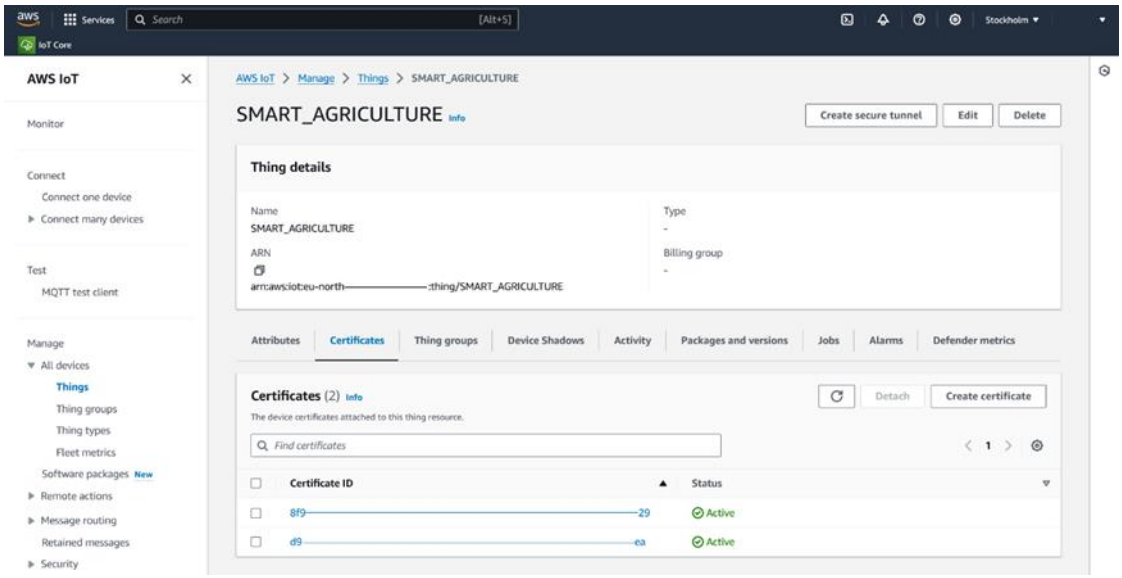


Figure 32. AWS IoT Core - Thing.

The figure provided above represents the thing currently utilised in this thesis. It showcases the name of the thing, its Amazon Resource Name (ARN), and the active certificates associated with it.

6.4.2 Creating a Policy

A **policy** defines a set of authorised actions. To create an action, a name must be specified. In this thesis, one policy named ’SMART_AGRICULTURE_POLICY’ was defined. A policy can encompass several statements, each outlining the types of actions that a resource can perform. Each statement consists of an action, a resource ARN, and an effect, which can be allowed or denied. For this project, the following statements were implemented:

For this project, were implemented the following statements:

- **IoT:Connect:** Allows the client named ’SMART_AGRICULTURE’ to connect to AWS IoT Core.

- **IoT:Connect:** Allows the client named ’SMART_AGRICULTURE_NODE_RED’ to connect to AWS IoT Core.
- **Iot:Publish:** Authorizes publishing messages to the topic ’smart-agriculture/pub’.
- **IoT:Publish:** Authorizes publishing messages to the topic ’smart-agriculture/sub’.
- **Iot:Subscribe:** Enables clients to subscribe to messages from the ’smart-agriculture/pub’ topic filter.
- **Iot:Subscribe:** Enables clients to subscribe to messages from the ’smart-agriculture/sub’ topic filter.
- **Iot:Receive:** Allows receiving messages from the topic ’smart-agriculture/pub’.
- **Iot:Receive:** Allows receiving messages from the topic ’smart-agriculture/sub’.

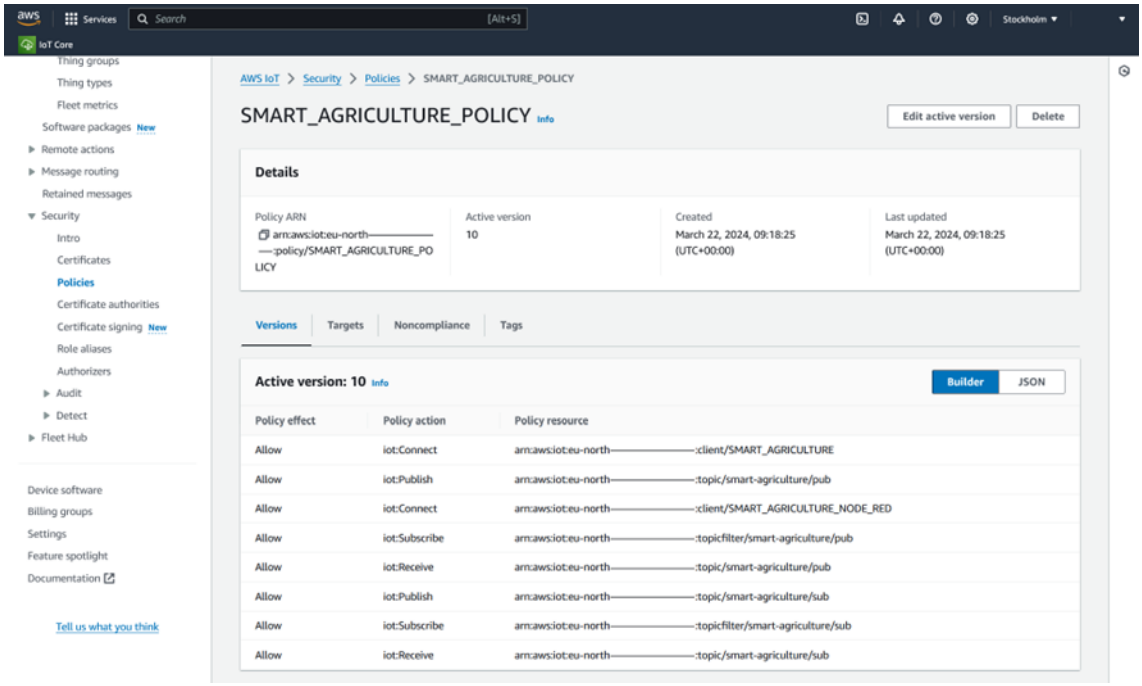


Figure 33. AWS IoT Core - Policy.

The figure provided above represents the policy currently utilised within this thesis. It showcases the policy's name and all the statements implemented.

6.4.3 Testing

AWS IoT Core includes a feature for monitoring MQTT messages. This can be accomplished by subscribing to or publishing to a topic. To view data being published by the client, the ESP32 device, a subscription to the topic `smart-agriculture/pub` is required. This feature also offers additional settings, such as specifying the number of messages to retain, the format of the MQTT payload, and the message delivery method, whether at most once or at least once.

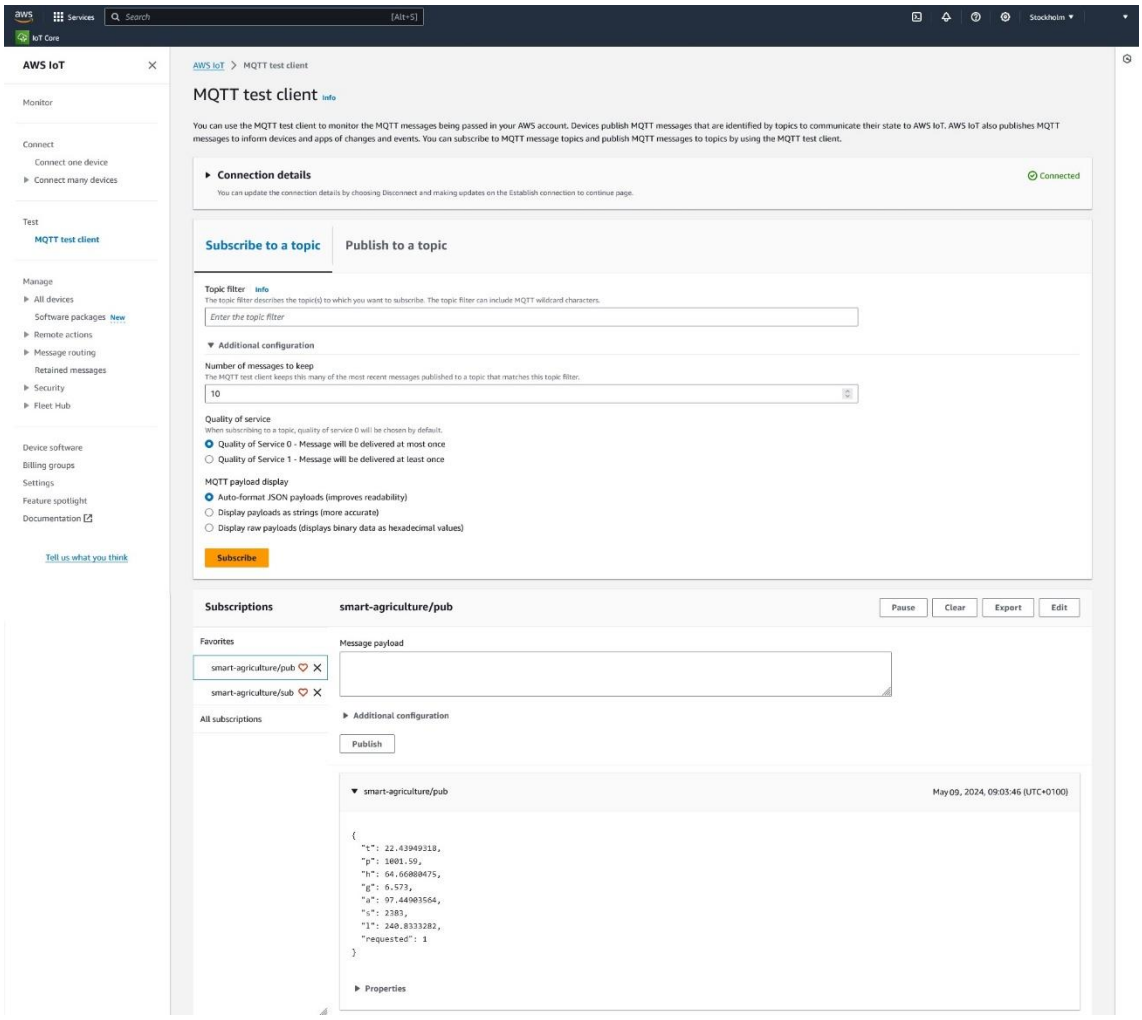


Figure 34. AWS IoT Core - Subscribing to a topic.

Additionally, a topic for publishing data is implemented. Data can be published to the topic ’smart-agriculture/sub’, which the client, the ESP32 device, is configured to receive. When a message with the content ’send_sensor_data’ is

sent to this topic, the device collects the current sensor data and publishes it to the ’smart-agriculture/pub’ topic.

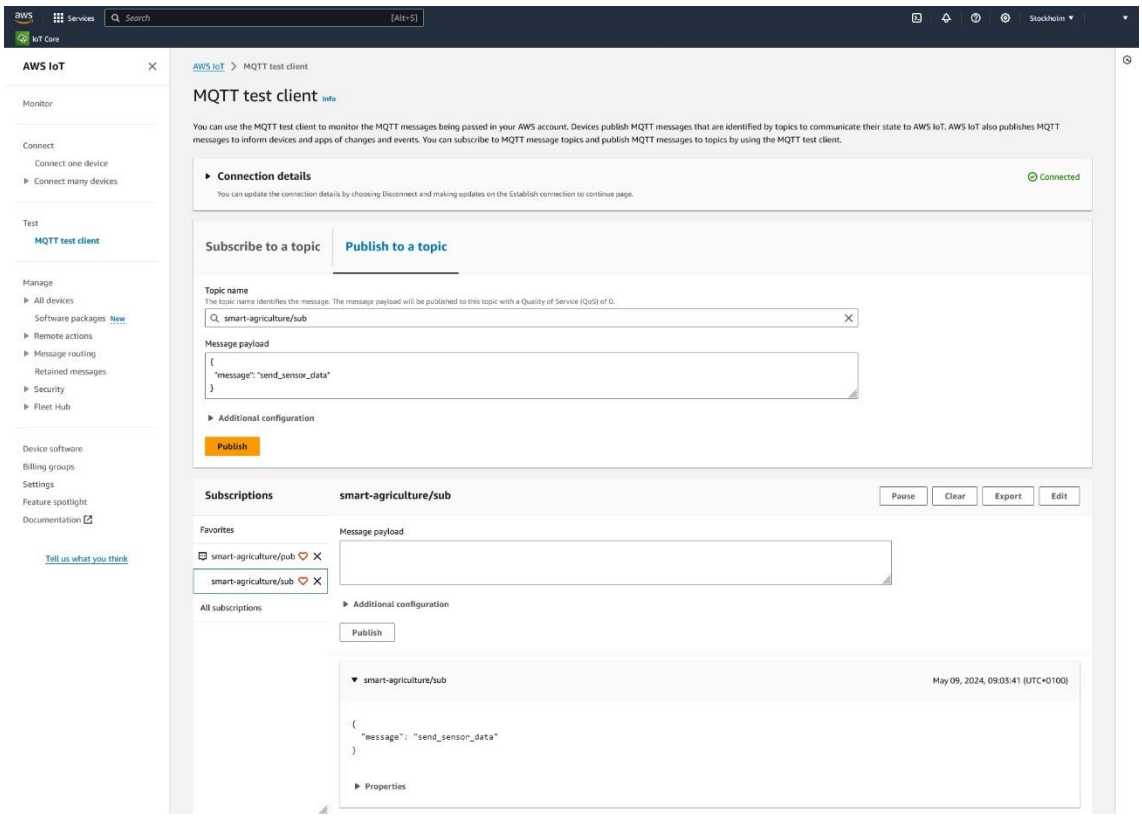


Figure 35. AWS IoT Core - Publishing to a topic.

6.5 Setting up Node-RED

Node-RED is an open-source flow-based development tool for visual programming in IoT applications. To install Node-RED, the following command must be executed: ’**npm install -g --unsafe-perm node-red**’. After installation, Node-RED can be launched by writing ’**node-red**’ on the command line. This will open Node-RED in a browser at the localhost address with the specified port number. The flow depicted in **Error! Reference source not found.** was implemented for this project. This implementation includes several nodes: MQTT In, Change, Function, MongoDB3 In, and Debug nodes.

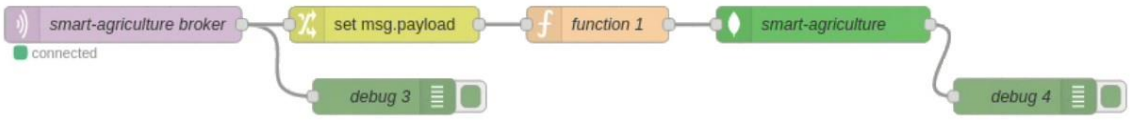


Figure 36. Node-RED implemented flow.

The first node is an MQTT, which is subscribed to the smart-agriculture/pub topic to get the data when the client publishes it. For this node access to the AWS IoT Core, it was necessary to create a server. The server has to contain the endpoint name, obtained from the AWS IoT Core settings, and the port number that, by default, is 8883. Then, it was defined that it should use TLS, and the protocol for the communication is the MQTT V3.1 (legacy). The last thing to have the server configured is the client ID, which in this case is ’SMART_AGRICULTURE_NODE_RED’, which was explicitly created to Allow the client to connect to AWS IoT Core. After the server configuration, it needed to configure the MQTT in the node that should be using this server that was created; the action should be subscribed to a single topic and the topic, as said earlier in this case, is smart-agriculture/pub, and for output it should expect a parsed JSON object. The complete configuration is presented in the image below.

The first node is an MQTT subscribed to the smart-agriculture/pub topic to receive data when published by the client. A server was created to enable access to the AWS IoT Core from this node. The server includes the endpoint name, obtained from the AWS IoT Core settings, and the default port number, which in this case was 8883. The MQTT V3.1 (legacy) protocol was used for communication. The final requirement for the server configuration is the client ID, which in this case is ’SMART_AGRICULTURE_NODE_RED’, created explicitly during the AWS IoT Core policy configuration to allow the client to connect to the AWS service.

After the server setup, the MQTT input node configuration was necessary. This node was set to use the created server, subscribe to the single topic smart-

agriculture/pub, and output a parsed JSON object. The complete configuration is shown in the Figure 37 below.

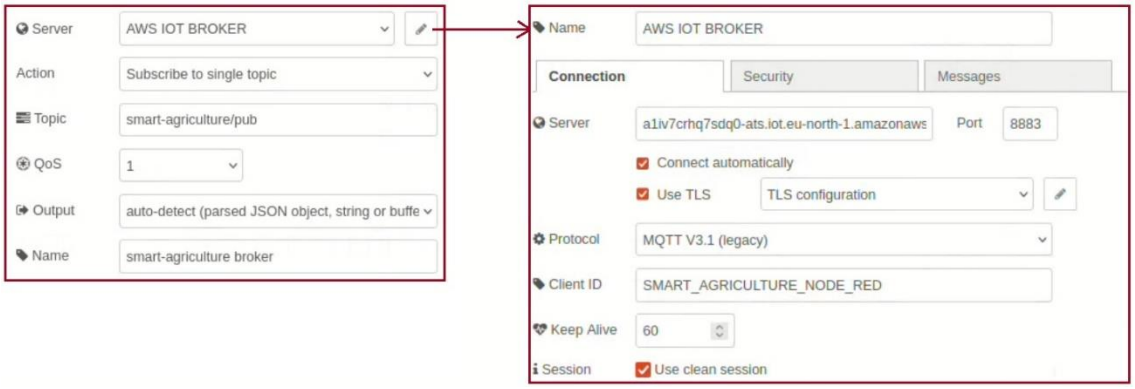


Figure 37. Node-RED - MQTT in configuration.

After the MQTT in node, a debug node and a change node were used. The debug node was utilised for debugging purposes, displaying the data retrieved from the subscription to the AWS topic. The change node was employed to convert the received data into a specific structure, defined as a JSON expression. The implemented structure is shown in the Figure 38.

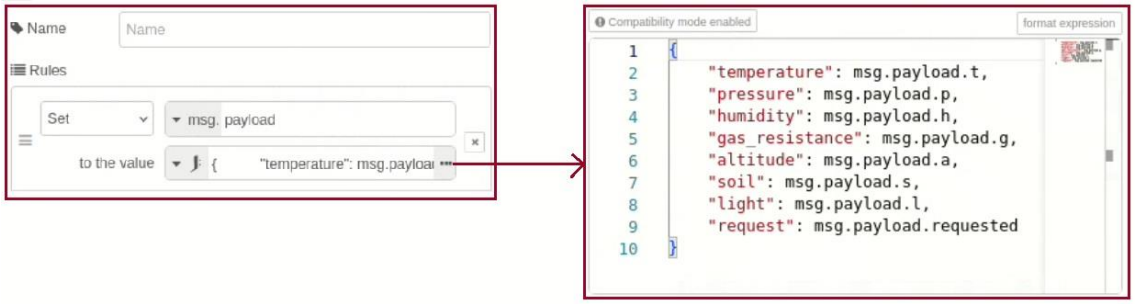


Figure 38. Node-RED - change node configuration.

A function node connected to the change node retrieves the current time and incorporates this variable into the existing structure. The code utilised in this function is presented below.

```
msg.payload.timestamp = new Date();
return msg;
```

Connected after the function node is the MongoDB3 in node. This node is included to store the pre-prepared structure in the MongoDB database. The

node contains a server where the URL for posting data needs to be defined, which can be found in the MongoDB database settings. In this case, the service is external, the operation performed is an insert, and the collection name is 'smart-agriculture'. The settings for this node are presented in the image below.

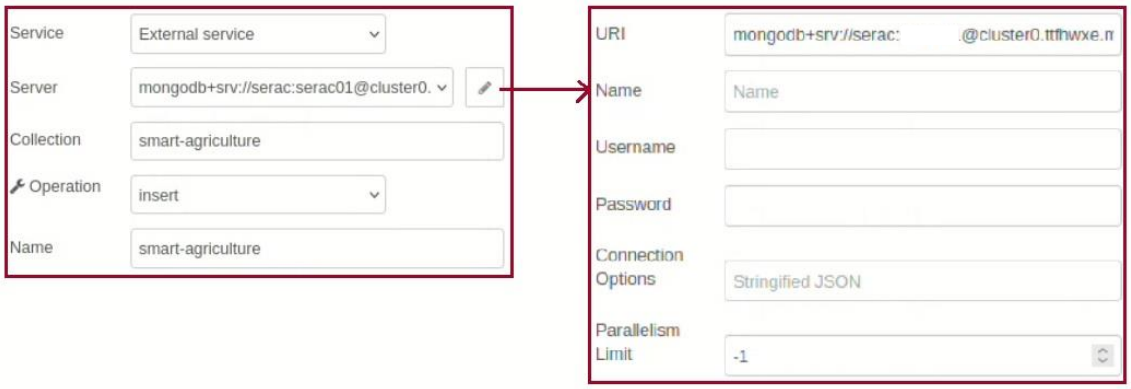


Figure 39. Node-RED - MongoDB3 in node configuration.

After the MongoDB3 node in Node.js, a debug node is used to verify the operation's successful completion.

6.6 Setting up the database

In this thesis, MongoDB was chosen as the database. MongoDB is a document-oriented NoSQL database with high performance, scalability, and flexibility for handling unstructured and semi-structured data. To create a database on MongoDB, an account must first be created. Subsequently, a project is needed, and only after that can a database be created. For this project, a shared cluster was created, recommended for learning and exploring MongoDB in a cloud environment and is also the only option available for free. This cluster offers 512MB of storage and has shared RAM and CPU. AWS was selected as the cloud provider, with the location set to Paris (eu-west-3). The cluster name was left as the default, Cluster0.

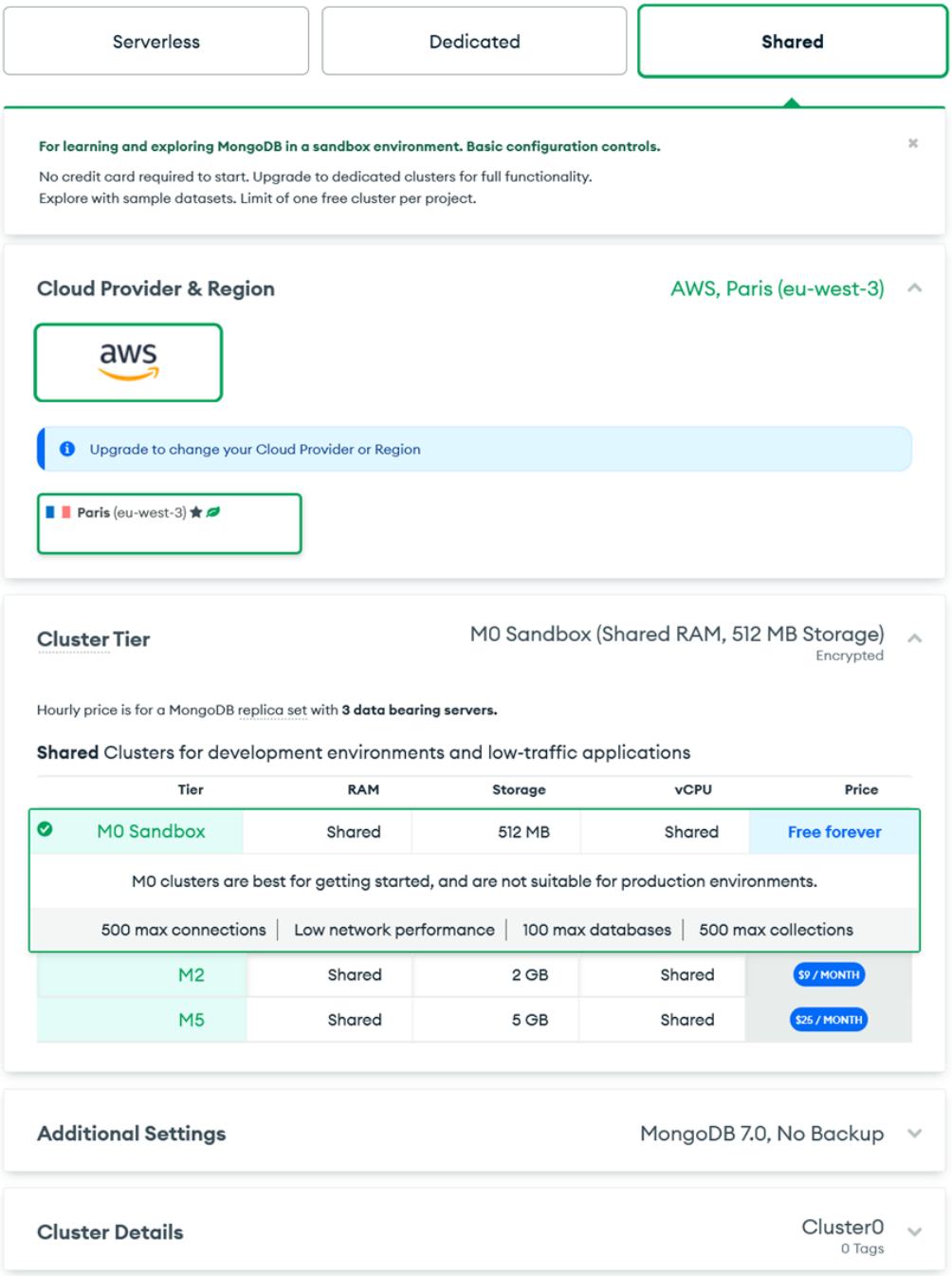


Figure 40. MongoDB cluster configuration.

A cluster user was then created by setting a username and password. The network configuration was left as default, with 'My local environment' chosen as the network that can access the cluster. The driver method was selected to connect to the cluster, and the provided instructions were followed. By doing

that, MongoDB created a link so that Node-RED could establish a connection to the database.

A database can now be created. In this case, a database named ’test’ has been made, with a collection inside called ’smart-agriculture’ where all sensor readings are inserted.

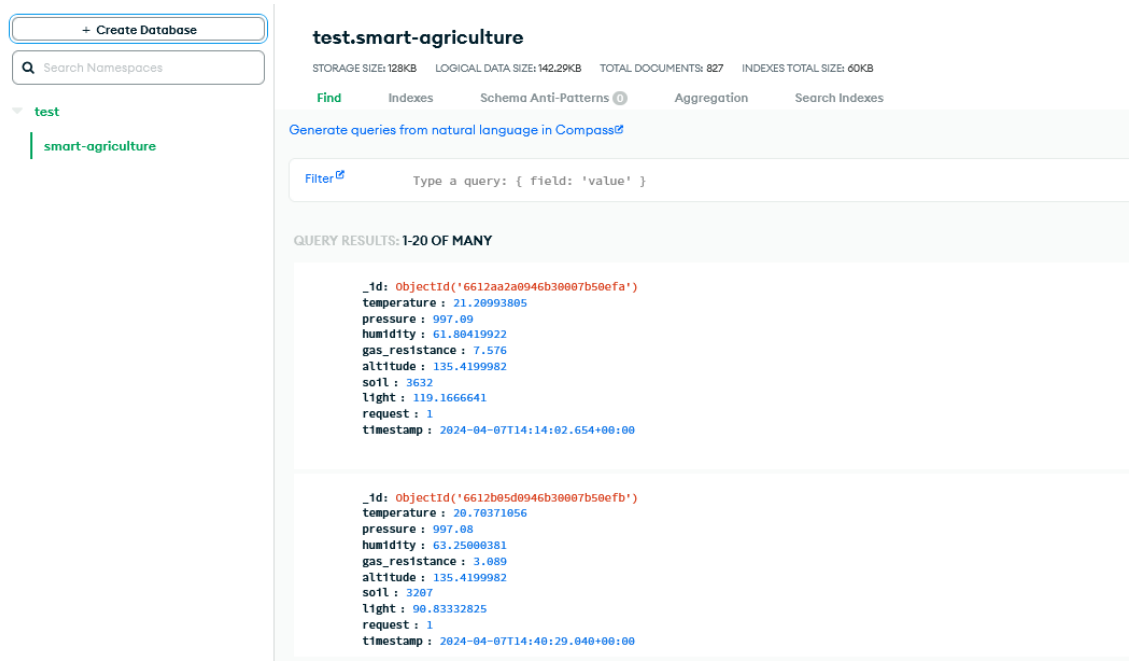


Figure 41. MongoDB database entries.

As shown in the Figure 41, several entries are displayed, each having the same structure. These entries include the ID, temperature, pressure, humidity, gas resistance, altitude, soil moisture, light, a boolean indicating whether the data was requested or automatically collected, and the timestamp of the reading.

6.7 System overview

To ensure the integrity and functionality of the developed hardware system within a greenhouse environment, it was encased in a protective box equipped with strategically placed air inlets to facilitate ventilation. This design choice serves a dual purpose: safeguarding the hardware from potential environmental hazards while allowing optimal exposure to external parameters necessary for accurate data collection. The air inlets ensure the system receives adequate airflow, preventing overheating and ensuring reliable operation over extended periods.

Figure 42. Hardware developed case.

Implementing this system within a greenhouse environment gives farmers greater control and insights into their operations. To achieve this, an integrated

application was developed, comprising two distinct projects: a backend project and a frontend project.

6.7.1 Backend project

The backend project is crucial as it connects directly to the database, ensuring secure and efficient data management. It includes several endpoints that the frontend application can use. This architecture enhances security by not distributing the backend to external users, mitigating potential attacks on the backend, database, and the overall system. The primary endpoints developed for the backend include:

- **Retrieve All Sensor Data:** This endpoint fetches comprehensive data from all the sensors within the greenhouse.
- **Delete Specific Timestamp Data:** This endpoint allows the deletion of specific readings based on their timestamp and corresponding sensor measurements.
- **Data Retrieval Request:** This endpoint enables the system to retrieve data upon request, facilitating on-demand data access.

6.7.2 Frontend project

The front-end project has a user-friendly interface that allows farmers to interact with the system easily. It features various graphs and visualisations to help users analyse the collected data effectively. The intuitive design of the front end ensures that users can quickly understand and respond to the information provided by the system.

Below are images demonstrating the implementation and functionality of the frontend interface.

Figure 43. Application - dashboard.

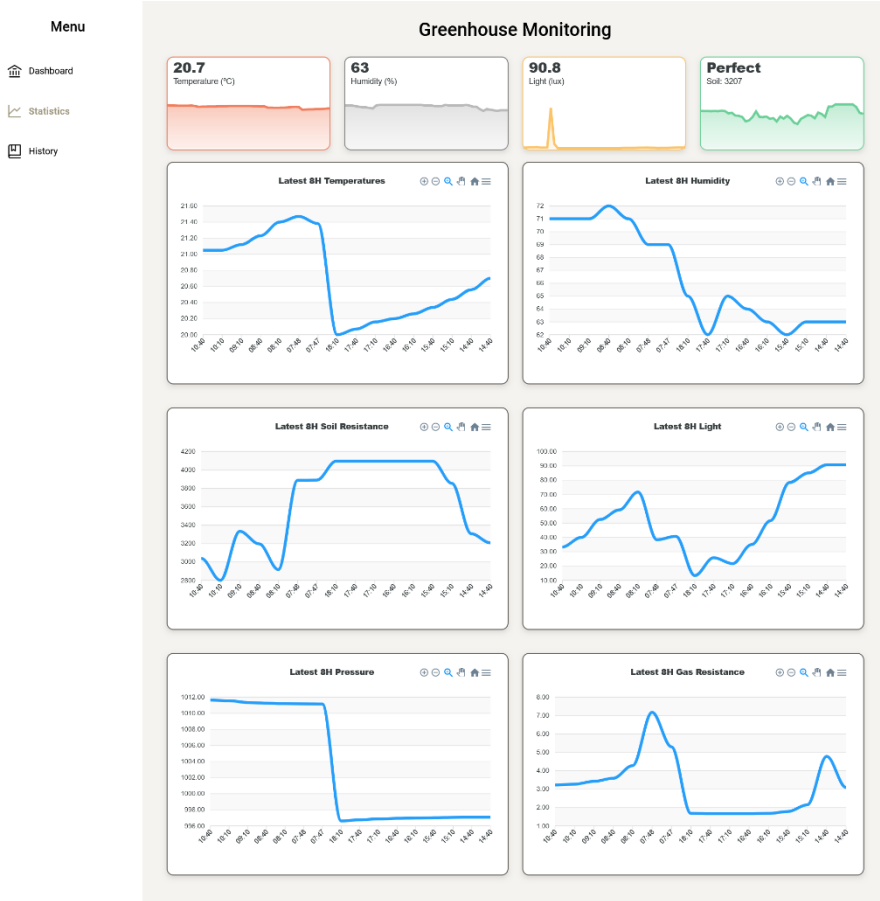
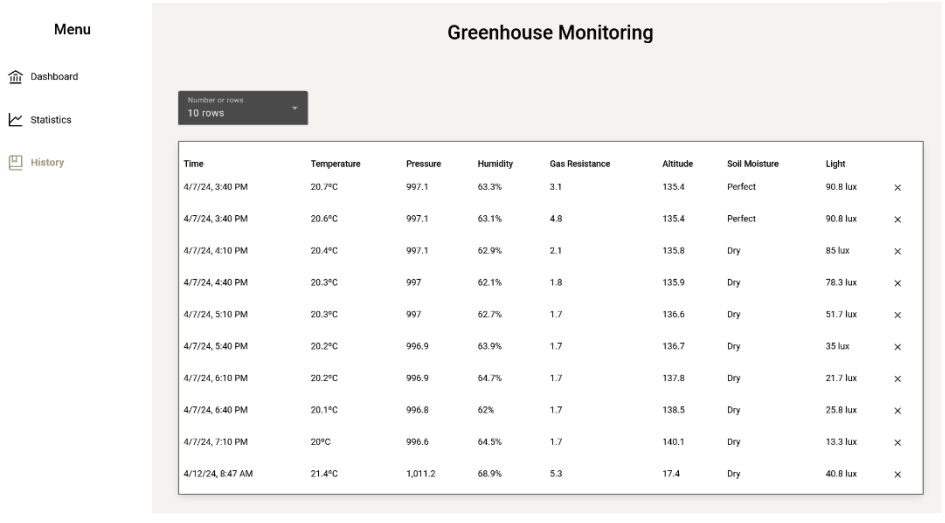


Figure 45. Application - statistics.



Time	Temperature	Pressure	Humidity	Gas Resistance	Altitude	Soil Moisture	Light	
4/7/24, 3:40 PM	20.7°C	997.1	63.3%	3.1	135.4	Perfect	90.8 lux	×
4/7/24, 3:40 PM	20.6°C	997.1	63.1%	4.8	135.4	Perfect	90.8 lux	×
4/7/24, 4:10 PM	20.4°C	997.1	62.9%	2.1	135.8	Dry	85 lux	×
4/7/24, 4:40 PM	20.3°C	997	62.1%	1.8	135.9	Dry	78.3 lux	×
4/7/24, 5:10 PM	20.3°C	997	62.7%	1.7	136.6	Dry	51.7 lux	×
4/7/24, 5:40 PM	20.2°C	996.9	63.9%	1.7	136.7	Dry	35 lux	×
4/7/24, 6:10 PM	20.2°C	996.9	64.7%	1.7	137.8	Dry	21.7 lux	×
4/7/24, 6:40 PM	20.1°C	996.8	62%	1.7	138.5	Dry	25.8 lux	×
4/7/24, 7:10 PM	20°C	996.6	64.5%	1.7	140.1	Dry	13.3 lux	×
4/12/24, 8:47 AM	21.4°C	1,011.2	68.9%	5.3	17.4	Dry	40.8 lux	×

Figure 44. Application - history.

The dual-project architecture, with its clear separation of backend and frontend functions, exemplifies a robust approach to developing secure and user-friendly applications for modern agricultural needs.

All the project code is available in this GitHub repository: https://github.com/serac01/smart-agriculture.

7 Testing

This chapter primarily focuses on the system's testing process. Several tests were conducted during the testing, and they are described herein.

7.1 System Functionality and Resilience Evaluation

The initial test involved evaluating the system's functionality and its preparation for storage in a greenhouse. During this process, the system response to vibration was tested to determine if the sensors and microcontroller would become dislodged due to shaking, potentially ceasing operation or producing invalid values. Adjustments to the programming were necessary during this phase; for instance, the system would halt operation upon a power loss, or if WiFi connectivity failed, the data read would not be saved in the database. These issues were addressed through programming enhancements. If the system experiences a power loss, it can restart and function correctly. In the event of WiFi unavailability, the read values are stored until reconnection, after which they are successfully posted using the MQTT communication protocol.

7.2 Long-Term Greenhouse Data Integrity

The system was stored inside a greenhouse for over 45 days. During this period, it operated flawlessly without missing data and the need for maintenance. Additionally, all requests for sensor data were successfully answered throughout this time.

7.3 Sensors accuracy

Three devices were used to measure the accuracy of the collected data: a digital illuminometer, a 3-in-1 soil pH meter/soil moisture meter/light meter, and a digital thermometer for measuring temperature and humidity.

Figure 46. Illuminometer, soil moisture meter and thermometer

The accuracy was measured by collecting data from the sensors and simultaneously gathering current values from these devices. Below is a table with some samples taken from one day of testing.

Table 2. Device measurement VS sensor measurement.

	Light	**Humidity**	**Temperature**	**Soil Moisture**
(07 AM) Device measurement	55 lux	11.2ºC	84%	Dry
(07 AM) Sensor measurement	63 lux	11.7ºC	81%	Dry

(09 AM) Device measurement	157 lux	16.2ºC	71%	Wet
(09 AM) Sensor measurement	170 lux	15.4ºC	69%	Wet
(11 AM) Device measurement	213 lux	17ºC	67%	Perfect
(11 AM) Sensor measurement	221 lux	17.3ºC	66%	Perfect
(01 PM) Device measurement	357 lux	23.7ºC	46%	Perfect
(01 PM) Sensor measurement	379 lux	23.6ºC	43%	Perfect
(05 PM) Device measurement	268 lux	24.9ºC	40%	Wet
(05 PM) Sensor measurement	257 lux	25.2ºC	40%	Wet
(09 PM) Device measurement	10 lux	16.1ºC	69%	Perfect
(09 PM) Sensor measurement	0 lux	16.3ºC	70%	Perfect

In the Table 2, the results demonstrate that although the values deviated slightly from the measurements, they are remarkably close.

7.4 Testing conclusions

Despite the issues identified during testing, the system's overall performance was satisfactory. Various metrics were evaluated, and it was determined that the system met the necessary performance standards.

8 Conclusions and future work

The thesis project successfully resulted in a system for greenhouse monitoring based on sensor readings. The initial objectives were to create a robust hardware architecture, server-ESP32 Communication, greenhouse integration, an interface for data visualisation and analysis, data storage, and a prototype of an autonomous greenhouse. Although not all aspects were covered due to time constraints and budget limitations, the project is notably effective and demonstrates significant potential for implementation, as evidenced by the results.

All required features were completed as initially planned. Although the autonomous greenhouse prototype requirement could not be implemented as expected, the current progress suggests that its future implementation would be straightforward and practical.

Overall, the project can be considered a success.

Future advancements for this project would need to focus on three key areas. Firstly, the autonomous greenhouse prototype would need to be developed to represent the culmination of the project's efforts. Secondly, improvements to the web application will be implemented to enhance functionality and user experience. Lastly, additional sensors will be integrated to increase the system's realism and efficiency. These enhancements are expected to innovate further and amplify the impact of this thesis.

9 References

[1] ”MoSCoW Prioritization,” ProductPlan, [Online]. Available: https://www.productplan.com/glossary/moscow-prioritization/. [Haettu 22 April 2023].

[2] ”Easy Raspberry Pi IoT Server,” Learn Embedded Systems, 6 November 6. [Online]. Available: https://learnembeddedsystems.co.uk/easy-raspberry-pi-iot-server. [Haettu 19 March 2024].

[3] ”What is ngrok? How to use ngrok tunneling?,” PubNub, [Online]. Available: https://www.pubnub.com/guides/what-is-ngrok/. [Haettu 25 March 2024].

[4] ”Quickstart,” ngrok, [Online]. Available: https://ngrok.com/docs/getting-started/. [Haettu 25 March 2024].

[5] ”Subnet routers and traffic relay nodes,” Tailscale, [Online]. Available: https://tailscale.com/kb/1019/subnets. [Haettu 01 April 2024].

[6] ”AWS IoT,” AWS, [Online]. Available: https://aws.amazon.com/iot/. [Haettu 03 April 2024].

[7] AZ-Delivery, ”ESP32 Dev Kit C Unlogged compatible with Arduino,” AZ-Delivery, [Online]. Available: https://www.az-delivery.de/en/products/esp32-dev-kit-c-unverlotet. [Haettu 11 April 2024].

[8] ”Raspberry Pi,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi. [Haettu 2024 April 11].

[9] ”ESP32: BME680 Environmental Sensor using Arduino IDE,” Random Nerd Tutorials, [Online]. Available: https://randomnerdtutorials.com/esp32-bme680-sensor-arduino/. [Haettu 13 April 2024].

[10] ”Interfacing Capacitive Soil Moisture Sensor with Arduino,” Last Minute Engineers, [Online]. Available: https://lastminuteengineers.com/capacitive-soil-moisture-sensor-arduino/. [Haettu 13 April 2024].

[11] ”ESP32 with BH1750 Ambient Light Sensor,” Random Nerd Tutorials, [Online]. Available: https://randomnerdtutorials.com/esp32-bh1750-ambient-light-sensor/. [Haettu 14 April 2024].

[12] ”Getting Started with Arduino IDE 2,” Arduino, [Online]. Available: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/. [Haettu 25 April 2024].

[13] ”MongoDB,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/MongoDB. [Haettu 25 April 2024].

[14] ”Node-RED,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Node-RED. [Haettu 25 April 2024].

[15] ”Adafruit_BME680,” Github, [Online]. Available: https://github.com/adafruit/Adafruit_BME680. [Haettu 28 April 2024].

[16] ”Adafruit_Sensor,” Github, [Online]. Available: https://github.com/adafruit/Adafruit_Sensor. [Haettu 28 April 2024].

[17] ”BH1750,” Github, [Online]. Available: https://github.com/claws/BH1750. [Haettu 28 April 2024].

[18] ”ArduinoJson,” ArduinoJson, [Online]. Available: https://arduinojson.org/?utm_source=meta&utm_medium=library.properties. [Haettu 28 April 2024].

[19] ”Arduino Client for MQTT,” Knolleary, [Online]. Available: https://pubsubclient.knolleary.net/. [Haettu 28 April 2024].

[20] ”Setting up your Raspberry Pi,” Raspberry Pi Foundation, [Online]. Available: https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up. [Haettu 29 April 2024].